Improving performance and security of indirect memory references on speculative execution machines

by

Vladimir Kiriansky

B.Sc., Massachusetts Institute of Technology (2002)
M.Eng., Massachusetts Institute of Technology (2003)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2019

Author . . . .
Department of Electrical Engineering and Computer Science
May 23, 2019

Certified by . . . .
Saman Amarasinghe
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by . . . .
Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

Improving performance and security of indirect memory references on speculative execution machines

by

Vladimir Kiriansky

Submitted to the Department of Electrical Engineering and Computer Science on May 23, 2019, in partial fulfillment of the requirements for the degree of Doctor of Philosophy

Abstract

Indirect memory references hobble efficient and secure execution on current processor architectures. Traditional hardware techniques such as caches and speculative execution are ineffective on demanding workloads, such as in-memory databases, machine learning, and graph analytics. While terabytes of DRAM are now available in public cloud machines, indirect memory references in large working sets often incur the full penalty of a random DRAM access. Furthermore, caches and speculative execution enable the recently discovered Spectre family of side-channel attacks, which allow untrusted neighbors in a public cloud to steal secrets. In this thesis, we introduce complementary software and hardware techniques to improve the performance of caches and speculative execution, and to block the largest attack class with low overhead.

MILK is our C++ extension to improve data cache locality. Milk's programming model preserves parallel program semantics and maps well to the Bulk-Synchronous Parallel (BSP) theoretical model. Within a BSP superstep, which may encompass billions of memory references, Milk captures the temporal and spatial locality of ideal infinite caches on real hardware and provides up to 4x speedup.

CIMPLE is our domain specific language (DSL) to improve the effectiveness of speculative execution in discovering instruction level parallelism and memory level parallelism. Improving memory parallelism on current CPUs allows up to ten memory references in parallel to reduce the effective DRAM latency. Speculative execution is constrained by branch predictor effectiveness and can only uncover independent accesses within the hardware limits of instruction windows (up to 100 instructions). With Cimple, interleaved co-routines expose instruction and memory level parallelism close to ideal hardware with unlimited instruction windows and perfect predictors. On in-memory database index data structures, Cimple achieves up to 6x speedup.

DAWG is our secure cache architecture that prevents leaks via measuring the cache effects of speculative indirect memory references. Unlike performance isolation mechanisms such as Intel's Cache Allocation Technology (CAT), DAWG blocks both speculative and non-speculative side-channels by isolating *cache protection domains*. DAWG incurs no overhead over CAT for isolation in public clouds. DAWG also enables OS isolation with efficient sharing and communication via caches, e.g., in system calls.

Thesis Supervisor: Saman Amarasinghe
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

To learn, read.
To know, write.
To master, teach.
Hindu proverb

This thesis is the culmination point of my work with many talented people. My mentors and mentees, and colleagues and friends have contributed in both tangible and intangible ways. Many of their questions and answers can be found in this thesis.

Thanks to my super SuperUROP Yoana Gyurova, and my undergraduate collaborators Rumen Hristov, Predrag Gruevski, and Damon Doucet. Without them trying the ideas behind Milk and Cimple in real-world applications and confirming that the methods are teachable, these projects would not have started. And the publications partially included in this thesis would not have been completed without my student co-authors who filled-in many of the remaining issues, respectively Yunming Zhang on Milk, Haoran Xu on Cimple, and Ilia Lebedev on DAWG.

William Hasenplaugh, Prof. Charles Leiserson, Prof. Julian Shun, and Prof. Matei Zaharia all brought valuable expert opinions to the development of the ideas and the performance analysis of their results. Stelios Sidiroglou-Douskos has been a great sounding board for many ideas, importantly the security work during its public embargo.

Thanks to Prof. Joel Emer for raising the bar on precision in my writing, presenting, and thinking. Carl Waldspurger has always offered valuable real-world perspectives on early designs and excellent editorial feedback. Our "fun Friday" calls have often been my lifeline.

Thanks also go to my thesis readers for their feedback, example, and guidance: Prof. Srini Devadas for showing me how high quality work can be done way ahead of deadlines. Prof. Martin Rinard for daily encouraging us to be brilliant, and yet showing how to simplify explanations so that anyone can understand.

Thanks foremost go to Professor Saman Amarasinghe, my advisor since 1999, when I walked into his office as an undergraduate transfer student speaking limited English. He has wisely given me freedom to explore when I wanted advice, and when I wanted more freedom — advice.

Thanks also to all members of the Commit and RAW groups over the years: the 1999–2003 students and postdocs who impressed and inspired me to start a Ph.D., and the 2012–2019 cohort whose feedback, fun, and camaraderie helped me finish.

Finally, my greatest gratitude goes to my family for always being there. In memory of my grandfather, who taught me to read when I was 3 to give me a head start, and of my grandmother, who always wanted me to become a doctor. Even though it still took me much too long, and I became the wrong kind of doctor, they would be proud.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

List of Figures

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

Indirect memory references, such as `a[b[i]]` or `p->next`, are essential for many modern applications such as relational databases, key-value stores, graph analytics, and machine learning [23, 61, 83, 193, 198, 200]. Since indirect memory references enable compact representations of sparse relationships between objects, they are ubiquitous in modern applications. For example, they can be found in search engines tracking links between billions of web pages, social network graphs of friendship among billions of users, or book recommendation engines combing which among millions of books have been purchased by which of billions of customers. In-memory databases, graph analytics, and machine learning systems work on data sets reaching terabytes, which now fit in memory on machines available in public clouds (e.g., instances with 12 TB DRAM [8]).

Traditional hardware techniques such as caches and speculative execution, however, are ineffective on large real-world data sets. While the memory system is optimized for accesses to consecutive data locations, indirect memory references usually result in unpredictable sequences of random address accesses. Caches are much smaller than the working set required to cover the random memory references, which therefore require a DRAM access. Speculative execution hardware is in turn unable to effectively hide the DRAM latency, since hard to predict data-dependent branches limit the opportunities to discover independent instructions. Indirect memory references in such large working sets, therefore, incur the full penalty of high latency DRAM accesses, which may leave CPUs idle for hundreds of cycles.

Furthermore, caches and speculative execution enable the recently discovered Spectre [63, 96] family of side-channel attacks, which allow untrusted neighbors in a public cloud to steal victims' secrets. In this thesis, we introduce complementary software and hardware techniques to improve the performance of caches and speculative execution, and to block the largest attack class with minimal overhead.

1.1 The "Memory Wall"

This thesis is primarily focused on finding novel ways around the proverbial "memory wall" and its performance and security challenges. While up to the 1980s CPU and memory speeds were comparable (i.e., equally slow), the dramatic improvement in transistor switching frequency improved CPU clock rates (especially in 1980–2000). DRAM random access latency depends on capacitor recharge cycles and has experienced minimal improvements [76, 77, 78, 79].

On a typical server CPU [45], the latency and throughput of L1 cache and DRAM references result in two to three orders of magnitude differences. For example, while the latency of an L1 access is $\sim$1 ns, a DRAM access is 80–200 ns. Throughput differences between the best case and the worst case can be up to 100,000$\times$ when comparing a sequence of indirect memory reference exhibiting an ideal distribution vs the worst-case resource contention. Theoretical models, as well as traditional programming languages, assume that a memory reference always costs $O(1)$, while these dramatic differences of orders of magnitude are hidden as 'constant factors'.

We must also take into account new technology constraints: Moore's law may soon retreat to a distant memory. While current systems have up to 128 cores and support 512-bit SIMD instructions, Amdahl's law is unrelenting: parallelization, vectorization, and specialized hardware accelerators have diminishing returns bottlenecked by single-thread performance. We also get very little help from Little's law to harness instruction level parallelism (ILP) and memory level parallelism (MLP) to address the performance challenges. What is worse, the best tools employed by hardware architects — speculative out-of-order execution (OoO) and simultaneous multithreading (SMT) — may not be available in public clouds due to novel security challenges due to side-channel attacks.

1.2 Performance Challenges

Currently, the best hardware techniques to scale the "memory wall" rely on large caches and speculative execution. As more and more transistors are available with each technology generation, CPU architects have invested primarily in larger shared caches and more complex out-of-order mechanisms. While such improvements require increased costs in power, area, and complexity, they offer diminishing returns in performance.

While CPU caches are too small to completely fit real-world application data sets, they are often effective thanks to locality of reference. Such effectiveness is accounted to the fractal nature of the world, as many natural and man-made phenomena are described by power-law distributions (such as the Pareto distribution). Therefore, indirect memory references often result in cache hits to frequently accessed, recently accessed, or adjacent to previously accessed cache blocks. The flip side of fractal self-similarity is that cache hits are concentrated in a small portion of caches, thus additional cache capacity provides rapidly diminishing returns. In structured data accesses (such as regular grids, where address patterns are predictable), compiler optimizations can improve cache locality and hardware prefetchers handle cache fill effectively. Indirect memory references, however, are neither predictable statically by compilers, nor typically predictable dynamically by hardware prefetchers. Thus, no standard techniques exist for addressing the performance bottlenecks of sparse and unstructured indirect memory references.

Larger and more advanced microarchitectural structures to capture instruction-level parallelism also offers diminishing returns for typical scalar code [126]. Larger instruction windows for out-of-order execution try to uncover instructions beyond cache misses, while larger and more advanced branch predictors improve the effectiveness of aggressive speculation. Due to Amdahl's Law, in order to improve the maximum efficiency of parallelization and vectorization, even small gains in single threaded performance are deemed valuable. For the important classes of applications we study, however, these bountiful hardware resources remain largely under-utilized.

We introduce novel programming models and software tools (*Milk* and *Cimple*) to improve performance beyond practical hardware capabilities by maximizing the utilization of caches and out-of-order resources. Milk captures the temporal and spatial locality of ideal infinite caches and provides up to 4× speedup on real hardware. Cimple captures the instruction and memory level parallelism of ideal hardware with unlimited instruction windows and perfect predictors, and we demonstrate up to 6× single-thread performance gain on current CPUs.

1.3 Security Challenges

The Spectre [63, 96] speculative side-channel attacks, including the *speculative buffer overflow* class of vulnerabilities we reported [91], take advantage of the interplay between caches and speculative execution. Since shared caches offer the easiest and highest-bandwidth channel for data ex-filtration, we propose a new secure cache architecture *DAWG*. We demonstrate how to protect untrusting virtual machines from each other, and how to protect an operating system from untrusted users. We trade abundant cache associativity and marginally useful cache capacity to re-gain confidentiality.

1.4 Contributions

This thesis makes the following contributions. We introduce programming models and language support to optimize large-scale data analysis by improving ILP, MLP, and TLP, with dramatic performance gains demonstrated in graph analytics, in-memory databases, bioinformatics, and machine learning. Our software contributions are tuned to modern hardware and span new programming models, expressive programming languages, and runtime systems. Our hardware/software co-design of secure caches ensures efficient support for operating systems and hypervisors.

1. In Chapter 3 we introduce *Milk* – our abstract programming model to improve data cache locality. *Milk* preserves parallel program semantics and maps well to the Bulk-Synchronous Parallel (BSP) theoretical model [186]. Bulk-synchronous parallel programs are a superset of the MapReduce [39] paradigm, e.g., graph analytics, database join operations, logistic regression, etc. Within a BSP superstep, which may encompass billions of memory references, Milk captures the temporal and spatial locality of ideal infinite caches on real hardware.

2. **`milk`** is our proposed C++ extension to allow programmers to annotate programs that fit the Milk programming model. Programmers mark individual memory references when completion order can be relaxed. Random indirect memory references are transformed into batches of efficient sequential DRAM accesses. We implement `milk` as an OpenMP extension in Clang.
3. MILK is the complete compiler-generated and runtime system to implement a highly optimized cache-sensitive radix-partitioning on CPUs. Our techniques improve cache locality for large graph and machine learning applications operating on working sets of gigabytes to terabytes range. Evaluation on parallel graph algorithms (PageRank-Δ, Δ-SSSP, etc.) shows up to 3x speedups.
4. In Chapter 4 we analyze the vicious cycle of poor ILP contributing to poor MLP and vice versa. Speculative execution is constrained by branch predictor effectiveness and can only uncover independent accesses within the hardware limits of instruction windows (up to 100 instructions). This dissertation introduces the *IMLP* programming model, where interleaved co-routines expose instruction and memory level parallelism close to ideal hardware with unlimited instruction windows and perfect predictors. IMLP creates a virtuous cycle of high MLP causing high ILP and vice versa.
5. We introduce the domain specific language (DSL) CIMPLE based on coroutines. Our code generation of both *dynamic scheduling*, *static scheduling*, and *hybrid scheduling* maximizes the effectiveness of both out-of-order execution and SIMD vectorization. Cimple allows programmers to expose instruction level parallelism and memory level parallelism not available to current compilers and CPUs.
6. We demonstrate up to 6.4× speedup with CIMPLE on in-memory database index data structures. Improving memory parallelism on evaluated CPUs allows up to ten memory references in parallel and thus reduces the effective DRAM latency.
7. The language systems of CIMPLE and MILK, also depend on carefully optimized runtime libraries to avoid branch misprediction, and cache misses. We also carefully reduce load imbalance in parallelization, avoid serializing locks and expensive cache-to-cache transfers, and enable SIMD vectorization.
8. We introduce the *speculative buffer overflow* class of speculative side-channel vulnerabilities [91], including SPECTRE1.1 (CVE-2018-3693) in Chapter 5.
9. DAWG (*Dynamically Allocated Way Guard*) a secure hardware cache partitioning technique based on way-partitioning of shared and private caches is introduced in Chapter 6. DAWG prevents ex-filtration via timing the cache effects of speculative indirect memory references. We also introduce a replacement policy (e.g., PLRU) attack and defense.
10. DAWG blocks both speculative and non-speculative side-channels via caches by isolating *cache protection domains.* For the important use case of isolating virtual machines in a public cloud, DAWG incurs low overhead over Quality-of-Service (QoS) mechanisms [59].
11. We introduce further facilities in DAWG for confidentiality isolation of an OS from untrusted users. We introduce a layer of indirection to memory references in DAWG to support traditional resource sharing and system calls while enabling efficient sharing and fast communication via caches.

1.5 Roadmap

The rest of this thesis is organized as follows. In Chapter 2, we provide the salient background on modern out-of-order and speculative execution CPUs, caches, and other memory system components.

In Part I, we introduce the two complementary techniques for improving the performance of indirect memory references. First, by improving locality with Milk in Chapter 3, and second, by improving memory parallelism with Cimple in Chapter 4.

In Part II, we introduce the challenges due to speculative side-channel attacks. We review side-channel attacks and introduce a novel vulnerability class due to speculative execution mechanisms in Chapter 5. To secure caches, we introduce DAWG in Chapter 6.

We conclude in Chapter 7, after having demonstrated that the combined software and hardware techniques can achieve both solid security and high performance.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 2

Background

We first review the fundamental laws applied to CPU and DRAM design in Section 2.1. We then examine the organization of DRAM in Section 2.2 and caches and non-blocking cache miss handling in Section 2.3. Finally, in Section 2.4, we review the performance optimizations of speculative-execution out-of-order superscalar CPUs and their implications to performance and security.

2.1 Fundamental Laws

2.1.1 Amdahl's Law: $\boxed{\textbf{Speedup} = \frac{1}{(1-f)+\frac{f}{s}}}$

Amdahl's Law rules the maximum speedup achievable when f is the fraction of the program that can benefit from an s times speedup. For example, Amdahl's Law explains the diminishing returns from superscalar execution, SIMD vectorization, or multi-core parallelization. Our primary focus, therefore, is on improving single-thread performance and memory operations. Nevertheless, we also minimize synchronization in Milk (cf. Section 3.4.3), and we maximize the opportunities for compilers to enable vectorization in Cimple (cf. Section 4.6.1.2).

2.1.2 Little's Law: $\boxed{L = \lambda W}$

Little's Law [117] is a remarkable theorem that states that L – the mean length of a queue of outstanding requests to a system, equals the product of the mean throughput λ and the mean response time W. Little's law is applicable to both CPU and DRAM design, and it informs the design of techniques for instruction level parallelism (ILP) and memory level parallelism (MLP). In this thesis, we will improve throughput λ by optimizations to either L or W (respectively in Cimple and Milk). Next, we review the memory level parallelism accommodated by hardware at all levels of the memory hierarchy.

2.2 DRAM Background

Random memory references in DRAM are often 10–20× slower than sequential memory accesses. The reason for this growing discrepancy is that the memory hierarchy is optimized for sequential accesses, often at the expense of random accesses, at all levels: DRAM chip, DRAM channel, memory controllers, and caches. Hardware approaches for hiding long latency penalties require many independent parallel requests. DRAM parallelism, in turn, is limited by the total number of DRAM banks and the size of queues holding pending requests.

2.2.1 DRAM Organization

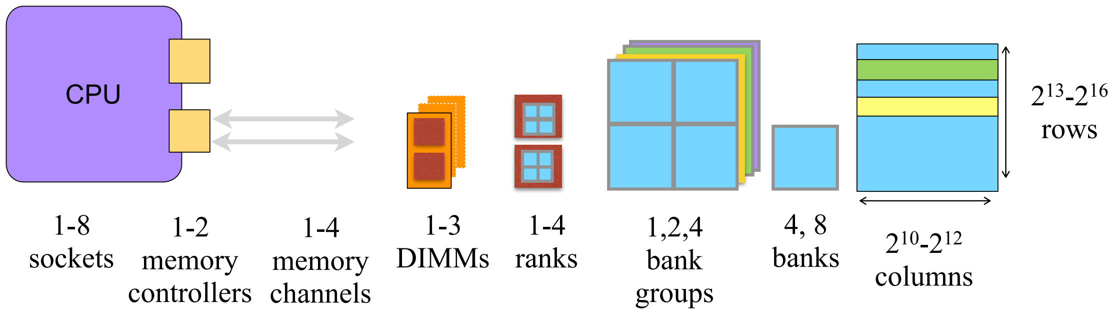


Figure 2-1: Physical structure of DDR3 [78] and DDR4 [79] based systems.

Sequential DRAM reads benefit from spatial locality in DRAM chip *rows*, as illustrated on Figure 2-1. Before a DRAM read or write can occur, a DRAM row – typically 8–16KB of data – must be destructively loaded (*activated*) into an internal buffer for its contents to be accessed. Consecutive read or write requests to the same row are limited only by DRAM channel bandwidth, thus sequential accesses take advantage of spatial locality in row accesses. In contrast, random accesses must find independent request streams to hide the high latency of a row cycle (t_{RC} ∼50 ns [77, 79]) of different banks (*DRAM page misses*), or worse the additional latency of accessing rows of the same bank (*DRAM page conflicts*).

Hardware memory controllers reorder requests to minimize page misses, and other high-latency request sequences to minimize bus turnarounds and read-to-write transitions. Requests, however, can be rescheduled only within a short window of visibility, e.g., 48 cache lines per memory channel [144] shared among cores. Furthermore, requests are equally urgent from the memory controller perspective and cannot be reordered outside of this window.

Random memory accesses use only a small portion of the memory bandwidth consumed. By contrast, sequential memory accesses benefit significantly from large cache lines, as they have good spatial locality and can use all of the data within each line. In many applications with irregular memory accesses, each DRAM access uses only 4–8 bytes out of cache lines transferred – 64 bytes transferred for reads, and 128 bytes for writes (a cache-line read and write-back) – netting 3–6% effective bandwidth.

DRAM interfaces are also optimized for sequential accesses. Modern CPUs transfer blocks of at least 64 bytes from DRAM. Double Data Rate (DDR) DRAM interfaces transfer data on both rising and falling edges of a clock signal: e.g., a DDR3-1600 [78] chip operating at 800 MHz delivers a theoretical bandwidth of 12.8 GB/s. Current generation CPUs use DDR3/DDR4 interfaces with burst transfers of 4 cycles (mandatory for DDR4 [79]), leading to a 64-byte minimum transfer for a 64-bit DRAM. Even though larger transfers would improve DRAM power, reliability, and peak performance, this size coincides with current cache line sizes. High DRAM bandwidth and more memory controllers have kept up with providing high per-core memory bandwidth. The share of total bandwidth for cores on current Intel Xeon servers is 4–6 GB/s.

2.2.2 DRAM Latency Limits

By using Little's Law to solve for throughput of memory requests, we reach that the memory throughput equals the ratio of outstanding memory requests divided by the DRAM latency. The visible effect of either low memory level parallelism or high DRAM latency is underutilized memory bandwidth. Such "latency bound" programs perform well in-cache, and may have no explicit data dependencies, but become serialized on large inputs incurring CPU stall cycles up to the total DRAM latency.

DRAM device latency has been practically stagnant at $\sim$50 ns for the last decade [159, 160]. On top of the typical DDR4 bank latency, additional interconnect, cache coherence, and queuing delays add to a total memory latency of 80–200 ns. Thus, random DRAM requests at the maximum achievable bandwidth have 100$\times$ worse latency than an L1 cache lookup.

Ample parallelism at the DRAM level is usually available, as long as random accesses are distributed well across all banks. The typical maximum of simultaneously open banks on a DDR4 server is 16 banks$\times$2 ranks$\times$(4—6) memory channels. The memory controllers track more pending requests in large queues, e.g., 48+ cache lines per memory channel [144].

For independent random accesses on a well-provisioned system, the parallelism bottlenecks are thus in the shorter queues in caches (cf. Section 2.3.2). Applications with low MLP may have dependent accesses, due to many levels of indirection or pointer chasing. Alternatively, have too many non-memory instructions may fill up the queues of out-of-order cores. To take full advantage of the maximum MLP all intervening instructions must also fit within the out-of-order instruction window (cf. Section 2.4.2).

2.3 Caches

Modern multi-core CPUs employ a multilevel cache hierarchy, typically with 1 or 2 levels of per-core private caches, backed by a shared last level cache. For example, the Intel Haswell [2, 45] systems we use for evaluations in this thesis use a 32 KB L1 data cache, and a 256 KB unified instruction and data cache per-CPU. A shared L3 cache is built with 2–2.5 MB contributions per core for a total shared capacity of respectively 8 MB [2] and 30 MB [45].

We now examine the hardware mechanisms for handling cache misses and memory level parallelism in DRAM and CPUs. Two independent loads may execute every cycle at the L1 latency (typically 4 cycles). Outstanding loads that hit in the L1 cache are limited only by the length of the load buffer (e.g., 72 entries on Haswell) which tracks all non-committed loads (cf. Section 2.4.2).

2.3.1 Non-blocking Caches

CPU out-of-order execution allows multiple long delay memory accesses to be discovered. A non-blocking cache allows more memory requests past older cache miss instructions to be processed but only up to the limits of small hardware structures.

The state of cache lines with outstanding cache misses is handled in a small number of Miss Status Holding Registers (MSHRs) [102]. They allow not only further hits while waiting for a miss, but also even misses after misses. Misses to a cache line with an already scheduled miss simply refer to the previously allocated line. Since a content-associative search is expensive in area and power, the number of MSHRs is hard to scale [183]. The total available MSHRs limit the maximum number of unique cache lines — the maximum memory level parallelism (MLP).

2.3.2 Memory Level Parallelism – MLP and MSHRs

The achievable Memory Level Parallelism (MLP) is limited by the size of the buffers connecting the memory hierarchy layers. The primary MLP limit for single threaded execution is the number of MSHRs.

L1 cache MSHRs are the primary limiter to MLP random access. For example, Intel's Haswell microarchitecture maintains 10 L1 MSHRs (Line Fill Buffers) for handling outstanding L1 misses [65]. These have remained constant on Intel CPUs since Nehalem, while Intel's Pentium4 supported only 8 L1 MSHRs. Similarly, on the high-performance ARM A72 processor, 6 L1 MSHRs support up to six unique cache lines targeted by outstanding cache misses [6].

L2 cache MSHRs allow higher MLP for sequential accesses, including for random accesses to data-structures spanning adjacent cache lines. Since hardware prefetchers track L2 cache accesses, L2 cache misses include requests not only for the needed cache lines (demand requests) but also for hardware prefetchers. With 16 L2 MSHRs, sequential accesses increase memory level parallelism with more outstanding DRAM requests. Current hardware has no prefetchers for random requests. Therefore, sequential reads and writes achieve higher DRAM bandwidth than random accesses.

For current software with low MLP, the MSHRs are hardly a bottleneck. Hardware prefetching and speculative instructions (after branch prediction) are important hardware techniques that put to use the rest of the MSHRs. Hardware prefetching is effective for sequential access – in that case, a few MSHRs are sufficient to hide the access latency to the next level in the memory hierarchy. Speculatively-loaded cache lines on the wrong path are sometimes useful after recovering a branch misprediction. When hardware prefetches are wrong, and wrong-path loads are never needed, these techniques are wasting memory bandwidth and power.

2.4 Speculative Out-of-Order Superscalar CPUs

Helping cores scale the "memory wall" is the most compelling reason for speculative execution, thus modern out-of-order CPUs attempt to uncover independent instructions and most importantly independent memory requests. For comparison, for the time it takes to wait for a DRAM access, a SIMD superscalar core can execute $\sim$5,000 64-bit operations.

2.4.1 Superscalar Execution and SIMD Vectorization

A modern superscalar core can execute up to 8 speculative micro-ops in a given cycle (and up to 4 instructions can commit non-speculative results). For example, in the same cycle Intel's Skylake can execute up to four arithmetic instructions or up to two branches, as well as two loads and one store.

Modern CPUs also support SIMD execution on up to 512-bits vector registers and ALUs. For example, the AVX-512 vector extension uses 16 lanes of 32-bit floats. On Intel's Skylake-E two fused multiply-add (FMA) AVX-512 operations can execute per cycle achieving 64 single-precison floating point operations (FLOPs) per cycle.

2.4.2 Out-of-Order Execution and Reorder Buffers

The Reorder Buffer (ROB) tracks all instructions in flight. Out-of-order processors dynamically schedule instructions whose input dependencies have been satisfied. The instructions, however, are fetched and committed in-order. Therefore, a long latency instruction at the head of the ROB can preclude further instructions from executing. Processors can find unrelated work when stalled on a cache miss, only when all data-dependent instructions can fit in the ROB. Hiding the latency of one DRAM reference (80–200 ns on servers) without stalls, however, often requires more instructions than the entire instruction window.

Instruction Level Parallelism (ILP) is constrained by the reorder buffer as a maximum. Current processors thus support large speculative windows. For example, the re-order buffer (ROB) on Intel's Skylake has space for 224 micro-ops, or about 200 instructions for typical code. Each SMT thread of identical workloads would generally get a half. Branch misprediction, especially of hard to predict branches that depend on indirect memory accesses, however, reduce the effective ROB to a much smaller size.

2.4.3 Speculative Execution

There are three main optimizations that depend on speculative execution: branch speculation, exception speculation, and address speculation.

Branch speculation takes advantage of temporal and spatial locality in the program control flow, and for most programs achieves low branch misprediction rates; high-performance microarchitectures speculate through multiple branches. Exception speculation assumes that most operations, e.g., loads, do not need to trap on unauthorized or invalid MMU translations. Address speculation is used for memory disambiguation, where loads are assumed not to conflict with earlier stores to unknown addresses. Loads are also speculated to hit L1 caches, and immediately-dependent instructions may observe the value 0 (before mini-replay [63, 199]). The first two speculation types are control speculations, and all subsequent instructions are killed; for the third type, only loads and their dependent instructions need to be replayed.

Attempts to expose all three major speculation mechanisms to software — respectively, via predication, speculative loads, and advanced loads [166] — have been largely unsuccessful. Modern instruction set architectures (ISAs), such as RISC-V [192] and ARMv8 [6], are designed to assume high-performance CPUs will use speculation techniques implemented in out-of-order hardware. As a result, they avoid introducing features such as branch hints and predicated execution, and specify a relaxed memory-ordering model.

2.4.4 Branch Misprediction Handling

Highly mispredicted branches are detrimental to speculative execution, especially when a burst of branch mispredictions results in a short effective instruction window. Mispredicted branches that depend on long latency loads also incur a high speculation penalty. Instruction profiling with hardware performance counters can be used to precisely pinpoint such critical branches. The most portable solution for avoiding branch misprediction is to use data or address dependence instead of control dependence. While no further execution of dependent instructions is possible, independent work items can still be serviced.

Most instruction set architectures (ISAs) also support conditional move instructions (`cmov` on x86) (or `csel` on ARM), as simple cases of instruction predication. Automatic predication is also available in simple cases on IBM Power7 [172] where unpredictable branches that jump over a single integer or memory instructions are converted to a predicated conditional selection. The ternary select operator in C (`?:`) is often lowered to conditional move instructions, however, use of assembly routines is unfortunately required to ensure that mispredictable branch instructions are not emitted instead of the desired conditional move instructions.

Speculative attack execution is also limited to the number of instructions that execute until a mispredicted branch has been resolved. When a branch is resolved wrong-path instructions are flushed. However, their side effects such as loaded cache lines, or outstanding cache requests in MSHRs are not cancelled, with the anticipation that they may be useful.

2.4.5 Software Prefetching

Software prefetch instructions achieve higher MLP than regular loads. The effective Memory Level Parallelism (MLP) is constrained not only by the ROB size, but also by the capacity limits of resources released in FIFO order, e.g., or load and store buffers. A loop body with a large number of instructions per memory load may reduce the effective ILP and MLP, i.e., the reorder buffer (ROB) must fit all micro-ops since the first outstanding non-retired instruction.

The instruction reorder buffer, or any resource held up by non-retired instructions may become the limiting factor: 192-entry reorder buffer, 168 registers, 72-entry load and 42-entry store buffers on Haswell [65]. These resources are plentiful when running inefficiently one memory request at a time. Dividing the core resources over 10 parallel memory requests, however, means that each regular load can be accompanied by at most 19 µops using at most 16 registers, 7 loads and 4 memory stores.

Prefetch instructions free up the instruction window as they retire once the physical address mapping is known, e.g., either after a TLB hit, or after a page walk on a TLB miss. As soon as the virtual to physical translation has completed, an L2 memory reference using the physical address can be initiated. On current Intel microarchitectures the PREFETCH*h* family of instructions always prefetch into the L1 cache. Software prefetches are primarily limited by the number of L1 MSHR entries. Maintaining a longer queue of in-flight requests (limited by load buffers), however, helps to ensure that TLB translations of the following prefetches are ready as soon as an MSHR entry is available. If hardware performance counters show that dependent loads miss both the L1 cache and MSHRs then prefetches are too early; if loads hit MSHRs instead of L1 then prefetches are too late.

THIS PAGE INTENTIONALLY LEFT BLANK

Part I

Performance Optimizations

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 3

Optimizing Indirect Memory Reference Locality with `milk`

3.1 Overview

Modern applications such as graph and data analytics operating on real-world data have working sets much larger than cache capacity and are bottlenecked by DRAM. To make matters worse, DRAM bandwidth is increasing much slower than per CPU core count, while DRAM latency has been virtually stagnant [76, 77, 78, 79]. Parallel applications that are bound by memory bandwidth fail to scale, while applications bound by memory latency draw a small fraction of much-needed bandwidth. While expert programmers may be able to tune important applications by hand through heroic effort, traditional compiler cache optimizations have not been sufficiently aggressive to overcome the growing DRAM gap.

In this chapter, we introduce **`milk`** — a C/C++ language extension that allows programmers to concisely annotate loops with performance-critical indirect memory references. Efficient multi-pass radix partitioning using optimized compiler-emitted intermediate data structures transforms random indirect memory references into streamlined batches of sequential DRAM accesses. Milk's programming model maps well to the Bulk-Synchronous Parallel (BSP) theoretical model [186] and preserves parallel program semantics. This simple semantic model enhances programmer productivity for efficient parallelization with OpenMP with simple **`milk`** annotations.

We evaluate the Milk compiler on parallel implementations of traditional graph applications, demonstrating performance gains of up to 3×. Hardware mechanisms, including memory controllers and out-of-order schedulers, can reorder only a limited window of tens of operations. Milk, on the other hand, efficiently orchestrates billions of accesses and finds long-term locality that hardware is unable to identify.

The rest of this chapter is organized as follows. We motivate the need for Milk in Section 3.2 by evaluating the available ideal locality in graph applications. We then introduce the Milk programming model and Milk compiler and runtime library design in Section 3.3, **`milk`** syntax and semantic extensions in Section 3.4, and implementation details of Milk optimizations in Section 3.5. We demonstrate Milk expressiveness on parallel graph applications in Section 3.6, and we evaluate performance in Section 3.7. Section 3.8 surveys related work. We highlight directions for future work in Section 3.9 and summarize in Section 3.10.

3.2 Motivation

Memory bottlenecks limit the performance of many modern applications such as in-memory databases, key-value stores, graph analytics, and machine learning [23, 61, 83, 193, 198, 200]. These applications process large volumes of in-memory data since DRAM capacity is keeping pace with Moore's Law. Indirect memory references in large working sets, however, cannot be captured by hardware caches. Out-of-order execution CPUs also cannot hide the high latency of random DRAM accesses, and most cycles are wasted on stalls. Compiler cache optimizations also cannot statically capture locality in irregular memory accesses even when loop-level locality exists.

Parallel loops already allow compilers to use not necessarily the serial order, but any execution order. Indirect memory reference performance dramatically depends on the access pattern. The following read-modify-write loop is typical in many domains, e.g., graph analysis (counting graph vertex degrees), image processing (histograms), and query planning (counting cardinality):

```
#pragma omp parallel for
for(int i=0; i<N; i++)
  #pragma omp atomic
  count[d[i]]++;
```

If the values of d[i] are monotonically increasing consecutive numbers over a large range such that count is DRAM-resident, this parallel loop achieves 18% of peak DRAM bandwidth of a multi-core CPU [2]. Note that if the inputs are known to be sorted we could have optimized this loop to remove atomic updates, and thus it can reach 54% of peak system-wide bandwidth. On uniform random values of d[i], however, this loop's effective bandwidth is only 2%! Every 4-byte update consumes 128 bytes of DRAM read-write traffic.

For this loop on random inputs, the Milk compiler achieves 4× end-to-end speedup (see Figure 3-15 in Section 3.7). Our compiler transformations dynamically partition the index values in d to capture all available locality in accesses to count. Each partition accesses a subset of memory that fits in cache. In addition, synchronization is eliminated since threads process disjoint partitions. Just adding **milk** clauses to the OpenMP directives enables these optimizations.

To achieve high performance and low overhead, Milk's *DRAM-conscious Clustering* transforms loops into three logical phases: Collection, Distribution, and Delivery. In Collection, dynamically generated indices of indirect accesses (e.g., simply d[i] above) are first collected in cache local buffers instead of making random DRAM accesses. In Distribution, indices are written to DRAM resident partitions using efficient sequential DRAM access. In Delivery, each partition's deferred updates are read from DRAM and processed, along with dependent statements.

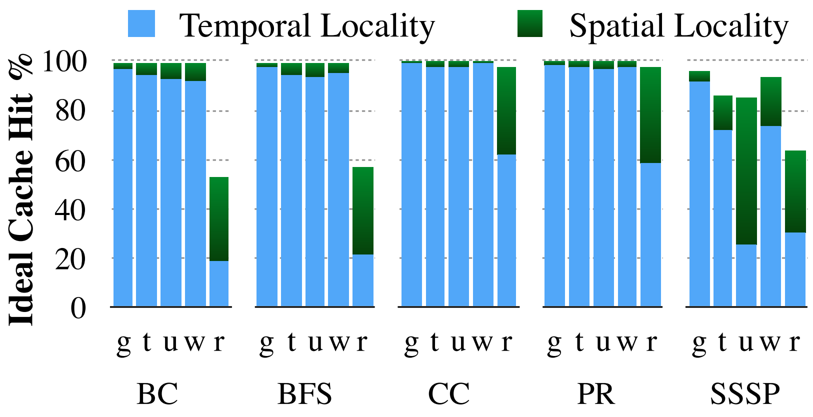


Figure 3-1: Temporal and spatial locality on an infinite cache with 64-byte lines of indirect memory references per superstep of **B**etweenness **C**entrality, **B**readth-**F**irst **S**earch, **C**onnected **C**omponents, **P**age**R**ank, and **S**ingle-**S**ource **S**hortest **P**aths. Synthetic graphs with V=128M vertices and avg. degree d=16: power law Graph500 (**g**) and uniform (**u**); and real-world graphs: US roads V=24M, d=2.4 (**r**); Twitter V=62M, d=24 (**t**); weblinks V=51M, d=39 (**w**).

These three logical phases are similar to inspector-executor style optimizations [42, 125, 158]; in Milk, however, to eliminate materialization of partitions and conserve DRAM bandwidth, the optimization phases are fused and run as coroutines. Prior research either focused on expensive preprocessing that resulted in net performance gain only when amortized over many loop executions, or explored simple inspection for correspondingly modest gains [42]. In contrast, our transformations are well optimized to pay off within a single execution — as required in real applications that use iterative algorithms with dynamically changing working sets or values.

The Milk execution model is a relaxed extension of the bulk-synchronous parallel (BSP) model [186], in which communicating processors work on local memory during a *superstep* before exchanging data at a global barrier. Milk virtual processors access randomly only one DRAM-resident cache line per superstep, in addition to sequential memory or cache-resident data. All indirect references are deferred to the next superstep. A Milk superstep may encompass loop nests that include billions of deferred references. References targeting the same cache line are grouped to maximize locality, and are processed by a single virtual processor to eliminate synchronization. This abstract programming model is similar to MapReduce [39] with discrete phases of *Map* and *Reduce* for processing references to distinct memory locations. Milk programs, however, allow programmers to continue thinking in terms of individual memory references and to maintain traditional serial or OpenMP syntax, usually without semantic change.

We investigate here the total locality available in applications, compare to locality captured by hardware, and then show Milk's effectiveness. To find an upper bound on locality within a Milk superstep, we use an ideal cache model. An ideal cache would not incur cache capacity or cache conflict misses, or true or false sharing coherence

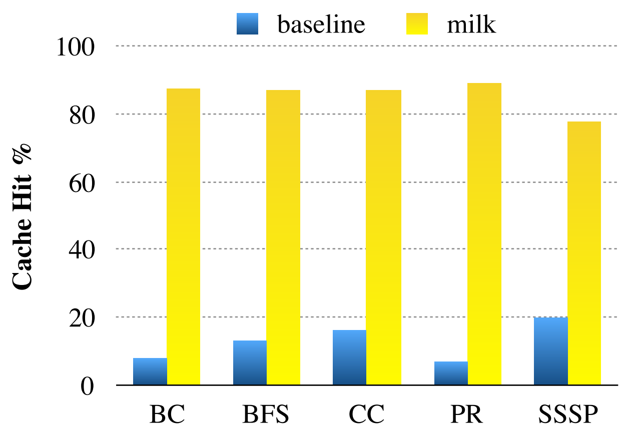


Figure 3-2: Cache hit rates of demand reads of baseline and with Milk on 256 KB L2, 8 MB shared L3 [2], for V=32M, d=16(uniform).

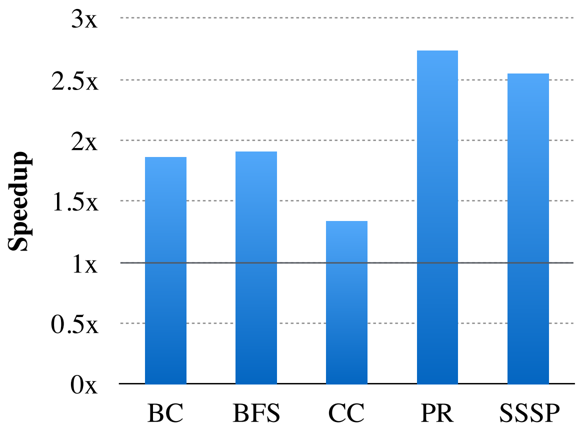


Figure 3-3: Milk overall speedup. 8MB L3, V=32M, d=16(u).

misses on a multiprocessor system. The only memory demands should be compulsory cache misses and output write-backs. (We assume all data is written to memory between supersteps, therefore reuses across supersteps are counted as compulsory misses.)

Since compulsory cache misses, temporal locality, and spatial locality are program- and data-dependent, we instrumented the indirect memory references of commonly used [18, 135, 163, 170] graph applications with state-of-the-art implementations in the Graph Algorithm Platform Benchmark Suite (GAPBS) [17]. The GAPBS datasets include public samples of real graphs (*Twitter*, *USA road*, and *web crawl*), and synthetic graphs (*uniform* and *power law* [30, 52]) to compare sensitivity to graph degree distributions and diameters. We describe the applications in more detail in Section 3.6, where we also show their handwritten in OpenMP critical loops.

Figure 3-1 shows that, on an ideal cache, the total locality within a superstep is close to 100% for most graphs and applications, and above 50% for all. Temporal locality on an infinite cache (i.e., reuse of the same address) captures more than 90% of accesses for high-degree low-diameter graphs. However, for the road graph and for SSSP, harnessing spatial locality (i.e., capturing reuse of 64-byte cache lines) is important. Unlike hit rates on the ideal infinite cache, the cache hit rates observed on real hardware are much lower, and most cycles are wasted on memory stalls.

Figure 3-2 shows the low cache hit rates of real hardware are improved by 4× with Milk. On a synthetic graph with 32 M vertices, most applications have greater than 128 MB total working sets, and little locality is captured by the 8 MB shared L3 cache and 256 KB L2 per core on a Haswell CPU [2]. To isolate indirect memory reads during full program execution, we use hardware performance counters that track non-prefetch demand reads that hit L2 or L3 caches. In the baseline 80–90% of all L1 misses require a DRAM access, while with Milk they are served by L2 and L3 caches. Figure 3-3 shows that these improved hit rates translate to end-to-end speedups, ranging from 1.3× to 2.7×.

3.3 Design

Given the inefficiency of random DRAM references, limited cache capacity, and limited latency-hiding mechanisms of out-of-order hardware, the Milk compiler harvests locality beyond hardware capabilities with a software-based approach. The Milk compiler plans all memory accesses within each *superstep* of programs that fit the Milk execution model and are annotated with **`milk`**[1].

Milk achieves significant performance improvement by reordering the memory accesses for improved cache locality. The reordered memory references maximize temporal locality by partitioning all indirect references to the same memory location within the cache capacity. The planned accesses also improve spatial locality by grouping together indirect references to neighboring memory locations. Furthermore, Milk avoids true sharing, false sharing, and expensive synchronization overheads by ensuring only one core writes to each cache line.

However, naively reorganizing indirect references, e.g., by sorting, adds non-trivial overhead. Milk keeps all additional bandwidth low by using only efficient sequential DRAM references. Although data transformations require an investment of additional CPU cycles and sequential DRAM bandwidth, we show how to minimize these overheads with *DRAM-conscious Clustering*.

In order for the Milk compiler to perform the optimization automatically, users must annotate indirect accesses in parallel OpenMP loops with a **`milk`** *clause*. Such simple use is illustrated for simple loops like line 2 in Figure 3-4a. Section 3.4 describes all clauses and directives in **`milk`**'s syntax. Here we illustrate the most common ones with an example on Figure 3-4b. An explicit **`milk`** *directive* selects indirect references that should be deferred, along with their context (see line 12). Optional *combiner* functions allow programmers to summarize the combined effects of updates targeting

[1]Milk's **`milk`** is an homage to Cilk's **`cilk`** [49].

```
void f(int a[], int ind[], int x) {
  #pragma omp parallel for milk
  for(int i=0; i<n; i++)
    a[ind[i]] = x;
}
```

(a) Implicit Indirect References with **milk**.

```
void g(float a[], int ind[],
       float b[]) {
#pragma omp parallel for milk
  for(int i=0; i<n; i++) {
     float value = b[i];
     int index = ind[i];
#pragma milk pack(value:+) tag(index)
#pragma omp atomic if(!milk)
     a[index] += value;
  }
}
```

(b) Explicit Indirect References with **milk**.

```
void f(int a[], int ind[], int x) {
  _Milk_containers c;
  #pragma omp parallel
  {
    #pragma omp for nowait
    for(int i=0; i<n; i++)
      _Milk_collect(ind[i], &c);
    _Milk_distribute(&c);
    _Milk_deliver(&c, Context(a, x),
                  .milk.outlined.f);
  }
}
void g(float a[], int ind[],
       float b[]) {
  _Milk_containers c;
  #pragma omp parallel
  {
    #pragma omp for nowait
    for(int i=0; i<n; i++)
      _Milk_collect(&c, ind[i], b[i]);
    _Milk_distribute(&c, SumCombine);
    _Milk_deliver(&c, Context(a),
                  .milk.outlined.g,
                  SumCombine);
  }
}
```

(c) Logical transformation overview.

Figure 3-4: DRAM-conscious Clustering transformation.

```
.milk.outlined.f(int _tag,
                 Context *c) {
  c->a[_tag] = c->x;
}
.milk.outlined.g(int index,
                 float value,
                 Context *c) {
    c->a[index] += value;
}
```

Figure 3-5: Outlined functions for Clustered Delivery.

the same address. Finally, atomics are needed only for the baseline OpenMP version (on line 13).

The Milk execution model is similar to MapReduce [39]. We do not offer a *Map* interface, however, since efficient iteration over in-memory data structures can use domain-specific knowledge (e.g., incoming vs. outgoing neighbor traversal in graph processing, 2-D loop nests for image processing, 3-D loop nests in physical simulations, etc.). Leaving the outer loop structure to programmers allows further algorithm-specific optimizations, such as use of bit vectors, Bloom filters, priority queues, etc. Further knowledge about power-law data distributions can be used to maximize load balance in parallel loop nests, or to identify accesses that are likely to be cached. Domain-specific frameworks that apply user-supplied *Map* functions over the input can be built on top of our **`milk`** primitives.

3.3.1 Collection, Distribution, and Delivery

In the *Collection* and *Distribution* phases, Milk plans memory accesses by grouping references into cache-sized partitions. In the *Delivery* phase, the reordered memory references and deferred dependencies are applied. These phases are a generalization of cache partitioning [167] used in heavily optimized database hash-join [4, 12, 88, 122, 167, 179, 191], with additional DRAM optimizations shown in Section 3.5.2.

Figure 3-4c shows a logical overview of the DRAM-conscious Clustering transformation to Collection, Distribution, and Delivery phases. The transformation for line 4 in `f()` shows how a tag index is collected, while the more complex case of line 14 in `g()` also includes a payload value, and a combiner function.

During Collection, indirect reference expressions are evaluated as *tags*, e.g., `ind[i]` on lines 4 and 11. If additional per-reference state is needed, it is also collected with *pack* as a payload; an optional *combiner* enables coalescing of payloads targeting the same tag (e.g., line 12). Tag and pack variables can be initialized with any dynamically evaluated expression and can have side effects (e.g., queue iterators can consume items), since they are written to temporary buffers.

During Distribution, tags are partitioned into clusters such that maximum cache line and DRAM page locality can be achieved for accesses during Delivery. Depending on the range of tag values, if too many partitions would be needed, multiple Distribution

Table 3.1: Reduction of DRAM bandwidth in bytes, for m random references to n locations with MILK.

	Original	**MILK**		
	Program	Collection	Distribution	Delivery
DRAM	$128m$		$8m$	$8m+8n$
L2		$8m$		$128m$

passes are required. Each additional Distribution pass uses only sequential transfers to DRAM.

During Delivery, an outlined program slice from the original code is executed using a dynamically constructed context valid for each Delivery thread. Milk captured statements allow arbitrary C++ expressions and function calls, e.g., arbitrary containers can be accessed during Delivery. Figure 3-5 shows the outlined functions created to capture the deferred execution of lines 4 and 14, respectively, on lines 45 and 50.

Distribution phases are always fused with the previous or next processing stage: the first Distribution pass is fused with Collection, and the final Distribution pass is fused with Delivery. The three distinct phases shown as functions are implemented as coroutines, i.e., Collection receives one element at a time to pass to Distribution, and early Delivery can be triggered at any time if memory for temporary buffers is exhausted. Unlike inspector/executor-style [158] preprocessing, the three phases are distinct only logically, and different threads and different tags may be in different phases at the same time.

3.3.2 Performance Model

A simple model predicts Milk DRAM bandwidth savings as a high order estimate of performance on bandwidth-bound applications. Let us take a workload that makes m random read-modify-writes to n distinct location, e.g., processing a graph with n vertices and m edges. Assume n is much larger than the cache size, and that updated values, indices, and payloads are 4 bytes. If $n \ll m$, the Milk version uses 8× less bandwidth than the original.

Table 3.1 summarizes the bandwidth required by the transformed workload and overheads of Milk phases. Original references read and write back a random cache-line on each update. The transformed updates are within L2 caches and use DRAM bandwidth only for compulsory cache-fill and write-back of accessed cache lines. When spatial locality is high, as seen earlier in Figure 3-1, these use $2n \cdot 4B$. Collection arranges tag and pack records in L1/L2 caches, and Distribution and Delivery write and read DRAM sequentially.

3.3.3 Milk Correctness Conditions

All BSP-model [186] applications are correct under Milk, as are some non-BSP programs allowed under the more relaxed Milk execution model. Correctness proofs of BSP algorithms hold for Milk execution. The only difference is the number of virtual

Table 3.2: **`milk`** OpenMP extensions.

Directive	**Clause**	**Section**
`milk`	`if`	3.4.1.1
`milk`	`pack(`*v*`[:all])`	3.4.1.3
`milk`	`pack(`*v*`:+│*│min│max│any)`	3.4.1.4
`milk`	`pack(`*v*`:first│last)`	3.4.1.4, 3.4.4
`milk`	`tag(`*i*`)`	3.4.1.2
`atomic`	`if`	3.4.3
`for`	**`milk`**	3.4.1
`ordered`	**`milk`**	3.4.4

processors, i.e., tens of threads vs. millions of cache lines. Loops in which read and write sets do not overlap will not change behavior when split in Milk supersteps. For non-BSP programs, a loop *may* read results produced within the same step. Programs that produce externally deterministic [21] output but vary intermediate results (such as Connected Components in Figure 3-11) may incur more per-iteration work or a lower convergence rate with Milk but will be correct.

Programs with read/write dependencies, which produce non-deterministic outputs under OpenMP, have to be validated or modified for Milk. Domain knowledge may allow divergence from sequential semantics, e.g., in traditional bank check processing all checking account debits are processed or bounced first, and only then any account credits are applied. Splitting a loop into separate parallel loops to break dependence chains allows the alternative sequential execution, OpenMP, and Milk to all agree on the output.

3.4 Milk Syntax and Semantics

The Milk compiler allows programmers to use the Milk execution model for C/C++ programs with OpenMP extensions and loop annotations. Adding a **`milk`** clause to a loop directive is often sufficient to enable optimizations. To achieve further optimization control, programmers can select which references in a loop should be deferred by adding a **`milk`** directive and its clauses. A `tag` clause selects the destination, while `pack` clauses specify the values necessary for deferred dependent statements. The `if` clause allows programmers to express dynamic knowledge about data distribution by deferring only references that are unlikely to be cached. Additional language and library features support safe avoidance of synchronization, deterministic performance, and deterministic outputs with minimal overhead. Table 3.2 summarizes the clauses of the `pragma` **`milk`** directive and extended OpenMP directives.

3.4.1 `milk` Directive and Clause

`milk` directives within a **`milk`** loop afford fine-level control when Collecting indirect accesses. This use is similar to the OpenMP `ordered` loop clause and the identically named directive used in the loop body; we borrow the same nesting and binding

```
 1 void BFS_Depth(Graph& g, int current,
 2                      int depths[]) {
 3 #pragma omp parallel for milk
 4   for (Node u=0; u<g.num_nodes(); u++)
 5     if (depths[u] == current)
 6       for (Node v : g.out_neigh(u))
 7 #pragma milk
 8           if(depths[v] == -1)
 9               depths[v] = current + 1;
10 }
```

Figure 3-6: Breadth-First Search (BFS) with **`milk`**.

rules [1]. Figure 3-6 illustrates this annotation syntax for a Breadth-First-Search traversal (BFS), which produces a vector of depths from the source node. Sequential access on line 5 can be processed directly, but random accesses on lines 8 and 9 are deferred with **`milk`**.

3.4.1.1 `if` Clauses and Filter Expressions

Milk optimizations are effective only when locality exists within a superstep, but is beyond the reach of hardware caches. When programmers can cheaply determine effectiveness, they can provide such a hint as a superstep property, or as a dynamic property of each indirect reference. A loop can be marked for execution without deferral if programmers can determine that the optimization will be ineffective because the range of accesses is small, there are too few requests, or if little locality is expected. For example, a `parallel for` can be marked with OpenMP 4.5 `if:` syntax, e.g.: `#pragma omp` **`milk`** `for if(milk: numV<cacheSize)`.

Individual **`milk`** directives can be tagged with an `if` clause containing a *filter expression* evaluated for each memory reference. This is useful whenever domain-specific knowledge can cheaply determine that a specific address is likely in cache. For example, in graph algorithm implementations where graph nodes are ordered by degree, the high-degree nodes that are in cache can be skipped, e.g.: `#pragma` **`milk`** `if(v>hot)`. Similarly, highly skewed hot features may be filtered thru an L1-cache sized hash table, while only misses need to be collected.

3.4.1.2 `tag` Clause

The `tag` clause can make explicit the definition of the index variable used in the following statement or block, as seen in Figure 3-4b. The clause takes as an argument the identifier of a variable that programmers should define to use the smallest possible type for supported inputs. An explicit tag is also required when using collections that do not use the C++ subscript `operator[]`, e.g., `vector.at(index)`.

3.4.1.3 `pack` Clause

The `pack` clause explicitly specifies data needed for correct continuation execution. It takes a list of variables that must be carried as a different payload for every tag. Programmers should minimize the number of variables whenever it is possible to evaluate expressions early (see for examples line 47 on Figure 3-10 and line 98 on Figure 3-13). Here we collect only one 32-bit value, rather than two:

```
float contribV = (1 + deltas[v]) /
                      paths[v];
...
#pragma milk pack(contribV : +)
```

Reducing the size of `pack` variables also directly reduces overhead. The performance of traditional indirect reference-bound kernels is a function of the number of cache lines touched regardless of the amount of data. The costs are identical whether 1 bit or 2 doubles are written: a random DRAM read, and a random DRAM writeback upon cache eviction. Reducing precision is ineffective traditionally if the reduced working set is still larger than cache. MILK transformations, however, are bandwidth bound. Thus, the smallest type for the expected range should be used for `pack` variables. Floating point variables can be cast to lower precision types, e.g., users can explore the non-standard `bfloat16` communication type from TensorFlow [3], which is simply a 16-bit bitcast of `float`.

3.4.1.4 `pack` Combiners

For each packed variable, a combiner function can be specified, e.g., `pack(u:min)`, `pack(value:+)`, etc., which may be used to combine `pack` payloads before Delivery. When a combiner is used, programmers must ensure the program is correct, whether all, one, or a subset of all items is delivered. Valid combiners can be defined for all applications we study in Section 3.6.

Combiners are optional, as their performance advantage depends on the data distribution and access density for each partition. For uniform distributions, for example, early evaluation of combiners will not discover temporal locality to match tags. Combiners evaluated immediately before executing the continuation allow faster processing of all updates targeting a given tag. While not saving bandwidth, late combiner execution enables deterministic outputs, e.g., with `min`/`max` combiners.

The default binary operations supported include the standard OpenMP reductions, e.g., `+`, `*`, `min`. Now that OpenMP 4.0 user-defined reductions [1] are supported by `clang`, we plan to allow similar syntax for user-defined combiner expressions. While OpenMP `reduction` clauses for scalars (or short dense arrays in OpenMP 4.5) use unit value initializers (e.g., 0, 1, `INT_MAX`), the explicit initialization incurs significant overhead on large and sparsely accessed data. Therefore, Milk combiners are assigned or copy/move constructed only on first access.

Additional pre-defined pack combiner identifiers include `all`, `any`, `first`, and `last`. Specifying `all` explicitly documents that each value must be delivered without a combiner. When any value can be used in any order, `any` marks that duplicate pack values can be dropped. Alternatively, `first` and `last` select one pack value in the respective order it would be collected in serial execution.

3.4.2 Elision Version Semantics

Same-source programs can be compiled in several versions – traditional serial, OpenMP parallel, and Milk (serial or parallel) – dispatched depending on input size. Eliding Milk annotations produces valid serial programs. Milk programs without atomics are succinct and easier to reason about than OpenMP parallel programs. Parallel non-Milk programs, however, may be desired for efficient execution on small cache-resident inputs. When correctness of parallel non-Milk versions requires additional synchronization, integration with OpenMP allows correct and efficient execution.

3.4.3 Atomic Accesses

When unnecessary for **`milk`** loops, synchronization can be eliminated by marking `omp atomic if(!milk)`. This OpenMP syntax extension allows an `if` clause in `atomic`.

All threads updating data concurrently must be in the same **`milk`** loop. As a safe default, we preserve the behaviour of unmodified `omp atomic` directives to use atomics, in case they were intended for synchronization with other thread teams, or with external threads via shared memory. During Delivery, atomic operations operate on cache resident data and are executed by a single thread. Such atomics are faster than DRAM resident or contended cache data with expensive cache-to-cache transfers. However, atomics still drain store buffers and destroy instruction and memory parallelism, incurring 3× overhead. When possible, unnecessary atomics should be eliminated.

Milk safely allows either only reads or only writes to the same array. Mixing reads and writes, or `atomic capture` clauses may change non-BSP program semantics. Currently, loops with internal producer-consumer dependencies should be transformed manually to unidirectional accesses per array per loop to ensure program semantics does not change. Deferring an `atomic update` can be treated as a write and requires no semantic change.

Read-modify-write `update` operations that do not peek into results can issue in parallel. For example, the CountDegrees kernel – used for building a graph in GAPBS – shown in Figure 3-7 can access two edges with an `atomic update`.

3.4.4 Deterministic and Ordered Extensions

Adding an `ordered` clause ensures deterministic outputs by restricting **`milk`** processing order. This improves programmer productivity and enables new application use cases, e.g., database replication and fault tolerance. Milk processing can give deterministic execution guarantees very efficiently: there are very few thread synchronization points during Distribution, while the cost of enforcing deterministic order is amortized across thousands of individual memory references. The main concern

```
CountDegrees(vector<Edge>& edges) {
    vector<Node> degrees(n, 0);
#pragma omp parallel for milk
    for (Edge e : edges) {
#pragma omp atomic if(!milk)
        degrees[e.u] += 1;
        if (!directed)
#pragma omp atomic if(!milk)
            degrees[e.v] += 1;
    }
}
```

Figure 3-7: `CountDegrees` with multiple atomic updates.

Table 3.3: Milk API summary.

Intrinsic	**Section and Use**
`milk_collect_thread_num`	3.5.1.4 Collecting thread ID
`milk_deliver_thread_num`	3.5.1.4 Delivering thread ID
`milk_set_max_memory`	3.5.2.2 Maximum memory
`milk_set_strict_bsp`	3.5.2.2 Superstep splitting
`milk_touch`	3.5.2.2 Working set estimate

in deterministic execution is that it exacerbates load imbalance during a Collection work-sharing loop, therefore programmers have to explicitly mark these after handling load imbalance.

3.5 Milk Compiler and Runtime Library

The Milk compiler and runtime library are implemented within the LLVM [108] framework, as Abstract Syntax Tree (AST) transformations in Clang and code generation of LLVM IR. Collection and Delivery phases are implemented as AST lowering phases emitting specializations of the Distribution container library. Table 3.3 summarizes the additional programmer-facing intrinsics.

3.5.1 Dynamically Constructed Continuations

Milk Delivery's indirect memory references are logically separate continuations, but we dynamically construct the full context using four classes of data: a static context shared among all continuations for each deferred block, a tag to construct address references to one or more arrays, an optional payload carried with each tag, and optional thread-local variables.

While similar to C++ lambda expressions, continuations need to allow access to out-of-scope variables across different threads. Compared to `delegate` closures in distributed systems [129], we minimize per-item overhead by optimizing for space and capturing only unique values across iterations. Explicit uses of `tag` and `pack` define the only variables that are unique per tag and packaged as payloads. All variables other than globals used in a continuation must be captured in a shared context.

3.5.1.1 Shared Context

To minimize space overhead, we use a single closure to capture the original block and all variable references necessary for execution during Delivery of continuations.

Variables that are independent of the iteration variable are simply captured by reference only once for all iterations, e.g., the array base and value in `{m[ind[i]] += val;}` can capture `m` and `val` as long as they are not aliased. If `m` may change, however, programmers should either add `pack(m)`, or recompute `m` in the continuation.

Loop-private variables cannot be passed by reference for Delivery; if programmers can hoist them out of the loop, they are automatically stored in the shared context. Otherwise, variables need to be either explicitly passed by value with `pack`, or recomputed.

3.5.1.2 Tags

Each block must have a single implicit or explicit `tag`, but multiple references based on the same tag can access different arrays, vectors, or C++ objects with overloaded `operator[]`, e.g., `if (a[v]) b[v]++`. Therefore, tags stored in Collection are indices, not pointers.

3.5.1.3 Pack payloads

Initialization of variables in `pack` clauses is evaluated during Collection. Variables are not added automatically to `pack` clauses; any non-packed variable reference that is not in the shared context results in a compile-time error. Currently, programmers have to decide whether an expression should be evaluated during Collection, Delivery, or in both.

Evaluation during Collection is best for expressions that make sequential accesses, or can be `pack`ed as a compact payload. For example, here the convenience `Pack` macro expands to `_Pragma("milk pack")`:

```
bool b = (a[i]>1); Pack(b) { ... }.
```

Evaluation during Delivery is favorable when indices are smaller than values and refer to cache-resident data, or expressions have larger size, e.g.:

```
Pack(i) {double ai=a[i], r = 1.0/i; ... }
```

Evaluation on both sides trades memory bandwidth for arithmetic unit cycles; programmers must explicitly recompute any shadowed variables, e.g.:

```
float f = b[i];
double d = f*f;
...
Pack(f) {double d = f*f; ... }
```

3.5.1.4 Thread-Local Variables

Thread-local variable expressions may be expected to be exclusive to their owner, but are now accessed in both Collection and possibly different Delivery threads. Thread-local variables here refers both to OpenMP `threadprivate` global variables and to indirection via CPUID, thread ID (TID), or `omp_get_thread_num`.

TID references in left-hand expressions in continuations are usually intended to use variables of Delivery threads. A common use for thread-local variables is to implement reductions manually, e.g., lines 32 and 103 in Section 3.6. The alias `milk_deliver_thread_num` explicitly documents the intent that the OpenMP function `omp_get_thread_num` should use Delivery thread local variables. When user-defined reductions [1] are explicitly declared, thread-local references use Delivery threads.

TID references in right-hand expressions may have to be evaluated in the Collect threads. For `const` references, the intrinsic `milk_collect_thread_num` should replace TID references. This helps reduce payload size as many requests share the same source TID value. Thread-local variables cannot be passed by non-`const` reference to continuations, as multiple Delivery threads may update the same item. To guarantee correct execution without atomics, expressions can be evaluated and packed during Collection, e.g., to generate unique ID's in a pre-allocated per thread range: `{m[ind[i]] = state[TID].id++;}`.

3.5.2 Milk Containers and Distribution via Radix Partitioning

Milk Distribution fuses a nested multi-pass radix partitioning optimized for all levels of the memory hierarchy from registers to DRAM. Distribution is a generalization of cache partitioned join [167], and TLB-aware radix join [122], adjusted for the capacity and access costs of modern hardware memory hierarchy with additional optimizations for maximizing DRAM bandwidth. Naively appending to a per-partition buffer in DRAM would require 2·64B random updates to and negate any cache partitioning benefits. The internal Distribution coalesces updates in full cache lines using a container library specialized for transportation of different sized tags and payloads. As illustrated in Figure 3-8, instead of naïve 'buckets', for each partition there is a *jug* and/or a *pail*, and a list of *tubs*. These append-only two-dimensional structures are logically simple, but they require careful choice of widths (i.e., number of elements), heights (i.e., fan-out), address placement and access patterns.

A *jug* is an L1-cache-resident container, optimized for fast writes with low register pressure. Branch misprediction impact on jug overflow can be reduced with wider jugs, but that increases total L1 cache occupancy, thus typical widths are 32–512 bytes. A full jug is emptied into one or two *pails*.

A *pail* is an L2-cache-resident container primarily designed to build full cache lines with streaming stores to the next stage. Pails are optimized for cache capacity, and further optimized to avoid cache associativity conflicts. Additionally, when appending to a pail unused bits can be trimmed, e.g., when partitioning into 256 pails with a bit mask, only the remaining 24-bits need to be stored. When compiled for Haswell,

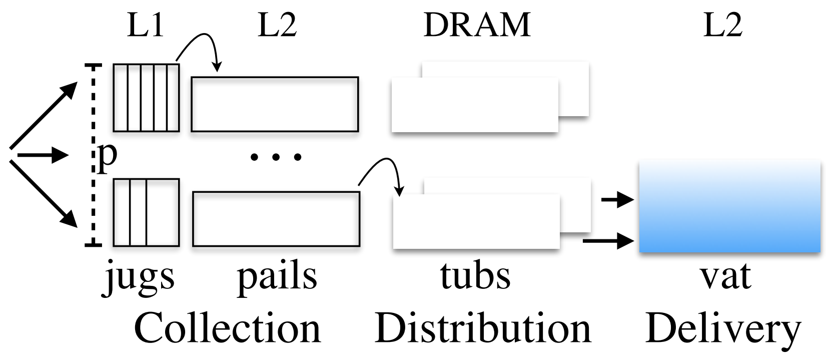


Figure 3-8: Containers for DRAM-conscious clustering.

streaming and partition indexing can take advantage respectively of 256-bit Advanced Vector Extensions (AVX2) instructions and Bit Manipulation Instructions (BMI).

A *tub* is a large DRAM-resident container used for multi-pass processing. When full or finalized, tubs are linked into thread-local lists and eventually added to global lists. Tubs are tagged with the creating thread-ID if access to thread-local variables is needed in a captured continuation. TLB reach and fan-out constrain the maximum size of a single tub. All active tubs for each thread are TLB-reachable, achieved at the cost of additional memory fragmentation.

A *vat* is a random access cache-resident container for mixing *tubs* targeting a fixed output subset. Depending on the expected spatial or temporal locality different variants are selected. The simplest vat handles the case of expected high spatial locality with dense range updates, e.g., when every cache line is used and the full content of each cache line is overwritten. Other specializations handle low density or low coverage cases, e.g., sparse cache-line writes when few cache lines within a range are modified, or when partial cache line content is either preserved or can be overwritten.

Milk containers are by default emptied in Last-In First-Out (LIFO) order to maximize cache locality. LIFO order is acceptable, when the sequential consistency of a `parallel for` is acceptable. Even if deterministic, this order does not match serial program semantics when that may be desirable, e.g., for `a[ind[i]] = i` to observe the last write. If `ordered` processing is requested, containers are emptied in First-In First-Out (FIFO) order, and a Milk program can match the output of a serial program.

3.5.2.1 Container Sizing to Hardware

For a desired cache level and a given cache budget, the total number of partitions (i.e., height) has priority over pail width. The height and the range of valid indices determine the number of partitioning passes, e.g., we can partition using 9-bit partitioning at every pass with height 512, while height 256 uses 8-bit partitioning that may require an additional pass to partition the full input.

Wider pails improve DRAM row hits on writes to tubs, as these are otherwise random cache-line writes. Pails width, however, is limited by cache capacity, including the need to fit cache resident additional application data and code during Collection.

Tubs are limited by physical memory capacity. Memory management is a simple linked list of chunks. Tub layout is guided by low-level optimization opportunities: the TLB entries are a per-core resource, thus the threads on a core fit all their active tubs within TLB reach; the DRAM banks are a global resource, thus the active tubs are interleaved across all threads on a socket to maximize DRAM open page hits.

3.5.2.2 Container Sizing for Applications

The maximum memory that can be allocated for temporary space during Distribution is the most important attribute, set with `milk_set_max_memory`. If memory limits are reached this will force compaction and/or BSP superstep splitting. Applications that target Milk-only execution can eschew double-buffering of superstep outputs, but must disallow superstep splitting with `milk_set_strict_bsp`.

Tuning hints that limit the container sizes are expressed from the perspective of the user program, i.e., L1, L2, L3 cache sizes and TLB entries that should be reserved for cache resident data and code that are not managed by Milk. Container implementation details do not need to be exposed to users. When optimal cache-resident size cannot be inferred, tuning hints (i.e., bytes touched per `tag`) control the number of partitions and fan-out. External library footprint can be summarized with the `milk_touch` intrinsic, e.g., a call to a function that touches 3 floats on average for the expected data distribution can be accompanied by a call to `milk_touch(3*sizeof(float))`.

3.6 Applications

To demonstrate the applicability and performance improvements provided by MILK, we use the five fundamental graph applications from the GAP Benchmark Suite [17, 18] building on the high performance OpenMP reference implementations.

In Figures 3-9, 3-10, 3-11, 3-12, and 3-13, we show the concise **`milk`** implementations of the inner loops of the kernels. These are also valid serial programs without **`milk`**, but parallel variants need additional compare-and-swap (CAS) logic as noted on lines 16, 31, and 102, e.g., line 102 replaces ten lines of a compare-and-swap loop in the reference implementation. Thread-local data structures are accessed using the delivery thread's ID, therefore the TID macro on lines 17, 32, and 103 is simply `omp_get_thread_num`.

Betweenness Centrality (BC). BC is an important graph algorithm that analyzes the relative importance of nodes based on the number of shortest paths going through each, e.g., to identify critical nodes in power grids, terrorist communication networks, or HIV/AIDS propagation [36, 114]. BC is also an integral building block for many state-of-the-art community detection algorithms [134].

The forward- and backward- step kernels of BC are shown in Figure 3-10. The continuation block line 28 defers random access to the neighbors of u, and `packs`

```
Graph<Node> g;
Queue<Node> queue;   // global
Queue<Node>* lqueue; // per-thread

vector<Node> parent;
long outedges;

void BFS_TopDown(int depth) {
  #pragma omp parallel for milk reduction(+ : outedges)
  for (Node u : queue)
    for (Node v : g.out_neigh(u))
    #pragma milk pack(u : min) tag(v)
    {
        Node curr = parent[v];
        if (curr < 0) {
          parent[v] = u;  // CAS
          lqueue[TID].push_back(v);
          outedges += -curr;
        }
    }
}
```

Figure 3-9: Breadth-First Search with **milk**

the number of paths to be added. An optional sum combiner can accumulate the contributions along all incoming edges of v. For backward propagation on line 50, we use a similar floating point combiner for each contribution.

Breadth-First Search (BFS). BFS traversal is an important component of larger graph kernels, such as BC. BFS is also used in path analytics on unweighted graphs. Figure 3-9 shows a **milk** version of TopDown, which tracks the parent of each vertex. Unlike the traversal in BC above, or the variant in Figure 3-6 which tracks only depths of each vertex, this variant tracks parents for validation. Therefore, source vertex id is `pack`ed along each visited edge, while a `min` combiner ensures deterministic results. Adaptive optimizations in state of the art implementations [16] also depend on a sum reduction of the outdegrees of all vertices added to the BFS frontier.

Connected Components (CC). CC is critical for understanding structural properties of a graph. It finds connected components in a large graph by propagating the same label to all vertices in the same connected component.

Our baseline implements Shiloach-Vishkin's algorithm [169] with optimizations [10]. The **milk** implementation of CC is shown in Figure 3-11. While the output of CC is deterministic, traditional implementations have non-deterministic performance, i.e., the number of iterations is dependent on race resolution of line 68, and results in 20% run-to-run variation. While the BSP model does not allow a synchronous step to observe updates within the same step, Milk allows updates during Delivery, e.g., `comp[v]` references can still observe simultaneous updates by other neighbors. A

```
vector<Node> paths, depths;
void Brandes_Forward(int depth) {
  #pragma omp parallel for milk
  for (Node u : queue) {
    Node pathsU = paths[u];
    for (Node v : g.out_neigh(u))
      #pragma milk pack(pathsU : +) tag(v)
      {
        if (depths[v] == -1) {
          depths[v] = depth;   // CAS
          lqueue[TID].push_back(v);
        }
        if (depths[v] == depth)
          #pragma omp atomic if(!milk)
          paths[v] += pathsU;
      }
  }
}

vector<Queue<Node>::iterator> wave;
vector<float> deltas;
void Brandes_Reverse(int d) {
#pragma omp parallel for milk
  for (auto it = wave[d-1];
            it < wave[d]; it++) {
    Node v = *it;
    float contribV = (1 + deltas[v]) /
                          paths[v];
    for (Node u : g.in_neigh(v)) {
#pragma milk pack(contribV : +) tag(u)
      if (depths[u] == d - 1)
#pragma omp atomic if(!milk)
        deltas[u] += paths[u] * contribV;
    }
  }
}
```

Figure 3-10: Betweenness Centrality with **milk**

```
vector<Node> comp;
bool change;

void ShiloachVishkin_Step1() {
  #pragma omp parallel for milk
  for (Node u = 0; u < g.num_nodes(); u++) {
    Node compU = comp[u];
    for (Node v : g.out_neigh(u))
      #pragma milk pack(compU : min) tag(v)
      if ((compU < comp[v]) &&
          (comp[v] == comp[comp[v]])) {
        comp[comp[v]] = compU;
        change = true;
      }
  }
}
```

Figure 3-11: Connected Components with **milk**

```
vector<float> contrib, new_rank;

void PageRank_Push() {
  #pragma omp parallel for milk
  for (Node u=0; u < g.num_nodes(); u++) {
    float contribU = contrib[u];
    for (Node v : g.out_neigh(u))
      #pragma milk pack(contribU : +) tag(v)
      #pragma omp atomic if(!milk)
      new_rank[v] += contribU;
  }
}
```

Figure 3-12: Page Rank with **milk**

`min` combiner allows deterministic performance.

PageRank (PR). PageRank is an iterative algorithm for determining influence within a graph, initially used to sort web search results [142]. The PageRank-Delta variant [121] more efficiently propagates updates from vertices that receive large deltas between iterations. The commonly benchmarked in prior work, however, traditional algorithm propagates till convergence contributions from all vertices in every iteration as shown in Figure 3-12.

Single-Source Shortest Paths (SSSP). For weighted graphs, SSSP computes the distance from a start vertex to all reachable vertices, e.g., travel distance or travel time on a road network. Δ-stepping [124] is among the few parallel SSSP variants that outperform a well-optimized serial Dijkstra algorithm. Figure 3-13 shows Δ-stepping with **milk**.

According to the benchmark specifications all non-pointer data types are 32-bit,

```
WeightedGraph<Node> wg;
vector<Weight> dist;

Weight delta;
vector<Node> frontier;
vector<vector<Node>>* lbins; // per-thread

void DeltaStep_Relax(int bin) {
  #pragma omp parallel for milk
  for (Node u : frontier) {
    if (dist[u] < delta * bin)
      continue;
    for (WEdge e : wg.out_neigh(u)) {
      Weight newDist = dist[u] + e.weight;
      Node v = e.dest;
      #pragma milk pack(newDist : min) tag(v)
      if (dist[v] > newDist) {
        dist[v] = newDist; // CAS
        lbins[TID][newDist/delta].push_back(v);
      }
    }
  }
}
```

Figure 3-13: Single Source Shortest Path with **milk**

therefore all tag keys are 32-bit integers, and pack values are 32-bit integer or floating point numbers. We do not modify the precision of floating point numbers in pack types or combiner accumulators.

3.7 Evaluation

We compare the performance of MILK on top of clang 3.7 to an OpenMP baseline, on the graph applications from GAPBS [17, 18] v0.8 using the modified kernels shown in Section 3.6 with **milk** extensions. We report end-to-end performance gains while exploring the effects of varying numbers of vertices and edges, as well as sensitivity to available temporary memory.

3.7.1 System Characterization

Table 3.4 summarizes the characteristics of the evaluated platform which has high memory bandwidth per core, as well as low latency. We measured achievable read and streaming write bandwidths using all cores to be >90% of 6.5 GB/s per-core DRAM theoretical bandwidth. Idle and loaded latencies are measured using MLC [66]. We cross-checked with perf that these measurements match the average L1 miss latency

Table 3.4: System specifications [2] and *measurements*.

Cores	**Cache**		**TLB**	
	L2	L3 Shared	Entries	Reach
4	256 KB	8 MB	1024	2 GB

Core	**DRAM Bandwidth**			**Latency**
Frequency	Peak	*Read*	*Write*	*Read*
4.2 GHz	26 GB/s	*92* %	*98* %	*58–130* ns

of 452 cycles on DRAM-bound references, e.g., PageRank on uniform graphs, which sustains Memory Level Parallelism (MLP) of 8 outstanding L1 misses.

The Haswell microarchitecture has much improved TLB reach compared to previous generations: each core can reach up to a 2 GB working set with low TLB overhead. For larger working sets, 2 MB page table entries are cache resident, and a TLB miss does not access DRAM.

3.7.2 Speedup

We show end-to-end speedups, ranging from 1.4× on CC and 2× for BFS, to 2.7× on PR, and analyze Milk's performance on synthesized random graphs with variable working set size and temporal and spatial locality.

Figure 3-14 shows speedups when vertices range from V=2^{21} to V=2^{25}, with an average degree of 16 in uniform degree distribution (i.e., Uniform⟨21..25⟩). We show two BFS variants – BFSd from Graph500 [52] (in Figure 3-6), as well as BFSp from GAPBS (Figure 3-9). The two variants help compare effects of payload size: the BFSd variant has no payload as it only tracks the constant per superstep depth of each vertex, while the BFSp variant uses more bandwidth as it needs to `pack` the parent ID. On CC, Milk per-iteration speedup is stable, however, CC is sensitive to vertex labeling and race resolution. These effects result in non-deterministic iteration count and per-iteration time, the former may vary by up to 20% in the baseline. All other applications have under 2% noise between runs.

Figure 3-14 also shows the characteristic transition from L3 to DRAM-bound references with a corresponding increase in Milk speedups. The randomly accessed data per vertex varies for the different benchmarks, e.g., either 1 or 3 random references to 32-bit values in BC, vs. a single 32-bit value in the other benchmarks. At $V = 2M$ the total working set is already larger than cache size for BC, while for the other applications requests are mostly served by the L3 cache. Once the working set size more than doubles the cache capacity, very little locality is captured by hardware caches and the baseline random accesses run at DRAM latency, while Milk's access time per indirect reference remains constant.

In Figure 3-15, we compare performance while varying vertex degrees to show sufficient spatial locality can be exploited even at a low average degree. We use the CountDegree (Histogram) kernel of Graph500, shown in Figure 3-7, which is used to construct a graph data structure from a list of edges in graph applications, and

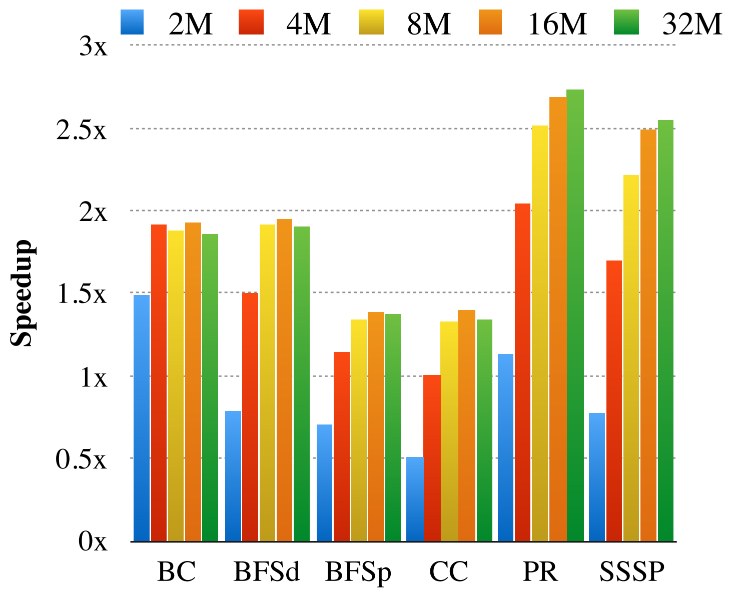


Figure 3-14: Overall speedup with MILK on random graphs with 2—32 million vertices, d=16(uniform), 8 MB L3.

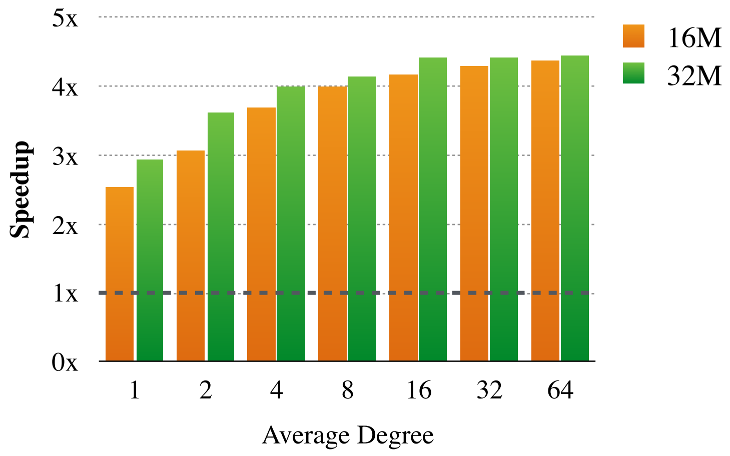


Figure 3-15: MILK Speedup on Histogram, counting degree of 16M or 32M vertices with 16M–2B edges.

is critical in other applications [82]. Figure 3-15 shows speedups on Histogram in uniform distributions with average degrees from 1 to 64. Low degree graphs have lower temporal and spatial locality, and only 2.5× speedup. When the average degree is larger than 16, however, all vertices have an incident edge, and spatial locality reaches 100%, i.e., all words accessed per cache line are useful. At degree 16 and higher, we gain 4.2×–4.4×.

Most real-world web and social graphs have much higher degrees than 16, even within subgraphs of graphs partitioned across distributed systems. Characterization of actual social network graphs [9, 34] shows that a sample of ∼300 million Twitter users have ∼200 followers on average, and Facebook's ∼1.5 billion users have 290 friends on average.

Real-world graphs also exhibit better cache hit rates than synthetic uniformly random graphs, due to power law structure, and to a lesser extent due to graph community structure. While current hardware caches are able to capture some locality in power law graphs [18], the actual social and web graphs are orders of magnitude larger than public datasets. We tested on 8 MB L3 cache size in order to simulate the effects of larger graphs while preserving the graph structure. On Twitter(V=62M, d=24) we observe 20–50% L3 hits, and only 5–20% L2 hits.

We evaluated performance while varying the size of synthetic power law graphs – R-MAT [30] used for Graph500 benchmarks as a proxy for real-world graph structure. On RMAT25 (V=32M,d=16) CountDegree achieves 3.2× speedup. Spatial locality is lower here since half of the generated vertices have degree 0, while the hottest vertices are cache-resident, e.g., the top 3% of vertices have 80% of the edges. Although the majority of edges are incident on hot vertices, processing hot vertex edges is faster than handling indirect references that miss caches. Speedups on RMAT25 are comparable to speedups on smaller size uniform graphs; e.g., BC goes from 1.9× speedup on Uniform25 to 1.3× on RMAT25, and SSSP goes from 2.5× to 2.2×.

Filtering accesses to hot references is necessary for effective use of both hardware caches and Milk. In graphs, hot references to high-degree vertices are easy to identify as described in Section 3.4.1.1.

3.7.3 Performance Analysis

Cache miss rates and cycles stalled metrics based on hardware performance counters indicate that Milk effectively converts random DRAM references into cache hits.

Figure 3-16 shows that effective blocking at L2-cache sized partitions dramatically reduces cycles stalled on L2 and L3 misses. While a Haswell core can execute 4 µops per cycle at ideal ILP, these measurements show cycles with complete execution stalls while there is a pending cache miss, i.e., waiting on L3 or DRAM.

Figure 3-17 further breaks down L3 misses by access type – indirect memory references appear as demand loads and stores, while most sequential accesses are prefetched. With Milk, demand accesses are no longer a major contributor to cache misses, and most misses are on sequential requests.

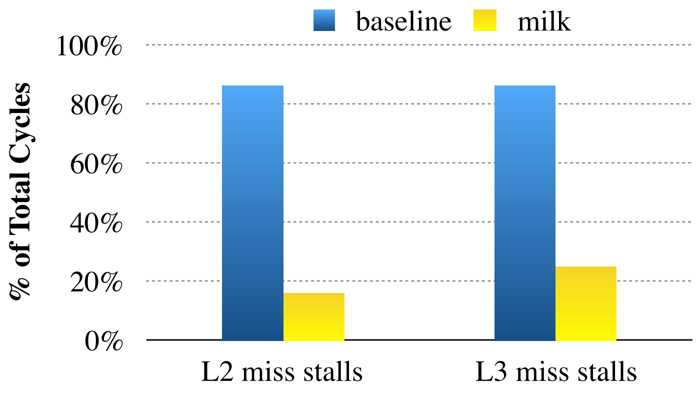


Figure 3-16: Stall cycle reduction with MILK on PR, V=32M, 256 KB L2, 8 MB L3.

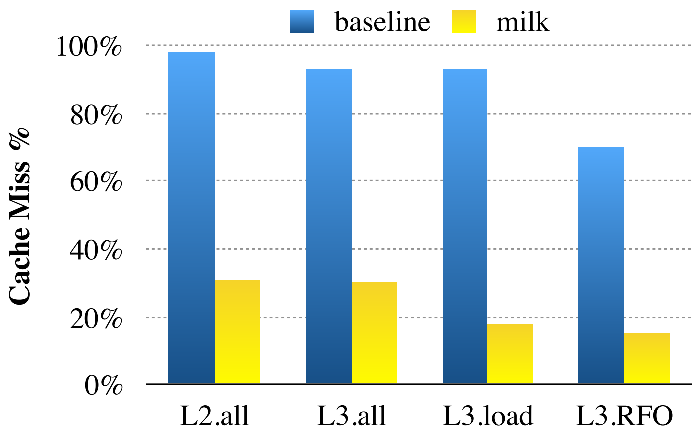


Figure 3-17: Cache miss rate reduction with MILK on PR, V=32M, 256 KB L2, 8 MB L3. L3 miss rates show all references (including prefetches), non-prefetch loads, and non-prefetch store Requests For Ownership (RFOs).

Table 3.5: Maximum memory overhead of Milk. Measured as maximum resident size over baseline for V=32M. `tag` clauses, and where needed `pack`, use 32-bit variables.

Kernel	**Overhead**	**pack**	**Tag + Pack**
	% Resident	Type	Bytes
BC	104 %	int/float	8
BFS_Depth	58 %	—	4
BFS_Parent	113 %	int	8
CC	176 %	int	8
PR	184 %	float	8
SSSP	4 %	int	8

3.7.4 Space Overhead

Milk's maximum space overhead is proportional to the number of references per superstep, but even within a fraction of the maximum space, applications with ample spatial locality attain good performance. For example, at only 12% memory overhead PageRank maintains $> 2.7\times$ speedup. For graph applications, the number of indirect references is a function of the number of active edges accessed during each iteration. Most algorithms make a different number of indirect references during each iteration: few edges are processed in BC/BFS, very few in SSSP, and only CC and PR always touch all edges. If less memory than the maximum required is available, a superstep will have to be forcefully divided, and may not be able to capture all available locality. Superstep splitting has minimal performance impact when temporal locality is high, e.g., as shown in Figure 3-15 where splitting is required on graphs with degree 64.

Table 3.5 shows the maximum additional memory required on the evaluated kernels without memory limits, compression, filters, or combiners. Baseline memory consumption varies based on the graph representation and other algorithm overheads, e.g., unweighted graph plus 1 bit per edge for BC vs. a weighted graph plus additional 4 bytes per edge for SSSP. Milk memory consumption is primarily a function of the number of indirect references per superstep, multiplied by the size of the `pack` and `tag` variables. For example, BC forward traversal carries an `int`, and backward traversal – a `float`. Small additional overhead is added by bookkeeping and internal fragmentation in Milk containers.

3.8 Related Work

Milk's distribution phases are most similar to the heavily optimized table join operations in relational databases. The two best join methods are to sort both database relations, or to build a hash table with tuples of one relation and then probe the hash table for each tuple of the other relation. Increasing data sizes and perennial hardware changes have fueled continuous innovation: Shatdal et al. [167] demonstrated hash join with cache partitioning, further improved to consider TLB impact [122], and to optimize for SIMD, NUMA, non-temporal write optimizations, software buffering, etc [4, 12, 88, 179, 191]. Our runtime library data structures and compiler generated code are designed for modern microarchitectures (e.g., on Haswell TLB overhead is less critical), and focus on avoiding increasingly important DRAM contention and turnaround delays. Milk's compiler support for indirect expressions is more general than database join, and we are the first to use similar techniques to accelerate graph applications.

Milk allows optimized iterators and algorithm-specific data structures to use familiar and concise indirect array accesses, while detailed machine-specific optimizations are left to the compiler and runtime library. Graph processing is an important application that we have shown significantly benefits from Milk optimizations. Graph applications expose irregular access patterns that render traditional compiler optimizations ineffective [146]. Creating efficient shared memory graph frameworks [135, 170] has received considerable research attention, but overheads and abstraction mismatch lead to significant slowdown compared to hand-optimized code [163]. More detailed analysis of partitioned processing of the classic PageRank algorithm in DRAM is presented in [15]. More recently, hardware-accelerated external sorting [81] makes possible graph analytics using SSDs, which would otherwise have impractically slow random accesses.

The inspector/executor [158] family of two-phase optimizations inspect indirect references before an alternative execution. The original optimization [158] rearranged the base array A in `A[ind[i]]` not the index, as it targeted better scheduling for message passing machines. Follow-up research explored two main directions: either very cheap per-step index array analysis (like determining bounding box (min/max) for MPI and NUMA boundary exchanges [176]), or very expensive data reorganizations to be applied as pre-processing steps. Coalescing references targeting different cores to reduce MPI messaging overhead is explored for the HPCC RandomAccess benchmark [50]. The workload is similar to Histogram of degree 4 (Figure 3-15) but disallows BSP semantics. Since the benchmark allows at most tens of requests per core to be reordered, short coalescing buffers are needed, yet cache resident buffers can discover very little locality in DRAM-sized working sets. In shared-memory settings, sharding memory and coalescing requests to owning cores eliminate atomics at the cost of message-passing overhead. Inexpensive methods, however, do not improve the performance of DRAM indirect references.

Expensive locality reorganization methods have not been able to amortize costs within a single iteration, and they are limited to applications that repeatedly process the same references, e.g., rearranging the index array [41, 125] and remapping all arrays in a loop, or graph partitioning [57, 58] or cheaper reorderings with lower benefits (e.g., space filling curves). Recent inspector/executor work [42] traded lower cache hit rate for improvement of DRAM row buffer hits for 14% net gains. Milk achieves up to 4× gains on static reference loops, and pays off in one iteration to also allow dynamic references.

3.9 Future Work and Hardware Trends

We highlight several directions for future work to take advantage of hardware trends.

3.9.1 Larger L2 caches

Radix partitioning is sensitive to the size of L2 caches, and to the ratio between sequential and random memory bandwidth for a given memory technology. The cache size determines the number of passes required for a given data range, while the sequential bandwidth advantage determines the maximum number of passes viable. Both of these parameters are growing through dramatic improvements.

For example, the L2 cache size on the Haswell systems evaluated above was 256KB, while recent L2 caches on Intel Skylake CPUs have grown to 1MB, a two-pass radix partitioning can now cover a 16× larger memory range.

3.9.2 3D Chip Integration for CPU-DRAM

The sequential DRAM bandwidth will increase much faster than random DRAM access bandwidth, with improved CPU-DRAM packaging. A dramatic increase in the number of pins will be possible with upcoming 3D integration, including 2.5D active interposers (at 7 nm), while DRAM chips have the same latency. This increased advantage will allow scaling to even larger memory sizes, and even applications with lower temporal and spatial locality would be able to benefit from Milk.

3.10 Summary

We introduced the **`milk`** C/C++ language extension in order to capture programmer's intention when making indirect memory references, while the compiler and runtime system choose the most efficient access order. As demonstrated, the annotations can easily be added across a suite of popular graph applications and when compiled with Milk enable performance gains of up to 3×.

We believe the right level of abstraction to enable high-performance applications for graph analytics, machine learning, or in-memory databases is not necessarily a domain specific framework, nor an optimized collection of algorithms — i.e., hash vs. sort join, nor a data structure — i.e., bit vector vs. hash table. Instead, best performance and flexibility are enabled by the most broadly relied upon low-level primitive — the indirect memory reference — and just adding **`milk`**.

Chapter 4

Cimple: Uncovering Instruction and Memory Level Parallelism

If it is fast and ugly, they will use it and curse you;
if it is slow, they will not use it.

David Cheriton

In Jain, *The Art of Computer Systems Performance Analysis* [71]

4.1 Overview

Modern out-of-order processors have increased capacity to exploit instruction level parallelism (ILP) and memory level parallelism (MLP), e.g., by using wide superscalar pipelines and vector execution units, as well as deep buffers for in-flight memory requests. These resources, however, often exhibit poor utilization rates on workloads with large working sets, e.g., in-memory databases, key-value stores, and graph analytics, as compilers and hardware struggle to expose ILP and MLP from the instruction stream automatically.

In this chapter, we first introduce the **IMLP** (Instruction and Memory Level Parallelism) task programming model. IMLP tasks execute as coroutines that yield execution at annotated long-latency operations, e.g., memory accesses, divisions, or unpredictable branches. IMLP tasks are interleaved on a single thread, and integrate well with thread parallelism and vectorization. Cimple [93], is our DSL embedded in C++, which allows exploration of task scheduling and transformations, such as buffering, vectorization, pipelining, and prefetching.

Cimple is intended for critical hot spots with low MLP in in-memory databases, graph queries, ML inference, and web search. We demonstrate state-of-the-art performance on core algorithms that operate on arrays, hash tables, trees, and skip lists. Cimple applications reach 2.5× throughput gains over hardware multithreading on a multi-core, and 6.4× single thread speedup.

The rest of this chapter is organized as follows. We motivate the need for Cimple in Section 4.2, and we walk through an end-to-end use case in Section 4.3. We introduce the IMLP programming model and Cimple compiler and runtime library design in Section 4.4, with more details of the Cimple DSL in Section 4.5, and implementation details of Cimple transformations in Section 4.6. We demonstrate expressiveness by building a template library of core indexes in Section 4.7, and performance – in Section 4.8. Section 4.9 surveys related work, in Section 4.10 we highlight directions for future work, and Section 4.11 summarizes this chapter.

4.2 Motivation

Barroso et al. [14] observe that "killer microseconds" prevent efficient use of modern datacenters. The critical gap between millisecond and nanosecond latencies lies outside the traditional roles of software and hardware. Existing software techniques used to hide millisecond latencies, such as threads or asynchronous I/O, have too much overhead to successfully address microsecond latencies and below. On the other hand, out-of-order hardware is capable of hiding at most tens of nanoseconds latencies. Yet, average memory access times now span a much broader range: from ~20 ns for L3 cache hits, to more than 200 ns for DRAM accesses on a remote NUMA node — making hardware techniques inadequate. In this chapter we will show that an efficient, flexible, and expressive programming model can fill the critical gap and scale the full memory hierarchy from tens to hundreds of nanoseconds.

Processors have grown their capacity to exploit instruction level parallelism (ILP) with wide scalar and vector pipelines, e.g., cores have 4-way superscalar pipelines, and vector units can execute 32 arithmetic operations per cycle. Memory level parallelism (MLP) is also pervasive, with deep buffering between caches and DRAM that allows 10+ in-flight memory requests per core. But long distances between independent operations in existing instruction streams prevent modern CPUs from fully exploiting this source of performance.

Critical infrastructure applications such as in-memory databases, key-value stores, and graph analytics, characterized by large working sets with multi-level address indirection and pointer traversals, push hardware to its limits: large multi-level caches and branch predictors fail to keep processor stalls low. Out-of-order windows of hundreds of instructions are also insufficient to hold all instructions needed in order to maintain a high number of parallel memory requests, which is necessary to hide long latency accesses.

The two main problems are caused by either branch mispredictions that make the effective instruction window too small, or by overflowing the instruction window when there are too many instructions between memory references. Since a pending load prevents all following instructions from retiring in-order, if the instruction window resources cannot hold new instructions, no concurrent loads can be issued. A vicious cycle forms where low ILP causes low MLP when long dependence chains and mispredicted branches do not generate enough parallel memory requests. In turn, low MLP causes low effective ILP when mispredicted branch outcomes depend on long latency memory references.

Context switching using a high number of hardware threads to hide DRAM latency was explored in Tera [5]. Today's commercial CPUs have vestigial simultaneous multithreading support, e.g., 2-way SMT on Intel CPUs. OS thread context switching is unusable as it is 50 times more expensive than a DRAM miss. We therefore go back to 1950s coroutines [132] for low latency software context switching in order to hide variable memory latency efficiently.

We introduce a simple Instruction and Memory Level Parallelism (IMLP) programming model based on concurrent tasks executing as coroutines. Coroutines yield execution at annotated long-latency operations, e.g., memory accesses, long dependence chains, or unpredictable branches. Our DSL Cimple (Coroutines for Instruction and Memory Parallel Language Extensions) separates program logic from programmer hints and scheduling optimizations. Cimple allows exploration of task scheduling and techniques such as buffering, vectorization, pipelining, and prefetching, supported in our compiler **Cimple** for C++.

Prior compilers have struggled to uncover many opportunities for parallel memory requests. Critical long latency operations are hidden in deeply nested functions, as modularity is favored by current software engineering practices. Aggressive inlining to expose parallel execution opportunities would largely increase code cache pressure, which would interact poorly with out-of-order cores. Compiler-assisted techniques depend on prefetching [109, 127], e.g., fixed look-ahead prefetching in a loop nest. Manual techniques for indirect access prefetching have been found effective for the tight loops of database hash-join operations [31, 95, 123, 147] - a long sequence of index lookups can be handled in batches (*static scheduling*) [31, 123], or refilled dynamically (*dynamic scheduling*) [95, 147]. Since the optimal scheduler style may be data type and data distribution dependent, Cimple allows generation of tasks for both styles, additional code-generation optimizations, as well as better optimized schedulers.

High performance database query engines [40, 100, 123, 131] use Just-In-Time (JIT) compilation to remove virtual function call overheads and take advantage of attendant inlining opportunities. For example, in Impala, an open source variant of Google's F1 [171], query generation uses both dynamically compiled C++ text and LLVM Instruction Representation (IR) [35]. Cimple offers much higher performance with lower complexity than using an LLVM IR builder: Cimple's Abstract Syntax Tree (AST) builder is close to C++ (and allows verbatim C++ statements). Most importantly, low level optimizations work on one item at a time, while Cimple kernels operate on many items in parallel.

We compare Cimple performance against core in-memory database C++ index implementations: binary search in a sorted array, a binary tree index lookup, a skip list lookup and traversal, and unordered hash table index. As shown on Figure 4-1, we achieve 2.5× peak throughput on a multi-core system, and on a single-thread — 6.4× higher performance.

Traditional database indices may even fall back to pointer chasing of linked leaf nodes on a range query or an index scan, which is actually the worst case — no memory level parallelism (MLP=1). Cimple reaches 9.9× higher single-thread throughput when traversing linked lists on Haswell CPUs (we explored peak ILP and MLP capabilities of modern hardware and trends in Section 2.2).

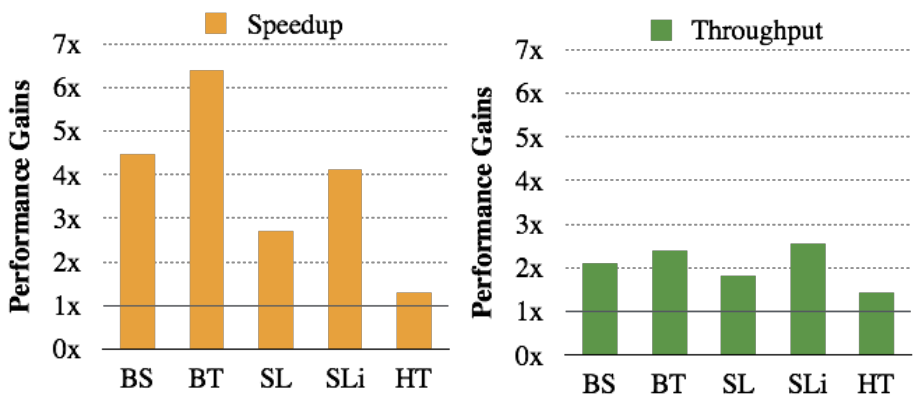


Figure 4-1: Speedup on a single thread and throughput gains on a full system (24 cores / 48 SMT [45]). **B**inary **S**earch, **B**inary **T**ree, **S**kip **L**ist, **S**kip **L**ist **i**terator, and **H**ash **T**able.

4.3 Example

We next present an example that highlights how the Cimple language and runtime system work together to efficiently expose available memory-level parallelism on current hardware. We use a classic iterative binary search tree lookup, which executes a while loop to traverse a binary tree in the direction of `key` until a match is found. It returns the node that contains the corresponding key/value pair, or `nil`.

4.3.1 Binary Search Tree Lookup in Cimple

Figure 4-2 presents the Cimple code for the example computation. The code identifies the name of the operation (`BST_find`), the result type (`node*`), and the two arguments (`n`, the current node as the computation walks down the tree, and `key`, the key to lookup in the tree).

In this code, there is one potentially expensive memory operation, specifically the first access to `n->key` in the if condition that checks to see if the key at the current node `n` matches the lookup `key`. Once the cache line containing this value has been fetched into the L1 data cache, subsequent accesses to `n->key` and `n->child` are accessed quickly. The Cimple code issues a prefetch, then yields to other lookup operations on the same thread.

Figure 4-3 presents the coroutine that our Cimple compiler (automatically) generates for the code in Figure 4-2. Each coroutine is implemented as a C++ struct that stores the required state of the lookup computation and contains the generated code that implements the lookup. The computation state contains the `key` and current node `n` as well as automatically generated internal state variables `_result` and `_state`. Here after the Cimple compiler has decomposed the lookup computation into individual steps, the computation can be in one of three states:

```
auto c = Coroutine(BST_find);
c.Result(node*).
Arg(node*, n).
Arg(KeyType, key).
Body().
  While(n).Do(
    Prefetch(n).Yield().
    If( n->key == key ).
    Then( Return(n) ).
    Stmt( n = n->child[n->key < key]; )
  ).
  Return(n);
```

Figure 4-2: Binary Search Tree Lookup in Cimple.

```
struct Coroutine_BST_Find {
  node* n;
  KeyType key;
  node* _result;
  int _state = 0;
  enum {_Finished = 2};

  bool Step() {
    switch(_state) {
    case 0:
      while(n) {
        prefetch(n);
        _state = 1;
        return false;
    case 1:
        if(n->key == key) {
          _result = n;
          _state = _Finished;
          return true;
        }
        n = n->child[n->key < key];
      } // while
      _result = n;
      _state = _Finished;
      return true;
    case _Finished:
      return true;
  }}};
```

Figure 4-3: Generated Cimple coroutine for BST_find.

Before Node: In this state the lookup is ready to check if the current node `n` contains `key`. However, the required access to `n->key` may be expensive. The step therefore issues a prefetch on `n` and returns back to the scheduler. To expose additional memory level parallelism and hide the latency of the expensive memory lookup, the scheduler will proceed on to multiplex steps from other lookup computations onto the scheduler thread.
At Node: Eventually the scheduler schedules the next step in the computation. In this step, the prefetch has (typically) completed and `n` is now resident in the L1 cache. The computation checks to see if it has found the node containing the key. If so, the lookup is complete, the coroutine stores the found node in `_result`, and switches to the Finished state. Otherwise, the coroutine takes another step left or right down the search tree, executes the next iteration of the while loop to issue the prefetch for left or right node, and then returns back to the scheduler.
Finished: Used only by schedulers that execute a batch of coroutines that require different number of steps.

4.3.2 Request Parallelism

Cimple converts available Request Level Parallelism (RLP) into memory-level parallelism (MLP) by exposing a queue of incoming requests to index routines, instead of queuing or batching in the network stack [113]. Our example workload is inspired by modern Internet servers [7, 25, 156] that process a high volume of aggregated user requests. Even though the majority of requests are for key lookups, support for range queries requires an ordered dictionary, such as a binary search tree or a skip list. Here each worker thread is given a stream of independent key lookup requests.

A coroutine *scheduler* implements a lightweight, single-threaded queue of in-flight partially completed request computations (e.g., BST lookups). The scheduler multiplexes the computations onto its thread at the granularity of steps. The queue stores the state of each partially completed computation and switches between states to multiplex the multiple computations. The Cimple implementation breaks each computation into a sequence of steps. Ideally, each step performs a sequence of local computations, followed by a prefetch or expensive memory access (e.g., an access that is typically satisfied out of the DRAM), then a yield.

Note we never wait for events, since loads are not *informing* [64]. We simply avoid reading values that might stall. This is the fundamental difference between Cimple and heavy-weight event-driven I/O schedulers. We also avoid non-predictable branches when resuming coroutine stages.

We maintain a pipeline of outstanding requests that covers the maximum memory latency. The scheduler queue has a fixed number of entries, e.g., ~50 is large enough to saturate the memory level parallelism available on current hardware platforms. The scheduler executes one step of all of the queued computations. A queue refill is requested either when all lookups in a batch complete (*static scheduling* [31]), or as soon as any lookup in a batch has completed (*dynamic scheduling* [95]). The scheduler then returns back to check for and enqueue any newly arrived requests. In this way, the scheduler continuously fills the queue to maximize the exploited memory level

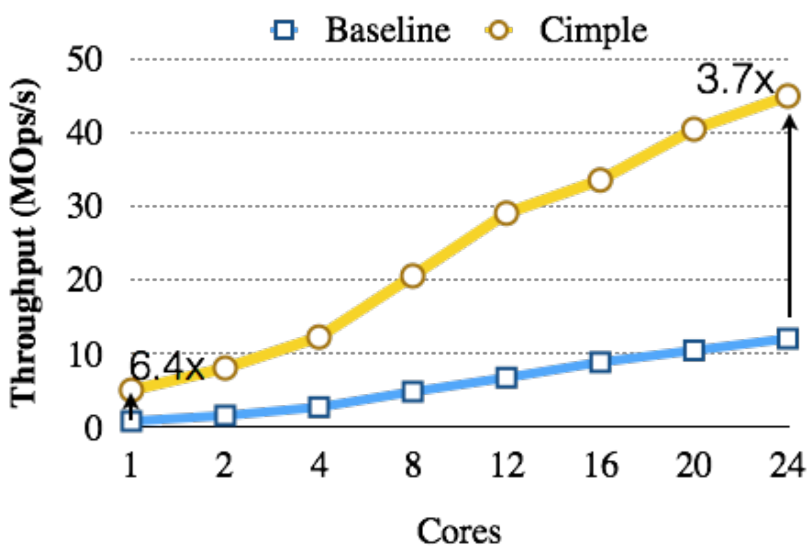


Figure 4-4: Throughput improvements for lookup in a partitioned binary search tree index (1GB per thread).

parallelism.

4.3.3 Cimple Execution On Modern Computing Platform

For large binary search trees, the aggregated lookup computation is memory bound. Its performance is therefore determined by the sustained rate at which it can generate the memory requests required to fetch the nodes stored in, e.g., DRAM or other remote memory. Our target class of modern microprocessors supports nonblocking cache misses, with up to ten outstanding cache misses in flight per core at any given time. The goal is therefore to maximize the number of outstanding cache misses in flight, in this computation by executing expensive memory accesses from different computations in parallel.

Here is how the example Cimple program works towards this goal. By breaking the tree traversals into steps, and using the Cimple coroutine mechanism to quickly switch between the lookup computation steps, the computation is designed to continuously generate memory requests (by issuing prefetch operations from coroutined lookups). This execution strategy is designed to generate an instruction stream that contains sequences of fast, cache-local instructions (from both the application and the Cimple coroutine scheduler) interspersed with prefetch operations. While this approach has instruction overhead (from the Cimple coroutine scheduler), the instructions execute quickly to expose the available MLP in this example.

4.3.4 Performance Comparison

We compare the performance of the Cimple binary tree lookup with the performance of a baseline binary tree lookup algorithm. The workload is a partitioned tree search in which each thread is given a stream of lookups to perform. The Cimple implementation interleaves multiple lookups on each thread, while the baseline executes the lookups sequentially. We use a 24 core Intel Haswell machine with 2 hyperthreads per core (see Section 4.8.1).

Figure 4-4 presents the results. The X-axis is the number of cores executing the computation, with each core executing a single lookup thread. The Y-axis presents the number of lookups in millions of operations per second. On one thread, the Cimple computation performs 6.4 times as many lookups per second as the baseline computation. This is true even though 1) due to coroutine scheduling overhead, the Cimple computation executes many more instructions than the baseline computation and 2) in theory, the baseline computation has as much memory parallelism across all requests as the Cimple computation (but the baseline MLP is unavailable to the processor because it is separated within the instruction stream by the long sequential lookups).

The performance of both computations increases up to 24 cores, with the Cimple implementation performing 3.7 times as many lookups per second as the baseline implementation (the difference narrows because the memory and coherence systems become increasingly loaded as the number of cores increases). Our machine supports two hyperthreads per core. Increasing the number of threads from 24 to 48 requires placing two threads on at least some of the cores. With this placement, the Cimple threads start interfering and performance decreases. The performance of the baseline computation increases (slowly) between 24 and 48 threads. Nevertheless, the best Cimple computation (on 24 threads) still performs 2.4 times as many operations per second as the best baseline computation (on 48 threads).

4.3.5 Three Key Techniques to Improve MLP and ILP

The example on Figure 4-2 illustrates the three essential techniques for achieving good performance with Cimple on current hardware.

1. The first and most important is to identify independent requests and allow parallelism across them by breaking up execution at `Yield` statements (line 7).
2. The second is to enable longer computation chains between memory requests via explicit software prefetching `Prefetch`.
3. The third is to eliminate unpredictable branches — by replacing a control dependence (if) with an address generation dependence (line 10). Otherwise branch mispredictions would also discard unrelated (correct) subsequent coroutines, since hardware speculative execution is designed to capture the control flow of only one sequential instruction stream.

4.4 Design Overview

The Cimple compiler and runtime library are used via an embedded DSL similar to Halide [152], which separates the basic logic from scheduling hints to guide transformations. Similarly we build an Abstract Syntax Tree (AST) directly from succinct C++ code. Unlike Halide's expression pipelines, which have no control flow, Cimple treats expressions as opaque AST blocks and exposes conventional control flow primitives to enable our transformations. Section 4.5 describes our Cimple syntax in more detail.

Coroutines are simply routines that can be interleaved with other coroutines. Programmers annotate long-latency operations, e.g., memory accesses or unpredictable branches. A `Yield` statement marks the suspension points where another coroutine should run. Dynamic coroutines are emitted as routines that can be resumed at the suspension points, with an automatically generated `struct` tracking all live variables.

Figure 4-2 presents a traditional Binary Search Tree written in Cimple. A coroutine without any Yield statements is simply a routine, e.g., a plain C++ routine can be emitted to handle small-sized data structures, or if Yield directives are disabled. The bottleneck in Figure 4-2 is the expensive pointer dereference on line 8. Yet, prefetching is futile unless we context switch to another coroutine. Figure 4-3 presents a portable [44] unoptimized coroutine for a dynamic coroutine scheduler (Section 4.6.1.3).

4.4.1 Target Language Encapsulation

Cimple emits coroutines that can be included directly in the translation unit of the original routines. Our primary DSL target language is C++. All types and expressions are opaque; statements include opaque raw strings in the syntax of the target language, e.g., native C++.

4.4.2 Cimple Programming Model

A coroutine yields execution to peer coroutines only at **Yield** suspension points.

Expensive memory operations should be tagged with **Load** and **Store** statements (which may yield according to a scheduling policy), or with an explicit **`Prefetch`** directive (see Section 4.5.7). Loads and stores that hit caches can simply use opaque native expressions.

`If`/**`Switch`** or **`While`**/**`DoWhile`** statements should be used primarily to encapsulate mispredicted branches. Most other control-flow statements can use native selection and iteration statements.

A coroutine is invoked using a coroutine scheduler. Regular routines are called from coroutines as usual in statements and expressions. Coroutines are called from inside a coroutine with a **Call**.

4.4.3 Scheduling Hints

Scheduling hints guide transformations and can be added as annotations to the corresponding memory access or control statements. The example in Figure 4-2 showed how a single source file handles four largely orthogonal concerns. First, the program structure is described in Cimple, e.g., `While`. Second, optional inline scheduling directives are specified, e.g., `Yield`. Third, scheduler configuration can be selected via AST node handles in C++, e.g., `auto c`. Finally, all target types and expressions are used unmodified, e.g., `list*`.

4.4.4 Parallel Execution Model

To maintain a simple programming model, and to enable efficient scheduling (Section 4.5.8), Cimple coroutines are interleaved only on the creating thread. IMLP composes well with thread and task parallelism [22, 137]. Instead of running to completion just a single task, a fork-join scheduler can execute multiple coroutines concurrently. The embedded DSL approach allows easy integration with loop and task parallelism extensions, e.g., `#pragma` extensions integrated with OpenMP [137] or Cilk [22, 111].

4.5 Cimple Syntax and Semantics

An original C++ program is easily mapped to the conventional control flow primitives in Cimple. Table 4.1 summarizes our statement syntax and highlights in bold the unconventional directives.

4.5.1 Coroutine Return

A coroutine may suspend and resume its execution at specified **`Yield`** suspension points, typically waiting on address generation, data, or branch resolution. Programmers must ensure that coroutines are reentrant.

`Return` stores a coroutine's result, but does not return to the caller. It instead may resume the next runnable coroutine. **`Result`** defines the coroutine result type, or `void`.

4.5.2 Variable Declarations

The accessible variables at all coroutine suspension points form its context. A target routine's internal variables need to be declared only when their use-def chains cross a yield suspension point. A **`Variable`** can be declared at any point in a block and is presumed to be live until the block end. **Arg**uments to a coroutine and its **Result** are Variables even when internal uses do not cross suspension points. Shared arguments among coroutines using the same scheduler can be marked **SharedArg** to reduce register pressure.

References in C++ allow variables to be accessed directly inside opaque expressions, e.g.:

Arg(int, n).**Variable**(int, x, {n∗2})

For C `Variable` accesses must use a macro: `Var(a)`. We do not analyze variable uses in opaque expressions, but judicious block placements can minimize a variable's scope.

4.5.3 Block Statement

A block statement encapsulates a group of statements and declarations. Convenience macros wrap the verbose Pascal-like `Begin` and `End` AST nodes, e.g., we always open

Table 4.1: Cimple Statements.

Statement	**Section**
`Return`, **`Yield`**	Section 4.5.1
`Arg`, `SharedArg`, `Result`, `Variable`	Section 4.5.2
`If`, `Switch`, `Break`	Section 4.5.5
`While`, `DoWhile`, `Continue`	Section 4.5.6
`Load`, `Store`, `Assign`, **`Prefetch`**	Section 4.5.7
`Call`	Section 4.5.8

a block for the **Then/Else** cases in `If`, **Do** in `While`, and **Body** for the function body block.

4.5.4 Opaque Statements and Expressions

Types and expressions used in Cimple statements are strings passed to the target compiler. Opaque statements are created from string literals, though convenient preprocessor macros or C++11 *raw strings* allow clean multi-line strings and unmodified code wrapping in a `.cimple.cpp` source file, e.g.:

```
<< R""( // Murmur3::fmix32
     h ^= h >> 16; h *= 0x85ebca6b;
     h ^= h >> 13; h *= 0xc2b2ae35;
     h ^= h >> 16;
   )""
```

4.5.5 Selection Statements

`If` and **`Switch`** selection statements can be used for more effective if-conversion to avoid mispredicted branches. For conventional 2-way branch and case selection, If and Switch statements give more control over branch-free if-conversion.

Well-predicted branches do not need to be exposed, and can simply use native `if/switch` in opaque statements. Opaque conditional expressions (`?:`) and standard if-conversion, which converts branches into conditional moves, are effective when only data content is impacted. Traditional predicated execution and conditional moves are less effective when address dependencies need to be hidden, especially for store addresses. Predicated execution also inefficiently duplicates both sides of a branch.

A **`Switch`** must also be used instead of a `switch` when a case has a suspension point, see Section 4.6.1.3.

4.5.6 Iteration Statements

`While` and `DoWhile` iteration statements are exposed to Cimple when there are internal suspension points to enable optimizations. Conventional `Continue` and `Break` respectively skip the rest of the body of an iteration statement, or terminate the body of the innermost iteration or `Switch` statement.

4.5.7 Informed Memory Operations

`Load` and **`Store`** statements mark expensive memory operations that may be processed optimally with one or more internal suspension points. **`Prefetch`** explicitly requires that one or more independent prefetches are issued before yielding. **`Assign`** can mark explicitly other assignments that are expected to be operating on cached data.

4.5.8 Coroutine Calls

A tail-recursive **`Call`** statement resumes execution to the initial state of a coroutine. Regular function calls can be used in all expressions, and are inlined or called as routines as usual. A Return calling a `void` coroutine is also allowed, as in C++, for explicit tail-recursion.

4.6 DSL Compiler and Runtime Library

The DSL allows exploration of multiple coroutine code generation variants and combinations of data layout, code structure, and runtime schedulers. We use two main code generation strategies for handling a *stage* (the code sequence between two `Yield` statements, or function entry/exit): *static* where a stage becomes a `for` loop body, and *dynamic* where a stage forms a `switch` case body. The `Yield` directive marking the boundary of a coroutine stage can select the schedule explicitly.

We first discuss the context of a single coroutine, and storage formats for tracking active and pending coroutines. Then we discuss how these are used in runtime schedulers that create, execute, and retire coroutines.

4.6.1 Coroutine Context

A coroutine's closure includes all private arguments and variables of a coroutine. Shared arguments between instances are stored only once per scheduler and reduce register pressure. Additional variables are optionally stored in the context depending on the code generation choices: a Finite State Machine `state` is used for dynamic scheduling on Yield; a `result` value (of user-defined type) holds the final result; a `condition` – when If yields before making decisions on hard to resolve branches; an `address` (or index) – when Load or Store yields before using a hard to resolve address.

```
struct BST::find__Context_AoS {
  node* n;      // Arg
  KeyType key; // Arg
  int   _state;  // for dynamic Yield
  node* _result; // for Return
  bool  _cond;   // for If
  void* _addr;   // for Load/Store
```

```
 1  bool SuperStep() {
 2    for(int _i = 0; _i < _Width ; _i++) {
 3      KeyType& k = _soa_k[_i];
 4      HashType& hash = _soa_hash[_i];
 5        hash = Murmur3::fmix(k);
 6        hash &= mask;
 7    }
 8    for(int _i = 0; _i < _Width ; _i++) {
 9      KeyType& k = _soa_k[_i];
10      HashType& hash = _soa_hash[_i];
11        prefetch(&ht[hash]);
12    }
```

Figure 4-5: Stages of a Static Coroutine for Figure 4-10.

4.6.1.1 Vectorization-friendly Context Layout

The primary distinctive design choice of Cimple is that we need to run multiple coroutines in parallel, e.g., typically tens. For homogeneous coroutines we choose between Struct-of-Array (SoA), Array-of-Struct (AoS), and Array-of-Struct-of-Array (AoSoA) layouts. Variable accesses are insulated from these changes via convenient C++ references.

4.6.1.2 Static Fused Coroutine

Homogeneous coroutines that are at the same stage of execution can be explicitly unrolled, or simply emitted as a loop. The target compiler has full visibility inside any inlined functions to decide how to spill registers, unroll, unroll-and-jam, or vectorize. An example of SIMD vectorization of a hash function (Figure 4-10 in Section 4.7.4) is shown on Figure 4-5. The hash finalization function called on line 5 has a long dependence chain (shown inlined earlier in Section 4.5.4). C++ references to variables stored in SoA layout, shown on lines 3–4 and 9–10, allow the opaque statements to access all Variables as usual.

Exposing loop vectorization across strands offers an opportunity for performance gains. Since we commonly interleave multiple instances of the same coroutine, we can fuse replicas of the basic blocks of the same stage working on different contexts, or stitch different stages of the same coroutine, or even different coroutines. These are similar to unroll-and-jam, software pipelining, or function stitching [51]. Stage fusion benefits from exposing more vectorization opportunities, reducing scheduling overhead, and/or improving ILP.

Basic block vectorization, e.g., SLP [106], can be improved by better Variable layout when contexts are stored in array of struct (AoS) format.

4.6.1.3 Dynamic Coroutine

Coroutines may be resumed multiple times unlike one-shot continuations. Typical data structure traversals may require coroutines to be suspended and resumed between one and tens of times.

Figure 4-3 presents the basic structure of a `switch` based coroutine that uses "Duff's device" [44] state machine tracking. This method takes advantage of the loose syntax of `switch` statements in ANSI C. Surprisingly to some, `case` labels can be interleaved with other control flow, e.g., `while` loops or `if` statements. Only enclosed `switch` statements can not have a suspension point. Mechanical addition of case labels within the existing control flow is appealing for automatic code generation: we can decorate the original control flow graph with jump labels at coroutine suspension points and add a top level `switch` statement.

This standard C syntax allows good portability across compilers. However, the reliance on `switch` statements and labels precludes several optimization opportunities. Alternatives include relying on computed `goto` (a `gcc` extension), indirect jumps in assembly, or method function pointers as a standard-compliant implementation for C++. The first two are less portable, while the latter results in code duplication when resuming in the middle of a loop.

Short-lived coroutines suffer from branch mispredictions on stage selection. Using a `switch` statement today leaves to compiler optimizations, preferably profile guided, to decide between using a jump table, a branch tree, or a sequence of branches sorted by frequency. Unlike threaded interpreters, which benefit from correlated pairs of bytecodes, [47, 157], the potential correlation benefits from threading coroutines come from burstiness across requests. An additional optimization outside of the traditional single coroutine optimization space is to group across coroutines branches with the same outcome, e.g., executing the same stage.

4.6.2 Coroutine Schedulers

We discuss the salient parameters of coroutine runtime scheduler flavors, and their storage and execution constraints. We target under 100 active coroutines (*Width*) with under 100B state each to stay L1-cache resident. Below is a typical use of a simple coroutine scheduler (for Figure 4-9):

```
template<int Width = 48>
void SkipListIterator_Worker(size_t* answers,
                      node** iter, size_t len) {
   using Next = CoroutineState_SkipList_next_limit;
   SimplestScheduler<Width, Next>(len,
      [&](Next* cs, size_t i) {
            *cs = Next(&answers[i], IterateLimit,
                       iter[i]);
   });
}
```

Static Batch Scheduler Tasks are prepared in batches similar to manual *group prefetching* [31, 123]. Storage is either static AoS, or in SoA format to support vectorization. Scaling to larger batches is less effective if tasks have variable completion time, e.g., on a binary search tree. Idle slots in the scheduler queue result in low effective MLP.

Dynamic Refill Scheduler Tasks are added one by one, and refilled as soon as a task completes, similar to the manual approach in AMAC [95]. Storage is in static or dynamic-width AoS. Further optimizations are needed to reduce branch mispredictions to improve effective MLP.

Hybrid Vectorized Dynamic Scheduler Hybrid across stages, where the first stages of a computation can use a static scheduler, but following stages use a dynamic scheduler while accessing the SoA layout.

4.6.2.1 Common Scheduler Interface

Runtime or user-provided schedulers implement common APIs for initialization, invoking coroutines, and draining results. A homogeneous scheduler runs identical coroutines with the same shared arguments. New tasks can either be pulled via a scheduler callback or pushed when available. A pull task with long latency operations or branch mispredictions, may become itself a bottleneck. Routines with non-`void` results can be drained either in-order or out-of-order. Interfaces are provided to drain either all previous tasks or until a particular task produces its result.

We show in Section A.1 the simplest scheduler and a typical scheduler use. Simple enqueue/dequeue initiated by an outside driver, and more flexible callback functors to `push/pull` tasks are shown in Section A.2.

4.7 Applications

We study Cimple's expressiveness and performance on core database data structures and algorithms used. Simple near-textbook implementations in Cimple ensure correctness, while scheduling directives are used to fine-tune performance. We compare Cimple C++, against naïve native C++ and optimized baselines from recent research.

We start with a classic binary search, which is often the most efficient solution for a read-only dictionary. For a mutable index, in addition to the binary search tree we have shown in Section 4.3, here we show search in a skip list. Since both of these data structures support efficient range queries in addition to lookup, these are the default indices respectively of the VoltDB and RocksDB commercial databases. Finally, we show a hash table as used for database join queries.

4.7.1 Array Binary Search

```
1   Arg(ResultIndex*, result).
2   Arg(KeyType, k).
3   Arg(Index, l).
4   Arg(Index, r).
5   Body().
6   While( l != r ).Do(
7     Stmts(R""( {
8       int mid = (l+r)/2;
9       bool less = (a[mid] < k);
10      l = less ? (mid+1) : l;
11      r = less ? r : mid;
12       } )"").
13     Prefetch(&a[(l+r)/2]).Yield()
14  ).
15  Stmt( *result = l; );
```

Figure 4-6: Cimple binary search.

Figure 4-6 shows our Cimple implementation. Current `clang` compilers use a conditional move for the ternary operators on lines 10–11. However, it is not possible to guarantee that compilers will not revert to using a branch, especially when compiling without Profile Guided Optimization. For finer control, programmers use provided helper functions or write inline assembly with raw statements.

Perversely, a naïve baseline performs better with a mispredicted branch as observed in [87], since speculative execution is correct 50% of the time. When speculative loads have no address dependencies, hardware aggressively prefetches useless cache lines, as we show in Section 4.8.3.

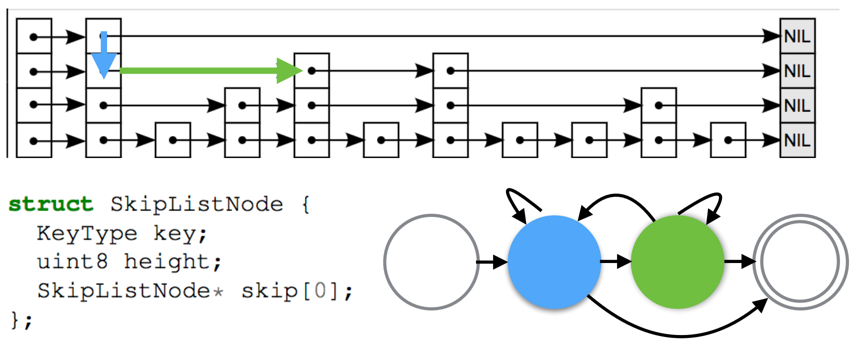
The Cimple version works on multiple independent binary searches over the same array. All of our prefetches or memory loads are useful.

4.7.2 Binary Search Tree

See the binary search tree example in Section 4.3, with the Cimple version in Figure 4-2.

4.7.3 Skip List

Lookup Our skip list baseline is Facebook's `folly` template library implementation of ConcurrentSkipList [60]. Figure 4-7 shows the skip list data structure layout, and the state machine generated for the code in Figure 4-8; we also illustrate how a lookup follows *down* and then *right*. Note that in the *down* direction (line 5) an array of pointers is explored, therefore speculative execution in the baseline is not blocked by address dependencies; the *right* direction (line 13) cannot be speculated.



```
struct SkipListNode {
  KeyType key;
  uint8 height;
  SkipListNode* skip[0];
};
```

Figure 4-7: SkipList traversal, data layout, and coroutine state machine

```
VariableInit(SkipListNode*, n, {}).
VariableInit(uint8, ht, {pred->height}).
While(true).Do(
    While(ht > 0).Do( // down
      Stmt( n = pred->skip[ht - 1]; ).
      Prefetch(n).Yield().
      If(!less(k, n->key)).Then(Break()).
      Stmt( --ht; )
    ).
    If (ht == 0).Then( Return( nullptr )).
    Stmt( --ht; ).
    While (greater(k, n->key)).Do(
      Stmt( pred = n; n = n->skip[ht]; ).
      Prefetch(n).Yield().
    ).
    If(!less(k, n->key)).Then(
       Return( n )));
```

Figure 4-8: Cimple Skip List lookup.

Range Query Range queries are the main reason ordered dictionaries are used as default indices. Skip list iteration requires constant, but still inefficient, pointer chasing (Figure 4-9). Request level parallelism in range queries is handled similarly to lookup by interleaving multiple independent queries for both finding the first node and for iterating and aggregating over successive nodes.

4.7.4 Hash tables

We compare the performance of an open-address hash table for the special case of database hash-join. An ephemeral hash table optimized for hash-join [12] only needs to support bulk insertion followed by a phase of lookups. The identity hash function can not be used in real workloads, both for performance due to non-uniform skew,

```
    While( limit-- ).Do(
        Prefetch(n).Yield().
        Stmt( n = n->skip[0]; )
    ).
    Prefetch(n).Yield().
    Return( n->key );
```

Figure 4-9: Cimple Skip List Iteration.

```
Result(KeyValue*).
Arg(KeyType, k).
Variable(HashType, hash).
Body().
  Stmt ( hash = Murmur3::fmix(k); ).
  Stmt ( hash &= this->size_1; ).Yield().
  Prefetch( &ht[hash] ).Yield()
  << R""(
  while (ht[hash].key != k &&
         ht[hash].key != 0) {
       hash++;
       if (hash == size) hash = 0;
  } )"" <<
  Return( &ht[hash] );
```

Figure 4-10: Cimple Hash Table lookup (linear probing).

and for security due to Denial-of-Service complexity attacks [38].

Figure 4-10 shows our classic linear probing hash table, similar to the implementation suggested in Menon et al. [123] — linear probing at 50% load factor, and Murmur3's finalization as a hash, masked to the table size. Menon et al. report 1.2× gains from LLVM SIMD vectorization and group prefetching [31], on a well-engineered hash table for state-of-the-art TPC-H performance.

A requested static schedule for all three stages (delineated by `Yield` on lines 6 and 7 of Figure 4-10) generates three independent static stages (shown in [92]). Using the SoA layout enables compiler loop vectorization to use AVX2 or AVX512 to calculate multiple hash functions simultaneously.

Variants Menon et al. [123] analyze the inefficient baseline used in AMAC [95], i.e., identity hash, chaining at 400% load factor, and using a linked list for handling duplicates.

For a chained hash table, which traverses a linked list, we can also produce a hybrid schedule. The first two steps use a static schedule (with SoA storage), while the third stage can use a dynamic scheduler to handle a variable number of cache lines traversed.

4.8 Evaluation

We report and analyze our performance gains using Cimple used as a template library generator. Our peak system throughput increases from 1.3× on HashTable to 2.5× on SkipList iteration; Cimple speedups of the time to complete a batch of queries on a single thread range from 1.2× on HashTable to 6.4× on BinaryTree (Figure 4-1).

4.8.1 Configuration

System Configuration We used a dual socket Haswell system [45] with 24 cores at 2.9GHz clock frequency, or 3.3GHz for a single core. Each socket has 4 memory channels populated with dual rank, 16-bank DDR4-2133 memory [160]. The DRAM memory level parallelism on each socket therefore allows 128 open banks for random access. The test applications were compiled with `gcc` 4.8 and executed on Linux 4.4 using huge pages.

Cimple Configuration We implemented the Cimple DSL in a combination of 2,500 lines of C++14 and 300 lines of C preprocessor macros. Cimple to C++ code was built with Apple clang 9.0. The template library of runtime schedulers adds less than 500 lines of C++11 code. Cimple suspend/resume of the minimum coroutine step (on **SLi** — 21 extra instructions) adds 4ns. We use 48 entry scheduler width — optimal for all DRAM-resident benchmarks.

4.8.2 Performance Gains

Binary Search (BS) The multithreaded version has all threads searching from the same shared array of 1 billion 64-bit keys. Branch-free execution is important for good performance as discussed in Section 4.3.5. When a branch is used on lines 10 and 11 of Figure 4-6, we see only a 3× performance gain. While CMOV in the baseline leads to a 0.7× slowdown, Cimple+CMOV reaches 4.5× over the best baseline.

Binary Tree lookup (BT) Each thread works on a private tree to avoid synchronization, as used in the context of partitioned single-threaded data stores, such as VoltDB or Redis. We use 1GB indexes scaled by the number of threads, i.e., 48GB for the full system. We achieve 2.4× higher peak throughput and 6.4× speedup for a single thread of execution. Our ability to boost a single thread performance much higher above average, will support handling of skewed or bursty workloads, which can otherwise cause significant degradation for partitioned stores [177].

SkipList lookup (SL) Concurrent SkipLists are much easier to scale and implement [60] compared to a binary tree, therefore practical applications use multiple threads looking up items in a shared SkipList.

All items are found after a phase of insertions with no deletions or other sources of memory fragmentation. We achieve 2.7× single thread speedup and 1.8× multithreaded throughput. Note that for SkipList lookup the "down" direction follows an array of pointers, therefore the baseline benefits from speculative execution prefetching nodes.

SkipList Iterator (SLi) We evaluated range queries on the same shared skip list index as above. For 1,000 node limit iterations, similar to long range queries in [7, 175] our total throughput gain is 2.5× and single thread speedup is 4.1×.

Hash Table lookup (HT) We evaluate hash table join performance on a table with 64-bit integer keys and values. We use a 16 GB hash table shared among all threads for an effective load factor of 48%. We replicate similar single thread speedups [123] of 1.2× when either no results or all results are materialized. Since there are few instructions needed to compare and store integer keys and values, hardware is already very effective at keeping a high number of outstanding requests. However, both the hash table load factor and the percentage of successful lookups impact branch predictability, and thus ILP and MLP for the baseline. For 50% materialized results, our speedup is 1.3×. When using 48 threads with 100% hits, we get a 1.3× higher throughput of 650 M operations/s.

We also compared to other traditional but inefficient on modern cores variants, e.g., if division by a prime number is used [31] the corresponding Cimple variant is 2× faster. When there are serializing instructions between lookups our speedup is 4×.

Table 4.2: Memory Level Parallelism (MLP), Instruction Level Parallelism (ILP), and Instructions Per Cycle (IPC).

Benchmark	**MLP**		**ILP**		**IPC**	
	Baseline	Cimple	Baseline	Cimple	Baseline	Cimple
BS	7.5	8.5	1.6	2.3	0.13	1.10
BT	1.2	4.3	1.6	2.3	0.10	0.70
SL	2	5	1.8	2.4	0.07	0.60
SLi	1	5	1.3	2.0	0.01	0.22
HT	4.9	6.4	1.9	2.4	0.37	0.40

4.8.3 Performance Analysis

We analyze hardware performance counters to calculate the total ILP and MLP, and to understand where our transformations increase the useful ILP and MLP.

4.8.3.1 ILP Improvements

Table 4.2 shows our improvements in ILP and IPC by increasing the useful work per cycle and reducing the total number of cycles. The ILP metric measures the average μinstructions executed when not stalled (max 4). Cimple may have either higher or lower instruction count: e.g., a pointer dereference in **SLi** is a single instruction, while with a dynamic scheduler that instruction is replaced by context switches with attendant register spills and restores. For a static scheduler, vector instructions reduce additional instructions in **HT**. The remaining stall cycles show that there is sufficient headroom for more expensive computations per load.

4.8.3.2 MLP Improvements

Improving MLP lowers the stall penalty per miss, since up to 10 outstanding L1 cache misses per core can be overlapped.

In Table 4.2 we show that measured MLP improved by 1.3–6$\times$ with Cimple. Measured as the average outstanding L2 misses, this metric includes speculative and prefetch requests. Therefore the baseline MLP may be inflated due to speculative execution which does not always translate to performance. Cimple avoids most wasteful prefetching and speculation, therefore end-to-end performance gains may be larger than MLP gains.

In BinarySearch the baseline has high measured MLP due to speculation and prefetching, however, most of it is not contributing to effective MLP. For BinaryTree the addresses of the children cannot be predicted, therefore the baseline has low MLP. For SkipList lookup the down direction is an array of pointers therefore speculative execution may prefetch correctly needed values, thus while the measured MLP is 2, the effective MLP is 1.5. SkipList iteration is following pointers and therefore has MLP of 1. For HashTable at low load and 100% hit rate, speculative execution is always correct, thus the baseline has high effective MLP.

There is also sufficient headroom in memory bandwidth and queue depth for sequential input and output streams, e.g., for copying larger payload values.

4.9 Related Work

We survey related work in hardware multithreading – in particular SMT, coroutines and tasks, and software optimizations.

4.9.1 Hardware Multithreading

Hardware context switching was explored in supercomputers of the lineage of Denelcor HEP [173] and Tera MTA [5], e.g., Tera MTA supported 128 instruction streams that were sufficient to hide the latency of 70 cycles of DRAM latency without using caches. Yet locality is present in real workloads, and caches should be used to capture different tiers of frequently used data. Larrabee [164] threading and vectorization model allowed SIMD rebundling to maintain task efficiency. Current GPUs offer large number of hardware threads, yet relying solely on thread-level parallelism is insufficient [187], and taking advantage of ILP and MLP is critical for GPU assembly-optimized libraries [104, 130].

4.9.1.1 SMT

SMT can improve effective MLP for pointer-chasing code, as long as the shared instruction window does not become the limiting factor. Most CPU vendors currently have only 2-way SMT, for example, Intel since Pentium4 [101] and AMD (recently in EPYC). In contrast, Cimple's lightweight schedulers effectively maintain 48 coroutine execution contexts.

Traditional multithreading requires programmers to map request level parallelism to thread level parallelism, use OS scheduling and heavy-weight task creation, and duplicate HW register state for identical values. Cimple is scheduled at user-level, and shares common registers to improve physical register occupancy. Single-threaded partitioned databases can eschew the complexities of loosely synchronized threads.

4.9.1.2 SMT Security and Cimple

SMT has been recently vilified since it allows more side-channel attacks [68]. The quintessential to SMT sharing of execution units and execution ports allows fine granularity side-channels, in addition to any multithreading shared caches/TLBs. Yet, disabling SMT is costly. Cimple is a secure alternative, where many independent tasks of the same protection domain run as short-lived co-routines on the same core.

4.9.2 Informing Loads and Speculation

Out-of-order CPUs can track the concurrent execution of tens of co-running coroutines per core, but provide no efficient notification of operation completion. Informing loads [64] were proposed as a change to the memory abstraction to allow hardware to set a flag on a cache miss and trap to a software cache-miss handler, similar to a TLB-miss handler. Proposals for hardware support for overlapping instructions from different phases of execution with compiler transformations have shown modest performance gains [141, 153, 168, 181, 185].

4.9.3 Coroutines and Tasks

Coroutines have been a low-level assembly technique since the 1950s, originally used in limited-memory stackless environments [132, 133]. Lowering continuation overhead has been approached by specialized allocation [62] and partial [149] or one-shot [26] continuations.

The C++20 standard is also slated to support coroutines with the keywords `co_yield`, `co_await`, and `co_return`. The original proposals [55, 136] motivate the goal to make asynchronous I/O maintainable. The runtime support for completion tracking is acceptable at millisecond scale for handling network and disk, but is too heavyweight for tolerating tens to hundreds of nanoseconds memory delays targeted by IMLP tasks. The concise syntax and automatic state capture are attractive and the underlying mechanisms in LLVM and Clang can be used to add Cimple as non-standard C++ extensions to delight library writers. Library users can use the generated libraries with mainstream compilers.

Cilk [22, 49] introduced an easy programming model for fork-join task parallelism, divide-and-conquer recursive task creation and work-stealing scheduler. More recently the CilkPlus [111] extensions to C/C++ were added to `icc` and `gcc`. C++20 proposals for `task_block` [56] incorporate task parallelism like Cilk, albeit using a less succinct syntax.

Our rough guide to these overlapping programming models would be to use C++20 tasks for compute bottlenecks, C++20 coroutines for I/O, and Cimple coroutines for memory bottlenecks.

4.9.4 Software Optimizations

Software prefetching by requesting data at a fixed distance ahead of the current execution is complex even for simple loop nests and reference streams without indirection [127]; and more recently surveyed in [109]. Augmented data structures help deeper pointer chasing [33, 98].

Optimized-index data-structures for in-memory databases [32, 85, 103, 107, 110, 112, 154, 165] try to reduce the depth of indirect memory references and use high fan-out and extra contiguous accesses while performing one-at-a-time requests. Techniques that uncover spatial or temporal locality by reordering billions of memory requests [19, 94] are not applicable to index queries which often touch only a few rows.

Group prefetching (static scheduling) and prefetching with software pipelining techniques were introduced in [31] where a group of hash table lookups is processed as a vector; similarly used for an unordered key-value store [113]. AMAC [95] is an extension to group prefetching to immediately refill completed tasks (dynamic scheduling) in order to handle better variable work or variable access time per-item on skewed inputs. In a well-engineered baseline in the state-of-the-art database engine Peloton [123], however, AMAC was deemed ineffective and only group prefetching on hash tables was beneficial and maintainable.

Using C++20 coroutines for easier programmability of AMAC-style dynamic scheduling was evaluated in concurrent work in SAP HANA [147], and more recently by [80]. While easier to use and more maintainable than manual interleaving methods [31, 95], C++20 coroutine backends preclude static or hybrid scheduling. Dependence on I/O-oriented compiler coroutine implementations adds high overhead compared to manual AMAC [95]; C++20 coroutines [80, 147] are even outperformed by static scheduling [31] in some cases in which dynamic scheduling with AMAC is otherwise better than static scheduling. Using a black-box compiler is also not practical for JIT query engines used in modern databases for these critical inner loops. For less critical code-paths implemented in C++, their promising results are a step in the right direction. Cimple's back-end seamlessly enables static and hybrid scheduling, with efficient dynamic scheduling coroutines optimized for caches and out-of-order processors.

4.10 Future Work

We highlight several directions for future work for the Cimple DSL, Cimple-optimized data structures, C++20 integration, and hardware support.

4.10.1 Language Support for Internal Parallelism

Internal parallelism is also available in many operations such as range queries in ordered database indices, priority queue operations, etc. Using manual techniques, we have demonstrated similar 4-5x performance wins, and a practical advantage of internal parallelism is that higher level APIs do not need to change, e.g., we process one query at a time. Such parallelism invisible above data-structure APIs improves modularity.

Spawning tasks dynamically, with `Spawn` primitive (much akin to `spawn` in Cilk [49]), allow natural recursive patterns to be expressed with deferred tasks. The same Cimple code can be used to emit variants for either request parallelism or internal data-structure parallelism. The optimal choice depends on the interfaces for issuing requests, the expected result lengths, and the overall system throughput and latency objectives.

Latency goals favor using the MLP resources to complete one query at a time. Throughput goals favor using the MLP resources most efficiently over multiple queries, then `Spawn` should not create new active tasks.

Creation of concurrently executed tasks with fork-join parallelism is similar to the model of Cilk or tasks OpenMP. The execution model is vastly simplified by cooperative scheduling of coroutines on a single thread. User tasks do not need synchronization among peers, and do not require atomic instructions for read-modify-write operations.

4.10.2 Language Support for Synchronization Hooks

Thread synchronization today has to be provided by the user-provided schedulers. Notably, the existing data structure synchronization may also not scale when 6x higher throughput per thread can be achieved with Cimple. DSL support for several common design patterns may highlight best practices based on the expected contention, operations mix, and burstiness.

Modification operations (update/insert/delete) may be accelerated with Cimple simply by opportunistically issuing a read-only lookup operation in order to warm up caches. The easiest case is when read-only operations can be interleaved without using any synchronization - possible when there is no deletion, or if the underlying data structure uses epoch-based memory deallocation (e.g., as used in typical concurrent skip-list implementations). Sorted lock-sets to avoid dead-locks must be used in place of simple mutex protection, striped locks, or shared locks (using the address of the innermost synchronization object). Try-locks can re-queue any operations in a batch that would block by using a multi-queue scheduler to avoid head-of-line blocking.

4.10.3 Novel Data structures Adapting to MLP

While we have shown Cimple uses for unmodified data structures, we believe new data structures for database indices should be re-designed to allow taking full advantage of memory level parallelism. Furthermore, designs that allow the same in-memory data structures to be accessed with routines optimized for different effective MLP will be able to adapt to task latency and throughput requirements. Detecting available memory bandwidth and effective memory latency, e.g., on bursty workload mixes, would allow optimal runtime selection.

With Cimple, the most critical metric is the number of cache lines needed to complete a task. Prior data-structure optimizations, which were focused on a request-at-a-time performance are often counterproductive to maximizing overall throughput. For example, use of software prefetches on both sides of a binary search, multi-cache-line B-tree designs that benefit from hardware prefetches, or skip-lists which benefit from speculative execution. These techniques are most useful only when optimizing for idle latency of a single request.

Maximizing single-threaded throughput may be beneficial for skewed workloads, for example in partitioned databases where one partition gets a much larger share of the work. While total system memory bandwidth in these conditions may be ample, the memory bandwidth for random accesses available to a single core is limited by queue depth limits. Hardware prefetchers at lower level caches can improve the effective sequential bandwidth available to a single core, e.g. for input and output.

When maximizing overall throughput on current systems which are memory bandwidth bound, all cache lines brought from DRAM must be useful. Any potentially wasteful hardware or software prefetches, and wrong-path speculation should be avoided.

4.10.4 Integration with C++20 co-routines

Future easier syntax of the current DSL facilities can be enabled with extensions to C++20. Mainstream users, however, want standards, while language standards want users. The intensified interest in these techniques [80, 148] and reported challenges and suboptimal performance from using the existing C++ coroutine facilities will hopefully result in better interfaces and implementations. We expect to be able to offer a similar C++ front-end, once coroutines are mature in Clang, with additional Cimple AST annotations as C++ double-bracketed `[[attributes]]`.

4.10.5 Hardware Support

Today our primary target is unmodified hardware, however, there are opportunities for future hardware and software co-design.

Our easier programming models would benefit from hardware extensions to take advantage of both external and internal parallelism. For example, hardware designs with 8-way SMT require a large increase of architectural state, while Cimple can use a large physical register file more efficiently and can take advantage of hardware-assisted context switch or `Spawn`.

4.11 Summary

Cimple is fast, maintainable, and portable. We offer an optimization methodology for experts, and a tool usable by end-users today.

We introduced the IMLP programming model and methodology for uncovering ILP and MLP in pointer-rich and branch-heavy data structures and algorithms. Our Cimple DSL and its AST transformations for C/C++ in Cimple allow quick exploration of high-performance execution schedules. Cimple coroutine annotations mark hot-spots with deep pointer dereferences or long dependence chains. Cimple achieves up to 6.4× speedup.

Our compiler-independent DSL allows low-level programmers to generate high-performance libraries that can be used by enterprise developers using standard tool-chains. We believe ours and others' promising early results are the first steps towards efficient future Coroutines for Instruction and Memory Parallel Language Extensions to C++.

Part II

Speculative Attacks and Defenses

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Security Implications of Indirect Memory References

5.1 Overview

Indirect memory references, like `a[b[i]]`, bring forth not only performance challenges, but also formidable security challenges via side-channel attacks. On current speculative execution microarchitectures, these innocuous looking ‘gadgets’ enable attackers to extract secrets via high-bandwidth cache-timing attacks. In addition to being the easiest to use in attacks, indirect memory references are the hardest to protect while maintaining good performance.

Allowing speculative execution past conditional branches is an important performance optimization employed by speculative out-of-order processors — essentially every modern high-performance CPU. Ideally, speculative machines should exhibit the same isolation guarantees for confidentiality and integrity as in-order machines. Therefore, neither local nor remote attackers should be able to extract secrets or to modify critical data that belong to victims. Early in 2018, however, almost three decades after the introduction of speculative execution to microprocessors, multiple independent researchers have disclosed ways for attackers to leak sensitive data across trust boundaries by exploiting speculative execution [63, 96, 116].

In this chapter, we first illustrate the basic Spectre attack methods and the security challenges they introduced. Then we extend the understanding of speculative attacks with *speculative buffer overflows*[91], a class of attacks that break type and memory safety during speculative execution (with one simple instance shown in Section 5.2.2).

5.2 Speculative Side-Channel Vulnerabilities

Using speculative execution, an attacker is able to influence code in the victim’s domain to access and transmit a chosen secret [63, 96]. Spectre attacks leverage speculative loads which circumvent access checks to read memory-resident secrets, transmitting them to an attacker via indirect memory references and receiving them using cache timing or other covert communication channels. We refer to Spectre

```
1     if (x < lenb)
2       return a[b[x] * 512];
```

Figure 5-1: Spectre1.0: Bounds Check Bypass (***on Loads***). Speculative secret access via attacker-controlled `x`, and indirect-load transmission gadget using attacker-controlled cache state for array `a`.

```
3     if (y < lenc)
4       c[y] = z;
```

Figure 5-2: Spectre1.1: Bounds Check Bypass (***on Stores***). Arbitrary speculative write with attacker-controlled `y`, and attacker-controlled or known value `z`.

variant 1 [63, 96] as Spectre1.0, (CVE-2017-5753, bounds check bypass on loads).

Shortly after Spectre's announcement, we discovered its dual variant Spectre1.1 (CVE-2018-3693, bounds check bypass on stores), and responsibly disclosed to industry responders [91]. With Spectre1.1, an attacker leverages speculative stores to create *speculative buffer overflows*. Much like classic buffer overflows, speculative out-of-bounds stores can modify data and code pointers. Data-value attacks can bypass some Spectre1.0 mitigations, either directly or by redirecting control flow. Control-flow attacks enable arbitrary speculative code execution, which can bypass fence instructions and all other software mitigations for previous speculative-execution attacks. It is easy to construct return-oriented-programming (ROP) gadgets that can be used to build speculative attack payloads. Our software and hardware mitigation recommendations for current systems can be found in [91].

5.2.1 Spectre1.0: Bounds Check Bypass on Loads

The original Spectre1.0 attack, as well as currently deployed automatic mitigations, target sequences like Figure 5-1. Since a speculative out-of-order processor may ignore the bounds check on line 1, an attacker-controlled value `x` is not constrained by `lenb`, the length of array `b`. A *secret value* addressable as `b[x]` can therefore be used to influence the index of a dependent load from array `a` into the cache.

The transmission channel in current proof-of-concept attacks uses microarchitectural cache state — a channel available to speculatively-executed instructions. Cache tag state was a previously-known channel for transmitting information in more limited scenarios — side channels (during execution of cryptographic software operating on a secret [28]), and covert channels (where a cooperating transmitter is used).

In the simplest attack scenario, the attacker also has access to the physical memory backing array `a`. (Page deduplication [188] across VMs allows one simple version of this attack, other exfiltration variants are illustrated in Section 6.4.2.) As a first step, the attacker flushes `a` from the cache before executing the victim code [197]. The indirect memory reference `a[b[x]*512]` is a typical ex-filtration gadget that forms a cache side-channel transmitter. The speculative branch on line 1 will be resolved only when `lenb` is available, and the attacker may be able to ensure that the memory access is slow. Furthermore, even when `lenb` is in the L1 cache and the branch is quickly resolved, within a few cycles an attacker can initiate an indirect memory reference. Since speculative memory requests that have missed in the L1 cache are not canceled, initiating a request and placing it in an MSHR within the speculative execution window is sufficient for a load to be cached. An attacker may simply repeat multiple re-executions in order to use values cached after previous attempts. The first attempt ensures the secret value `secret=b[x]` is cached, and subsequent attempts will refer to that value to compute the indirect address and reference `a[secret*512]`.

The speculative attack leaves a footprint in the cache, and the attacker measures each cache line of `a` to determine which one has the lowest access time — inferring the secret value from the address of the fastest line. Generic mitigations against Spectre1.0 and its variants, such as restricting shared memory or reducing timer precision, have been limited to this particular ex-filtration method.

5.2.2 Spectre1.1: Bounds Check Bypass on Stores

Code vulnerable to Spectre1.1 is shown in Figure 5-2. During speculative execution, the processor may ignore the bounds check on line 3. This provides an attacker with the full power of an arbitrary write. While this is only a *speculative* write, which leaves no architecturally-visible effects, it can still lead to information disclosure via side channels.

As a simple proof-of-concept attack, suppose `c[y]` points to the return address on the stack, and `z` contains the address of line 2 in Figure 5-1. During speculative execution of a function return, execution will be resteered away to the transmission gadget, as previously described. Note that even if a fence instruction (e.g., `lfence`) is added between lines 1 and 2 to mitigate against Spectre1.0, an attacker can simply adjust `z` to "jump over the fence". Return-oriented-programming (ROP) techniques can also be used to build alternative attack payloads.

In a speculative data attack, an attacker can also (temporarily) overwrite data used by a subsequent Spectre1.0 gadget. Performant mitigations may use data-dependent truncation (e.g., `x &= (lenb-1)`) rather than fences. However, an attacker may regain arbitrary read access by overwriting either the base of array `b` (line 2), or its length, `lenb` (line 1).

The ability to perform arbitrary speculative writes presents significant new risks, including arbitrary speculative execution. Unfortunately, this enables both local and remote attacks, even when Spectre1.0 gadgets are not present. It also allows attackers to bypass recommended software mitigations for previous speculative-execution attacks.

We have proposed several general strategies for blocking speculative attacks, including at attack entry (e.g., Sloth for Spectre1.1 [91]), and at attack exit (e.g., for caches with DAWG [89] which will evaluate in Chapter 6).

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 6

DAWG: Secure Cache Partitioning

Any problem in computer science can be solved with another level of indirection.

David Wheeler

In Lampson, *Principles for Computer System Design* [105]

6.1 Overview

Software side channel attacks have become a serious concern with the introduction of the Spectre family of speculative attacks [63, 91, 96, 116]. Prior (non-speculative) software side channel attacks primarily targeted cryptographic routines that leak secret values being operated on. Speculative side channel attacks give much more general abilities to unauthorized attackers to leak any memory contents. Most speculative and non-speculative attacks that have been demonstrated use secret-dependent indirect memory references to leak secrets via cache tag state. In this chapter, we propose minimal modifications to hardware to defend against a broad class of attacks, both speculative and non-speculative, with the goal of eliminating the entire attack surface associated with ex-filtration via the cache state covert channel.

We introduce DAWG (Dynamically Allocated Way Guard) — a generic mechanism for secure way partitioning of set associative structures including memory caches. DAWG endows a set associative structure with a notion of protection domains to provide strong isolation. When applied to a cache, unlike existing quality of service mechanisms such as Intel's Cache Allocation Technology (CAT) [59], DAWG fully isolates hits, misses, and metadata updates across protection domains. We describe how DAWG can be implemented on a processor with minimal modifications to modern operating systems.

DAWG's main contributions highlighted in this chapter include:

1. We design a cache way partitioning scheme, DAWG, with strong isolation properties that blocks old and new attacks based on the cache state exfiltration channel (cf. Section 6.5).

2. DAWG allows efficient system calls and communication via shared caches, and preserves the semantics of copy-on-write resource management. We introduce a level of indirection to memory references in cores (cf. Section 6.5.3) with minimal changes to modern operating systems (cf. Section 6.6.2.1).
3. We illustrate the limitations of cache partitioning for isolation by discussing a hypothetical leak framed by our attack schema (cf. Figure 6-1) that circumvents a partitioned cache. For completeness, we briefly describe a defense against this type of attack (cf. Figure 6-5 in Section 6.7.2).
4. We show the minimal additional hardware and software support on top of the commercially available QoS mechanisms in CAT [59].
5. We evaluate the performance impact of DAWG in comparison to CAT [59] and non-partitioned caches with a variety of workloads, including cache-sensitive graph workloads. We detail the overhead of DAWG's protection domains, which limit data sharing in the system in Section 6.8.

More detailed analysis and implementation details of DAWG are available in [89, 90].

The rest of this chapter is organized as follows. We motivate the need for DAWG by analyzing attacks in Section 6.2, and outline the defense approach in Section 6.3. We provide background and discuss related work in Section 6.4. The hardware modifications implied by DAWG are presented in Section 6.5, and software support is detailed in Section 6.6. Discussion of security analysis and limitations are the subjects of Section 6.7. Finally, we evaluate the performance impact of DAWG on the cache subsystem in Section 6.8, and Section 6.10 concludes.

6.2 Motivation

A *covert communication channel* transfers information between processes that should not be allowed to communicate by existing protection mechanisms. In a well-designed system an attacker cannot architecturally observe the *secret*, as the secret should be confined to a *protection domain* that prevents other programs from observing it. However, vulnerabilities may exist when an attacker can observe side effects of execution.

A generalized attack schema is pictorially illustrated in Figure 6-1. For example, when a side channel is used to convey a "secret" to an attacker, an attack would include code inside the victim's protection domain for accessing the secret and a *transmitter* for conveying the secret to the attacker. Together they form a *data tap* that will modulate the channel based on the secret. A *receiver* controlled by the attacker, and outside the victim's protection domain, will listen for a signal on the channel and decode it to determine the secret.

In traditional non-speculative side-channel attacks against cryptographic routines [24, 28, 139], a data tap may pre-exist in victim's code. Recently, however, multiple security researchers (e.g., [63, 96, 116]) have found ways for an attacker to create a *new* data tap in the victim. Here, an attacker is able to *create* a data tap in the victim's domain and/or influences the data tap to access and transmit a chosen secret.

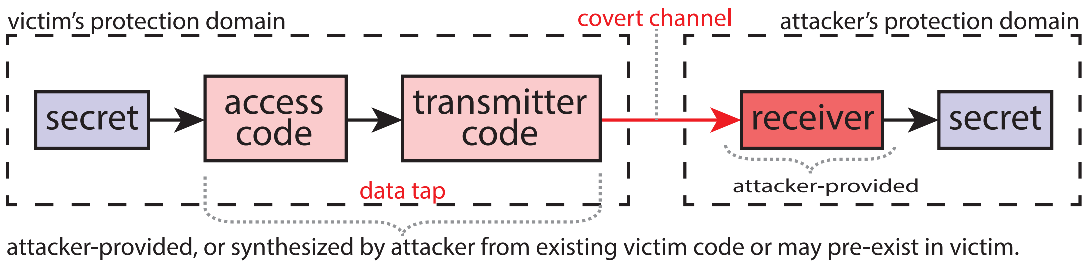

victim's protection domain
covert channel
attacker's protection domain
secret
access code
transmitter code
receiver
secret
attacker-provided
data tap
attacker-provided, or synthesized by attacker from existing victim code or may pre-exist in victim.

Figure 6-1: Attack Schema: an adversary 1) accesses a victim's secret, 2) transmits it via a covert channel, and 3) receives it in their own protection domain.

Spectre and Meltdown have exploited the fact that code executing *speculatively* has full access to any secret. In Meltdown, an attacker speculatively executing data tap code that illegally accesses the secret and causes a transmission via micro-architectural side effects before an exception is raised [116]. Fortunately, a viable OS workaround [37] has been deployed to address Meltdown comprehensively.

The Spectre variant 1 family remains most concerning, where software vulnerable instances of SPECTRE1.0 or SPECTRE1.1 may be reachable in more vulnerable programs or might get re-introduced in the future. While speculative execution is broadly defined, we focus on control flow speculation in this paper. Modern processors execute instructions *out of order*, allowing downstream instructions to execute prior to upstream instructions as long as dependencies are preserved. Most instructions on modern out-of-order processors are also *speculative*, i.e., they create checkpoints and execute along a predicted path while one or more prior conditional branches are pending resolution. When a processor does not know the future instruction stream because a conditional branch instruction is reached whose direction depends on upstream instructions that have not executed yet, the processor predicts the path that the program will follow, and speculates along that path after making a checkpoint. A prediction resolved to be correct discards a checkpoint state, while an incorrect one forces the processor to roll back to the checkpoint and resume along the correct path. Incorrectly predicted instructions are executed, for a time, but do not modify architectural state. However, micro-architectural state such as cache tag state *is* modified as a result of (incorrect) speculative execution causing a channel to be modulated, which may allow secrets to leak.

By exploiting mis-speculated execution, an attacker can exercise code paths that are normally not reachable, circumventing software invariants. Many speculative attack vulnerabilities have been reported [63, 91, 96]. Fortunately, all but Spectre variant 1 have complete and effective resolutions. In the SPECTRE1.x family the attacker coerces branch predictor state to encourage mis-speculation along an attacker-selected code path, which implements a data tap in the victim's domain. An indirect memory reference, as shown earlier in Figure 5-1 allows an unchecked access to any addressable memory (access code: `b[i]`), and the cache transmitter code is then the data-dependent access to `a[b[i]]`. Using SPECTRE1.1 (Figure 5-2) such sequence can be found anywhere in the mapped code pages, or even constructed as a return-oriented programming (ROP) program.

6.3 DAWG (Dynamically Allocated Way Guard)

DAWG is our generic mechanism for secure way partitioning of set associative structures including caches. DAWG endows a set associative structure with a notion of *protection domains* to provide strong isolation. We target minimal modifications to hardware that defend against a broad class of side channel attacks, for both speculative and traditional non-speculative attacks, with the goal of eliminating the entire attack surface associated with exfiltration via changing cache state.

To prevent exfiltration, we require strong isolation between protection domains, which prevents any transmitter/receiver pair from sharing the same channel. Cache partitioning is an appealing mechanism to achieve isolation. Unfortunately, set (e.g., page coloring [86, 178]) and way (e.g., Intel's Cache Allocation Technology (CAT) [59, 69]) partitioning mechanisms available in today's processors are either low-performing or do not provide isolation.

Unlike existing mechanisms such as CAT, DAWG disallows hits across protection domains. This affects hit paths and cache coherence [143], and DAWG handles these issues with minimal modification to modern operating systems, while reducing the attack surface of operating systems to a small set of annotated sections where data moves across protection domains, or where domains are resized/reallocated. Only in this handful of routines, DAWG protection is relaxed, and other defensive mechanisms such as speculation fences are applied as needed. We evaluate the performance implications of DAWG using a combination of architectural simulation and real hardware and compare to conventional and quality-of-service partitioned caches. We conclude that DAWG provides strong isolation with a reasonable performance overhead.

We describe a non-interference property that is orthogonal to speculative execution and therefore argue that existing attacks such as SPECTRE1.X [91, 96], SPECTRE2 [63], or SPECTRE4 [67] will not work on a system equipped with DAWG. DAWG is also effective against non-speculative side channel attacks, traditionally targeting cryptographic routines such as RSA and AES [24, 28, 139].

6.4 Background and Related Work

We focus on thwarting attacks by disrupting the channel between the victim's domain and the attacker for attacks that use cache state-based channels. We state our threat model in Section 6.4.1, describe relevant attacks in Section 6.4.2, and existing defenses in Section 6.4.3.

6.4.1 Threat model

Our focus is on blocking attacks that utilize the cache state exfiltration channel. We do not claim to disrupt other channels, such as L3 cache slice contention, L2 cache bank contention, network-on-chip or DRAM bandwidth contention, branch data structures, TLBs or shared functional units in a physical core. In the case of the branch data

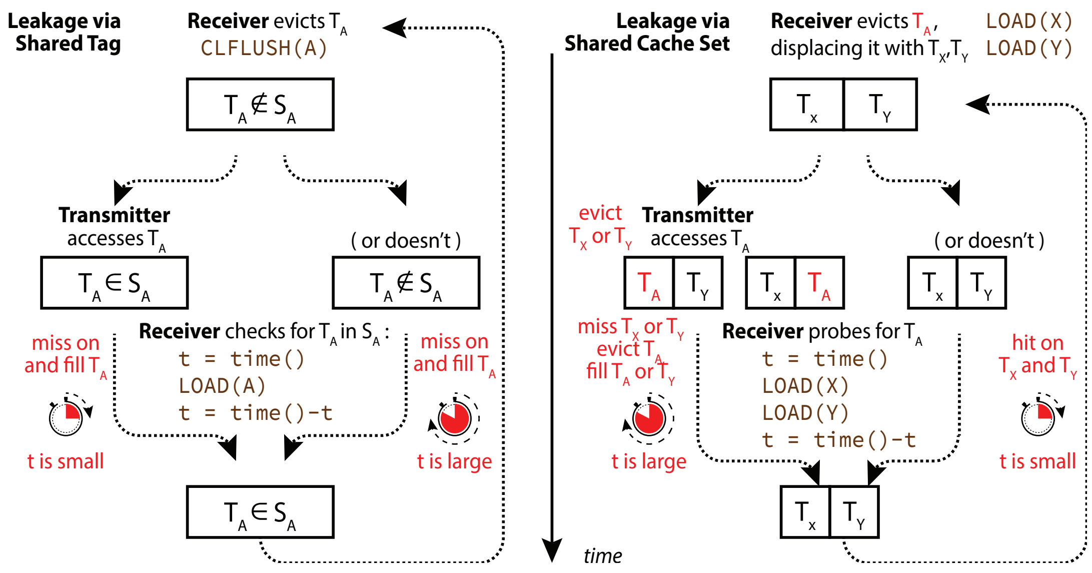


Figure 6-2: Leakage via a shared cache set, implemented via a shared tag T_A directly, or indirectly via $T_X, T_Y \cong T_A$.

structures, TLBs, or any other set associative structure, however, we believe that a DAWG-like technique can be used to block the channel associated with the state of those structures.

We assume an unprivileged attacker. The victim's domain can be privileged (kernel) code or an unprivileged process.

6.4.2 Attacks

The most common channel modulation strategy corresponds to the attacker presetting the cache tag state to a particular value, and then after the victim runs, observing a difference in the cache tag state to learn something about the victim process. A less common yet viable strategy corresponds to observing changes in coherence [196] or replacement metadata.

The replacement policy selects a cache line to evict (a "victim") when no invalid line is available on a fill. A review [11] of recent cache replacement and protection policies includes valuable commentary on less-cited excellent papers [73, 74, 75, 150, 151, 201] which have left little room for improvement in performance/power/area/complexity. Different replacement policies, however, offer different additional attack surfaces.

6.4.2.1 Cache tag state based attacks

Attacks using cache tag state-based channels are known to retrieve cryptographic keys from a growing body of cryptographic implementations: AES[24, 139], RSA [28], Diffie-Hellman [97], and elliptic-curve cryptography [27], to name a few. Such attacks can be mounted by unprivileged software sharing a computer with the victim software [13]. While early attacks required access to the victim's CPU core, more recent sophisticated channel modulation schemes such as flush+reload [197] and variants of prime+probe [120] target the last-level cache (LLC), which is shared by all cores in a socket. The evict+reload variant of flush+reload uses cache contention rather than flushing [120]. An attack in JavaScript that used a cache state-based channel was demonstrated [138] to automatically exfiltrate private information upon a web page visit.

These attacks use channels at various levels of the memory cache hierarchy and exploit cache lines shared between an attacker's program and the victim process. Regardless of the specific mechanism for inspecting shared tags, the underlying concepts are the same: two entities separated by a trust boundary share a channel based on shared computer system resources, specifically sets in the memory hierarchy. Thus, the entities can communicate (transmitting unwittingly, in the case of an attack) on that cross-trust boundary channel by modulating the presence of a cache tag in a set. The receiver can detect the transmitter's fills of tag T_A either directly, by observing whether it had fetched a shared line, or indirectly, by observing conflict misses on the receiver's own data caused by the transmitter's accesses, as shown in Figure 6-2.

6.4.3 Defenses

Broadly speaking, there are five classes of defenses, with each class corresponding to blocking one of the steps of the attack described in Figure 6-1.

1. *Prevent access to the secret.* For example, KAISER [37], which removes virtual address mappings of kernel memory when executing in user mode, is effective against Meltdown [116].
2. *Make it difficult to construct the data tap.* For example, randomizing virtual addresses of code, flushing the Branch Table Buffer (BTB) when entering the victim's domain[155].
3. *Make it difficult to launch the data tap.* For example, not speculatively executing through permission checks, keeping predictor state partitioned between domains, and preventing user arguments from influencing code with access to secrets. The Retpoline [184] defense against Spectre Variant 2 [29] makes it hard to launch (or construct) a data tap via an indirect branch.
4. *Reduce the bandwidth of side channels.* For example, removing the APIs for high-resolution timestamps in JavaScript, as well as support for shared memory buffers to prevent attackers from creating timers.
5. *Close the side channels.* Prevent the attacker and victim from having access to the same channel. For example, partitioning of cache state or predictor state.

The latter is the strategy of choice in our paper, and we consider three subclasses of prior approaches:

6.4.3.1 Set partitioning via page coloring

Set partitioning, i.e., not allowing occupancy of any cache set by data from different protection domains, can disrupt cache state-based channels. It has the advantage of working with existing hardware when allocating groups of sets at page granularity [115, 202] via page coloring [86, 178]. Linux currently does not support page coloring, since most early OS coloring was driven by the needs of low-associativity data caches[180].

Set partitioning allows communication between protection domains without destroying cache coherence. The downsides are that it requires some privileged entity, or collaboration, to move large regions of data around in memory when allocating cache sets, as set partitioning via page coloring binds cache set allocation to physical address allocation. For example, in order to give a protection domain 1/8 of the cache space, the same 12.5% of the system's physical address space must be given to the process. In an ideal situation, the amount of allocated DRAM and the amount of allocated cache space should be decoupled.

Furthermore, cache coloring at page granularity is not straightforwardly compatible with large pages, drastically reducing the TLB reach, and therefore performance, of processes. On current processors, the index bits placement requires that small (4KB) pages are used, and coloring is not possible for large (2MB) pages. Large pages provide critical performance benefits for virtualization platforms used in the public cloud [145], and reverting to small pages would be deleterious.

6.4.3.2 Insecure way and fine-grain partitioning

Intel's Cache Allocation Technology (CAT) [59, 69] provides a mechanism to configure each logical process with a *class of service*, and allocates LLC cache ways to logical processes. The CAT manual explicitly states that a cache access will hit if the line is cached in *any* of the cache's ways — this allows attackers to observe accesses of the victim. CAT only guarantees that a domain fill will not cause evictions in another domain. To achieve CAT's properties, no critical path changes in the cache are required: CAT's behavior on a cache hit is identical to a generic cache. Victim selection (replacement policy), however, must be made aware of the CAT configuration in order to constrain ways on an eviction.

Via this quality of service (QoS) mechanism, CAT improves system performance because an inefficient, cache-hungry process can be reined in and made to only cause evictions in a subset of the LLC, instead of trashing the entire cache. The fact that the cache checks all ways for cache hits is also good for performance: shared data need not be duplicated, and overhead due to internal fragmentation of cache ways is reduced. The number of ways for each domain can also be dynamically adjusted. For example, DynaWay [46] uses CAT with online performance monitoring to adjust the ways per domain.

CAT-style partitioning is unfortunately insufficient for blocking all cache state-based channels: an attacker sharing a page with the victim may observe the victim's use of shared addresses (by measuring whether a load to a shared address results in a cache hit). Furthermore, even though domains can fill only in their own ways, an attacker is free to flush shared cache lines regardless where they are cached, allowing straightforward transmission to an attacker's receiver via flush&reload, or flush&flush [54]. CAT-style partitioning allows an attacker to spy on lines cached in ways allocated to the victim, so long as the address of a transmitting line is mapped by the attacker. This is especially problematic when considering Spectre-style attacks, as the victim (OpenSSL, kernel, etc.) can be made to speculatively touch arbitrary addresses, including those in shared pages. In a more subtle channel, access patterns leak through metadata updates on hitting loads, as the replacement metadata is shared across protection domains.

Applying DAWG domain isolation to fine-grain QoS partitioning such as Vantage [161] would further improve scalability to high core counts. Securing Vantage, is similar to securing CAT: hits can be isolated, since each cache tag is associated with a partition ID; replacement metadata (timestamps or RRIP [75]) should be restricted to each partition; additionally Vantage misses allow interference, and demotion to the unmanaged 10% of the cache, which must be secured.

6.4.3.3 Reducing privacy leakage from caches

Since Spectre attacks are outside of the threat model anticipated by prior work, most prior defenses are ineffective. LLC defenses against cross-core attacks, such as SHARP [195] and RIC [84], do not stop same-core OS/VMM attacks. In addition, RIC's non-inclusive read-only caches do not stop speculative attacks from leaking through read-write cache lines in cache coherence attacks [182].

PLcache [99, 190] and the Random Fill Cache Architecture (RFill, [119]) were designed and analyzed in the context of a small region of sensitive data. RPcache [99, 190] trusts the OS to assign different hardware process IDs to mutually mistrusting entities, and its mechanism does not directly scale to large LLCs. The non-monopolizable cache [43] uses a well-principled partitioning scheme, but does not completely block all channels, and relies on the OS to assign hardware process IDs. CATalyst [118] trusts the Xen hypervisor to correctly tame Intel's Cache Allocation Technology into providing cache pinning, which can only secure software whose code and data fits into a fraction of the LLC, e.g., each virtual machine is given 8 "secure" pages. [174] similarly depends on CAT for the KVM (Kernel-based Virtual Machine) hypervisor. Using hardware transactional memory, Cloak [53] preloads secrets in cache within one transaction to prevent access pattern observation of secrets. Blocking channels used by speculative attacks, however, requires all addressable memory to be protected.

SecDCP [189] demonstrate dynamic allocation policies, assuming a secure partitioning mechanism is available; they provide only 'one-way protection' for a privileged enclave with no communication. DAWG offers the desired partitioning mechanism; we additionally enable two-way communication between OS and applications, and handle mutually untrusted peers at the same security level. We allow deduplication, shared libraries, and memory mapping, which in prior work must all be disabled.

6.5 Dynamically Allocated Way Guard (DAWG) Hardware

The objective of DAWG is to preclude the existence of any cache state-based channels between the attacker's and victim's domains. It accomplishes this by isolating the visibility of any state changes to a single protection domain, so any transmitter in the victim's domain cannot be connected to the same channel as any receiver in the attacker's domain. This prevents any communication or *leaks* of data from the victim to the attacker.

6.5.1 High-level design

Consider a conventional set-associative cache, a structure comprised of several *ways*, each of which is essentially a direct-mapped cache, as well as a controller mechanism. In order to implement Dynamically Allocated Way Guard (DAWG), we will allocate groups of ways to protection domains, restricting both cache hits and line replacements to the ways allocated to the protection domain from which the cache request was issued. On top of that, the *metadata* associated with the cache, e.g., replacement policy state, must also be allocated to protection domains in a well-defined way, and securely partitioned. These allocations will force strong isolation between the domains' interactions with one another via the cache structure.

DAWG's protection domains are *disjoint across ways and across metadata partitions*, except that protection domains may be nested to allow trusted privileged software access to all ways and metadata allocated to the protection domains in its purview.

Figure 6-3 shows the hardware structure corresponding to a DAWG cache, with the additional hardware required by DAWG over a conventional set-associative cache shown highlighted. The additional hardware state for each core is 24 bits per hardware thread – one register with three 8-bit active domain selectors. Each cache additionally needs up to 256 bits to describe the allowed hit and fill ways for each active domain (e.g., 16× intervals for a typical current 16-way cache).

6.5.2 DAWG's isolation policies

DAWG's protection domains are a high-level property orchestrated by software, and implemented via a table of *policy* configurations, used by the cache to enforce DAWG's isolation; these are stored at the DAWG cache in MSRs (model-specific registers). System software can write to these policy MSRs for each `domain_id` to configure the protection domains as enforced by the cache.

Each access to a conventional cache structure is accompanied with request metadata, such as a Core ID, as in Figure 6-3. DAWG extends this metadata to reference a *policy* specifying the protection domain (`domain_id`) as context for the cache access. For a last-level memory cache the `domain_id` field is required to allow system software to propagate the domain on whose behalf the access occurs, much like a capability. The hardware needed to endow each cache access with appropriate `domain_id` is described in Section 6.5.3.

Each policy consists of a pair of bit fields, all accessible via the DAWG cache's MSRs:

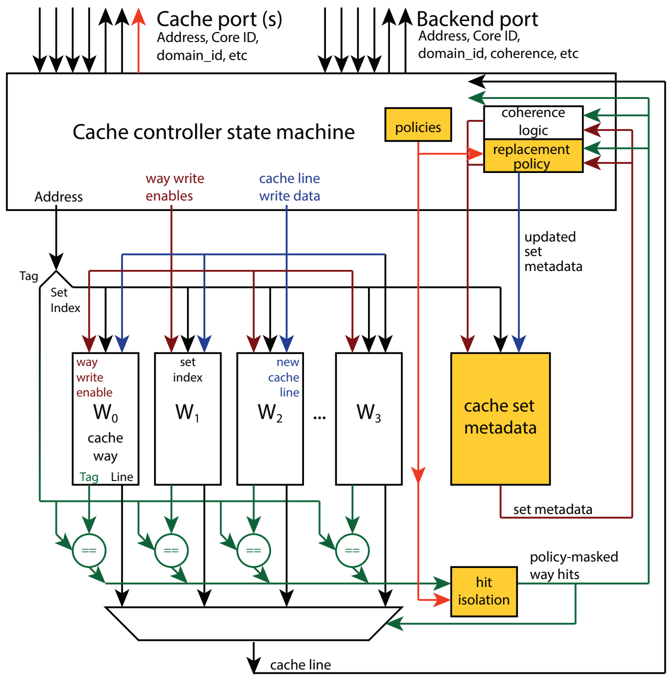


Figure 6-3: A Set-Associative Cache structure with DAWG.

- A policy_fillmap: a bit vector masking fills and victim selection, as described in Sections 6.5.4 and 6.5.5.
- A policy_hitmap: a bit vector masking way *hits* in the DAWG cache, as described in Section 6.5.6.

Each DAWG cache stores a table of these policy configurations, managed by system software, and selected by the cache request metadata at each cache access. Specifically, this table maps global domain_id identifiers to that domain's policy configuration in a given DAWG cache. We discuss the software primitives to manage protection domains, i.e., to create, modify, and destroy way allocations for protection domains, and to associate processes with protection domains in Section 6.6.1.1.

6.5.3 DAWG's modifications to processor cores

Each (logical) core must also correctly tag its memory accesses with the correct domain_id. To this end, we endow each hardware thread (logical core) with an MSR specifying the domain_id fields for each of the three types of accesses recognized by DAWG: instruction fetches via the instruction cache, read-only accesses (loads, flushes, etc), and modifying accesses (anything that can cause a cache line to enter the modified state, e.g., stores or atomic accesses). We will refer to these three types of accesses as *ifetches*, *loads*, and *stores*; (anachronistically[1], we name the respective domain selectors CS, DS, and ES). Normally, all three types of accesses are associated with

[1]As a nod to the now obsolete segment selectors in Intel's 8086.

the same protection domain, but this is not the case during OS handling of memory during communication across domains (for example when servicing a system call). The categorization of accesses is important to allow system software to implement message passing, and the indirection through domain selectors allows domain resizing, as described in Section 6.6.

The bit width of the `domain_id` identifier caps the number of protection domains that can be simultaneously scheduled to execute across the system. In practice, a single bit (differentiating kernel and user-mode accesses) is a useful minimum, and a reasonable maximum is the number of sockets multiplied by the largest number of ways implemented by any DAWG cache in the system (e.g., 16 or 20). An 8-bit identifier is sufficient to enumerate the maximum active domains even across 8-sockets with 20-way caches.

Importantly, MSR writes to each core's `domain_id`, and each DAWG cache's `policy_hitmap` and `policy_fillmap` MSRs must be a *fence*, prohibiting speculation on these instructions. Failing to do so would permit speculative disabling of DAWG's protection mechanism, leading to Spectre-style vulnerabilities.

6.5.4 DAWG's cache eviction/fill isolation

In a simple example of using DAWG at the last level cache (LLC), protection domain `0` (e.g., the kernel) is statically allocated half of DAWG cache's ways, with the other half allocated to unprivileged software (relegated to protection domain `1`). While the cache structure is shared among all software on the system, no access should affect observable cache state across protection domains, considering both the cache data and the metadata. This simple scenario will be generalized to dynamic allocation in Section 6.5.8 and we discuss the handling of cache replacement metadata in Section 6.5.10 for a variety of replacement policies.

Straightforwardly, cache misses in a DAWG cache must not cause fills or evictions outside the requesting protection domain's ways in order to enforce DAWG's isolation. Like Intel's CAT (Section 6.4.3.2), our design ensures that only the ways that a process has been allocated (via its protection domain's `policy_fillmap` policy MSRs) are candidates for eviction; but we also restrict `CLFLUSH` instructions. Hardware instrumentation needed to accomplish this is highlighted in Figure 6-3.

6.5.5 DAWG's cache metadata isolation

The *cache set metadata* structure in Figure 6-3 stores per-line helper data including replacement policy and cache coherence state. The metadata update logic uses tag comparisons (hit information) from all ways to modify set replacement state. DAWG does not leak via the coherence metadata, as coherence traffic is tagged with the requestors' protection domain and does not modify lines in other domains (with a sole exception described in Section 6.5.7).

DAWG's replacement metadata isolation requirement, at a high level, is a non-interference property: victim selection in a protection domain should not be affected by the accesses performed against any other protection domain(s). Furthermore, the cache's replacement policy must allow system software to sanitize the replacement data of a way in order to implement safe protection domain resizing. Details of implementing DAWG-friendly partitionable cache replacement policies are explored in Section 6.5.10.

6.5.6 DAWG's cache hit isolation

Cache hits in a DAWG cache must also be isolated, requiring a change to the critical path of the cache structure: a cache access must not hit in ways it was not allocated – a possibility if physical tags are shared across protection domains.

Consider a read access with address $A \implies (T_A, S_A)$ (tag and set, respectively) in a conventional set associative cache. A match on any of the way comparisons indicates a cache *hit* ($\exists\ i \mid T_{W_i} == T_A \implies hit$); the associated cache line data is returned to the requesting core, and the replacement policy metadata is updated to make note of the access. This allows a receiver (attacker) to communicate via the cache state by probing the cache tag or metadata state as described in Section 6.4.2.

In DAWG, tag comparisons must be masked with a policy (`policy_hitmap`) that white-lists ways allocated to the requester's protection domain, that is

$$\exists\ i \mid \texttt{policy\_hitmap}[\texttt{i}]\ \&\ (T_{W_i} == T_A) \implies hit.$$

By configuring `policy_hitmap`, system software can ensure cache hits are not visible across protection domains. While the additional required hardware in DAWG caches' hit path adds a gate delay to each cache access, we note that modern L1 caches are usually pipelined. We expect hardware designers will be able to manage an additional low-fanout gate without affecting clock frequency.

In addition to masking hits, DAWG's metadata update must use this policy-masked hit information to modify any replacement policy state safely, preventing information leakage across protection domains via the replacement policy state, as described in Section 6.5.5.

6.5.7 Cache lines shared across domains

DAWG effectively hides cache hits outside the white-listed ways as per `policy_hitmap`. While this prevents information leakage via adversarial observation of cached lines, it also complicates the case where addresses are shared across two or more protection domains by allowing ways belonging to different protection domains to have copies of the same line. Read-only data and instruction misses acquire lines in the **S**hared state of the MESI protocol [143] and its variants.

Neither a conventional set associative cache nor Intel's CAT permits duplicating a cache line within a cache: their hardware enforces a simple invariant that *a given tag can only exist in a single way of a cache at any time.* In the case of a DAWG cache, the hardware does not strictly enforce this invariant across protection domains; we allow read-only cache lines (in **S**hared state) to be replicated across ways in different protection domains. Replicating shared cache lines, however, may leak information via the cache coherence protocol (whereby one domain can invalidate lines in another), or violate invariants expected by the cache coherence protocol (by creating a situation where multiple copies of a line exist when one is in the **M**odified state).

In order to maintain isolation, cache coherence traffic must respect DAWG's protection domain boundaries. DAWG does not require any changes to the overall protocol and state diagrams, and only needs a wider physical address field in protocol messages to accommodate the representation of protection domains. Requests for the same line from different domains are therefore considered non-matching, and are filled by the memory controller. Cache flush instructions (`CLFLUSH`, `CLWB`) affect only the ways allocated to the requesting `domain_id`. Cross-socket invalidation requests must likewise communicate their originating protection domain. DAWG caches are *not*, however, expected to handle a replicated **M**odified line, meaning system software must not allow shared writable pages across protection domains via a TLB invariant, as described in Section 6.6.2.2.

Stale **S**hared lines of de-allocated pages may linger in the cache; DAWG must invalidate these before zeroing a page to be granted to a process (see Section 6.6.2.2). To this end, DAWG requires a new privileged MSR, with which to *invalidate all copies* of a **S**hared line, given an address, regardless of protection domain. DAWG relies on system software to prevent the case of a replicated **M**odified line, otherwise updates will not be visible by other domains.

6.5.8 Dynamic allocation of ways

It is unreasonable to implement a static protection domain policy, as it would make inefficient use of the cache resources due to internal fragmentation of ways. Instead, DAWG caches can be provisioned with updated security policies *dynamically*, as the system's workload changes.

In order to maintain its security properties, system software must manage protection domains by manipulating the domains' `policy_hitmap` and `policy_fillmap` MSRs in the DAWG cache. These MSRs are normally equal, but diverge to enable concurrent use of shared caches.

In order to re-assign a DAWG cache way, when creating or modifying the system's protection domains, the way must be *invalidated*, destroying any private information in form of the cache tags and metadata for the way(s) in question. In the case of write-back caches, dirty cache lines in the affected ways must be written-back, or swapped within the set. A privileged software routine flushes one or more ways via a hardware affordance to perform fine-grained cache flushes by set&way, e.g., available on ARM [6].

We require hardware mechanisms to flush a line and/or perform write-back (if **M**), of a specified way in a DAWG memory cache, allowing privileged software to orchestrate way-flushing as part of its software management of protection domains. This functionality is exposed for each cache, and therefore accommodates systems with diverse hierarchies of DAWG caches. We discuss the software mechanism to accommodate dynamic protection domains in Section 6.6.1.2.

While we have described the mechanism to adjust the DAWG policies in order to create, grow, or shrink protection domains, we leave as future work resource management support to securely determine the efficient sizes of protection domains for a given workload.

6.5.9 Scalability and cache organization

Scalability of the number of active protection domains is a concern with growing number of cores per socket. Since performance critical VMs or containers usually require multiple cores, however, the maximum number of active domains does not have to scale up to the number of cores.

DAWG on non-inclusive LLC caches [72] can also assign zero LLC ways to single-core domains, since these do not need communication via a shared cache. Partitioning must be applied to cache coherence metadata, e.g., snoop filters. Private cache partitioning allows a domain per SMT thread.

On inclusive LLC caches the number of concurrently active domains is limited by the number of ways — for high-core count CPUs this may require increasing associativity, e.g., from 20-way to 32-way. Partitioning replacement metadata allows high associativity caches with just 1 or 2 bits per tag for metadata to accurately select victims and remain secure.

6.5.10 Replacement policies

The optimal policy for a workload depends on the effective associativity and may even be software-selected, e.g., ARM A72[6] allows pseudo-random or pseudo-LRU cache-replacement. Our main requirement to avoid metadata leaks is that stateful replacement policies do not share any state across partitions. The modifications needed for several common replacement policies compatible with DAWG's isolation requirement are detailed in [90].

6.6 Software Modifications

We describe software provisions for modifying DAWG's protection domains, and also describe small, required modifications to several well-annotated sections of kernel software to implement cross-domain communication primitives robust against speculative execution attacks. Without these critical modifications system call input/output passing or any interprocess communication will be either impossible or exploitable.

6.6.1 Software management of DAWG policies

Protection domains are a software abstraction implemented by system software via DAWG's policy MSRs. The policy MSRs themselves (a table mapping protection `domain_id` to a `policy_hitmap` and `policy_fillmap` at each cache, as described in Section 6.5.2) reside in the DAWG cache hardware, and are atomically modified.

6.6.1.1 DAWG Resource Allocation

Protection domains for a process tree should be specified using the same `cgroup`-like interface as Intel's CAT. In order to orchestrate DAWG's protection domains and policies, the operating system must track the mapping of process IDs to protection domains. In a system with 16 ways in the cache with highest associativity, no more than 16 protection domains can be concurrently scheduled, meaning if the OS has need for more mutually distrusting entities to schedule, it needs to virtualize protection domains by time-multiplexing protection domain IDs, and flushing the ways of the multiplexed domain whenever it is re-allocated.

Another data structure, `dawg_policy`, tracks the resources (cache ways) allocated to each protection domain. This is a table mapping `domain_id` to pairs (`policy_hitmap`, `policy_fillmap`) for each DAWG cache. The kernel uses this table when resizing, creating, or destroying protection domains in order to maintain an exclusive allocation of ways to each protection domain. Whenever one or more ways are re-allocated, the supervisor must look up the current `domain_id` of the owner, accomplished via either a search or a persistent inverse map cache way to `domain_id`.

6.6.1.2 Secure Dynamic Way Reassignment

When modifying an existing allocation of ways in a DAWG cache (writing policy MSRs), as necessary to create or modify protection domains, system software must sanitize (including any replacement metadata, as discussed in Section 6.5.5) the re-allocated way(s) before they may be granted to a new protection domain. The process for re-assigning cache way(s) proceeds as follows:

1. Update the `policy_fillmap` MSRs to disallow fills in the way(s) being transferred out of the shrinking domain.
2. A software loop iterates through the cache's set indexes and flushes all sets of the re-allocated way(s). The shrinking domain may hit on lines yet to be flushed, as `policy_hitmap` is not yet updated.
3. Update the `policy_hitmap` MSRs to exclude ways to be removed from the shrinking protection domain.
4. Update the `policy_hitmap` and `policy_fillmap` MSRs to grant the ways to the growing protection domain.

Higher level policies can be built on this dynamic way-reassignment mechanism.

6.6.1.3 Code Prioritization

Code prioritization reduces the performance impact of disallowing code sharing across domains, especially when context switching between untrusted domains sharing code. Programming the domain selectors for code and data separately allows ways to be dedicated to code without data interference. Commercial studies of code cache sensitivity of production server workloads [72, 83, 140] show large instruction miss rates in L2, but even the largest code working sets fit within 1–2 L3 ways.

DAWG domains cannot reuse shared code in L2 in the worst case scenario of context switching to untrusted domains. While partitioning protects against context switching to other code, adding dynamic way reassignment and code prioritization can offset the loss of sharing.

6.6.2 Kernel changes required by DAWG

Consider a likely configuration where a user-mode application and the OS kernel are in different protection domains. In order to perform a system call, communication must occur across the protection domains: the supervisor extracts the (possibly cached) data from the caller by copying into its own memory. In DAWG, this presents a challenge due to strong isolation in the cache.

6.6.2.1 DAWG augments SMAP-annotated sections

We take advantage of modern OS support for the Supervisor Mode Access Prevention (SMAP) feature available in recent x86 architectures, which allows supervisor mode programs to raise a trap on accesses to user-space memory. The intent is to harden the kernel against malicious programs attempting to influence privileged execution via untrusted user-space memory. At each routine where supervisor code intends to access user-space memory, SMAP must be temporarily disabled and subsequently re-enabled via `stac` (Set AC Flag) and `clac` (Clear AC Flag) instructions, respectively. We observe that a modern kernel's interactions with user-space memory are diligently annotated with these instructions, and will refer to these sections as *annotated sections*.

Currently, Linux kernels use seven such sections for simple memory copy or clearing routines: `copy_from_user`, `copy_to_user`, `clear_user`, `futex`, etc. We propose extending these annotated sections with short instruction sequences to correctly handle DAWG's communication requirements on system calls and inter-process communication, in addition to the existing handling of the SMAP mechanism. Specifically, as illustrated on Figure 6-4, data movement *from* user to kernel memory is annotated with an MSR write to `domain_id`: *ifetch* and *store* accesses proceed on behalf of the kernel, as before, but *load* accesses use the caller's (user) protection domain. This allows the kernel to efficiently copy from warm cache lines, but preserves isolation. After copying from the user, the `domain_id` MSR is restored to perform all accesses on behalf of the kernel's protection domain. Likewise, sections implementing data movement *to* user memory *ifetch* and *load* on behalf of the kernel's domain, but *store* in the user's cache ways. While the annotated sections may be interrupted by asynchronous events, interrupt handlers are expected to explicitly set `domain_id` to the kernel's protection domain, and restore the MSR to its *prior* state afterwards.

Finally, to guarantee isolation, we require the annotated sections to contain only code that obeys the following properties: to protect against known and future speculative attacks, indirect jumps or calls, returns [91], and potentially unsafe branches are not to be crossed. Further, we cannot guarantee that these sections will not require patching as new attacks are discovered, although this is reasonable given the small number and size of the annotated sections.

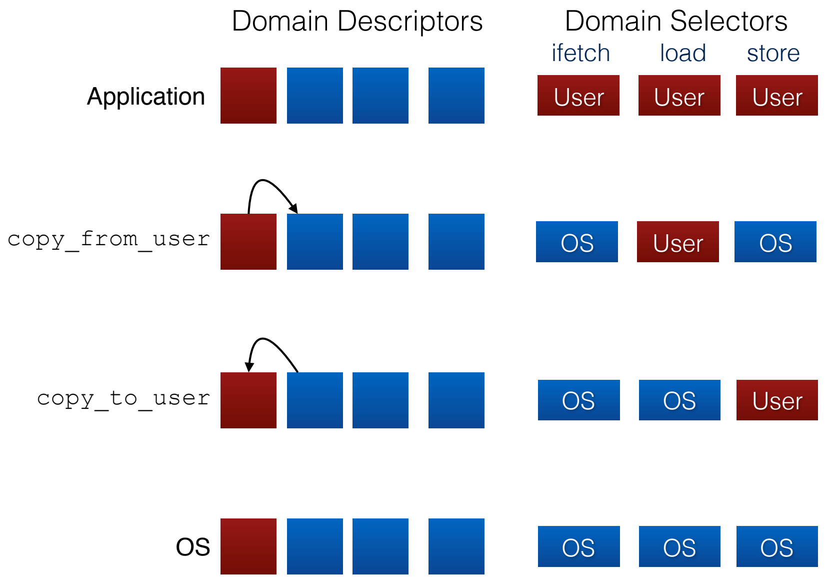


Figure 6-4: Typical uses of domain selectors. Normal application where all selectors are mapped to the User domain descriptor. OS routines, including `copy_from_user` and `copy_to_user` point the code selector *ifetch* to the OS domain descriptor, and where necessary allow *load*/*store* from/to the User domain.

6.6.2.2 Read-only and CoW sharing across domains

For memory efficiency, DAWG allows securely mapping read-only pages across protection domains, e.g., for shared libraries, requiring hardware cache coherence protocol changes (see Section 6.5.7), and OS/hypervisor support.

This enables conventional system optimizations via page sharing, such as read-only `mmap` from page caches, Copy-on-Write (CoW) conventionally used for `fork`, or for page deduplication across VMs (e.g., Transparent Page Sharing [188]; VM page sharing is typically disabled due to concerns raised by shared cache tag attacks [70]). DAWG maintains security with read-only mappings across protection domains to maintain memory efficiency.

Dirty pages can be prepared for CoW sharing eagerly, or lazily (but cautiously against page fault attacks [194]) by installing non-present pages in the consumer domain mapping. Preparing a dirty page for sharing *requires* a write-back of any dirty cache lines on behalf of the producer's domain (via `CLWB` instructions and an appropriate *load* `domain_id`). The writeback guarantees that read-only pages appear only as **S**hared lines in DAWG caches, and can be replicated across protection domains as described in Section 6.5.7.

A write to a page read-only shared across protection domains signals the OS to create a new, private copy using the original producer's `domain_id` for reads, and the consumer's `domain_id` for writes.

6.7 Security and Limitations of Cache Partitioning

We explain why DAWG protects against practical attacks on speculative execution processors by stating and arguing a non-interference property in Section 6.7.1. Then we show a generalization of our attack schema in Section 6.7.2 to point out the limitations of cache partitioning, and generally egress-oriented defenses.

6.7.1 DAWG Isolation Property

DAWG enforces isolation of *exclusive* protection domains among cache tags and replacement metadata, as long as:

1. victim selection is restricted to the ways allocated to the protection domain (an invariant maintained by system software), and
2. metadata updates as a result of an access in one domain do not affect victim selection in another domain (a requirement on DAWG's cache replacement policy).

Together, this guarantees non-interference – the hits and misses of a program running in one protection domain are unaffected by program behavior in different protection domains. As a result, DAWG blocks the cache tag and metadata channels of non-communicating processes separated by DAWG's protection domains.

6.7.2 Cache Reflection Attack

DAWG's cache isolation goals are meant to approach the isolation guarantees of separate machines, yet, even remote network services can fall victim to leaks employing cache tag state for communication. Consider the example of the attacker and victim residing in different protection domains, sharing no data, but communicating via some API, such as system calls. As in a remote network timing leak [28], where network latency is used to communicate some hidden state in the victim, the *completion time* of API calls can communicate insights about the cache state [96] within a protection domain. Leakage via *reflection* through the cache is thus possible: the receiver invokes an API call that accesses private information, which affects the state of its private cache ways. The receiver then exfiltrates this information via the latency of another API call. Figure 6-5 shows a cache timing leak which relies on cache reflection entirely within the victim's protection domain. The syscall completion time channel is used for exfiltration, meaning no private information crosses DAWG's domain boundaries in the caches, rendering DAWG, and cache partitioning in general, ineffective at closing a leak of this type.

The transmitter is instructed via an API call to access $a[b[i]]$, where i is provided by the receiver (via `syscall1`), while a, b reside in the victim's protection domain. The cache tag state of the transmitter now reflects $b[i]$, affecting the latency of subsequent syscalls in a way dependent on the secret $b[i]$. The receiver now exfiltrates information about $b[i]$ by selecting a j from the space of possible values of $b[i]$ and measuring the completion time of `syscall2`, which accesses $a[j]$. The syscall completion time communicates whether the transmitter hits on $a[j]$, which implies $a[j] \cong a[b[i]]$, and

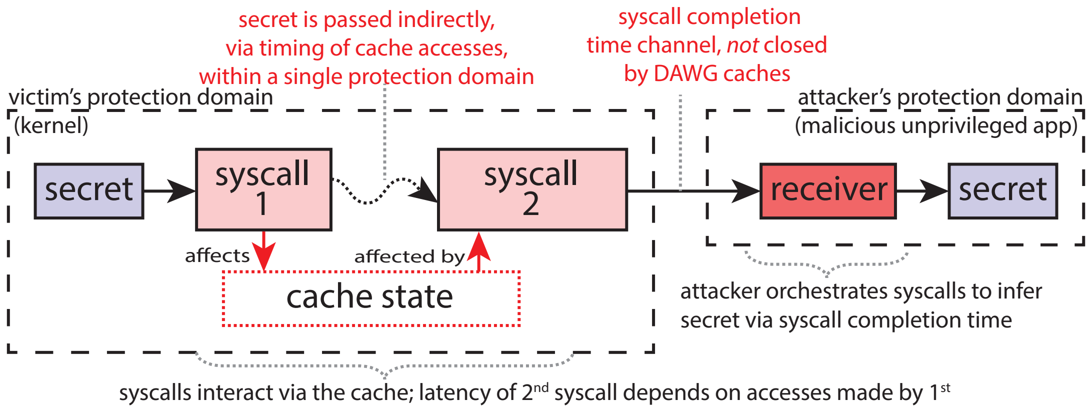

secret is passed indirectly, via timing of cache accesses, within a single protection domain
syscall completion time channel, *not* closed by DAWG caches
victim's protection domain (kernel)
attacker's protection domain (malicious unprivileged app)
secret
syscall 1
syscall 2
receiver
secret
affects
affected by
cache state
attacker orchestrates syscalls to infer secret via syscall completion time
syscalls interact via the cache; latency of 2nd syscall depends on accesses made by 1st

Figure 6-5: Generalized Attack Schema: an adversary 1) accesses a victim's secret, 2) reflects it to a transmitter 3) transmits it via a covert channel, 4) receives it in their own protection domain.

for a compact a, that $b[i] = j$ – a leak. This leak can be amplified by initializing cache state via a bulk memory operation, and, for a machine-local receiver by malicious mis-speculation. While we have demonstrated this attack on real hardware, remote attacks are much more difficult and have lower bandwidth than local attacks.

While not the focus of this chapter, for completeness, we outline a few countermeasures for this type of leak. Observe that the completion time of a public API is used here to exfiltrate private information. The execution time of a syscall can be padded to guarantee constant (and worst-case) latency, no matter the input or internal state. This can be relaxed to bound the leak to a known number of bits per access [48].

A zero leak countermeasure requires destroying the transmitting domain's cache state across syscalls/API invocations, preventing reflection via the cache. DAWG can make this less inefficient: in addition to dynamic resizing, setting the replacement mask `policy_fillmap` to a subset of the `policy_hitmap` allows locking cache ways to preserve the hot working set. This ensures that all unique cache lines accessed during one request have constant observable time.

6.8 Evaluation

To evaluate DAWG, we use the `zsim` [162] execution-driven x86-64 simulator and Haswell hardware [45] for our experiments.

6.8.1 Configuration of insecure baseline

Table 6.1 summarizes the characteristics of the simulated environment. The out-of-order model implemented by `zsim` is calibrated against Intel Westmere, informing our choice of cache and network-on-chip latencies. The DRAM configuration is typical for contemporary servers at ∼5 GB/s theoretical DRAM bandwidth per core. Our baseline uses the Tree-PLRU replacement policy for private caches, and a 2-bit NRU for the shared LLC. The simulated model implements inclusive caches, although DAWG domains with reduced associativity would benefit from relaxed inclusion [72]. We simulate CAT partitioning at *all* levels of the cache, while modern hardware only offers this at the LLC. We do this by restricting the replacement mask `policy_fillmap`, while white-listing all ways via the `policy_hitmap`.

6.8.2 DAWG Policy Scenarios

We evaluate several protection domain configurations for different resource sharing and isolation scenarios.

6.8.2.1 VM or container isolation on dedicated cores

Isolating peer protection domains from one another requires equitable LLC partitioning, e.g., 50% of ways allocated to two active domains. In the case of cores dedicated to each workload (no context switches), each scheduled domain is assigned the entirety of its L1 and L2.

6.8.2.2 VM or container isolation on time-shared cores

To allow the OS to overcommit cores across protection domains (thus requiring frequent context switches between domains), we also evaluate a partitioned L2 cache.

6.8.2.3 OS isolation

Only two DAWG domains are needed to isolate an OS from applications. For processes with few OS interventions in the steady state, e.g., SPECCPU workloads, the OS can reserve a single way in the LLC, and flush L1 and L2 ways to service the rare system calls. Processes utilizing more OS services would benefit from more ways allocated to OS's domain.

6.8.3 DAWG versus insecure baseline

Way partitioning mechanisms reduce cache capacity and associativity, which increases conflict misses, but improves fairness and reduces contention. We refer to CAT [59] for analysis of the performance impact of way partitioning on a subset of SPEC CPU2006. Here, we evaluate CAT and DAWG on parallel applications from PARSEC [20], and

Table 6.1: Simulated system specifications.

Cores		**DRAM**	**Bandwidth**
Count	Frequency	Controllers	*Peak*
8 OoO	3 GHz	4 x DDR3-1333	42 GB/s

Private Caches			**Shared Cache**	
L1	L2	Organization	L3	Organization
2× 32 KB	256 KB	8-way PLRU	8× 2 MB	16-way NRU

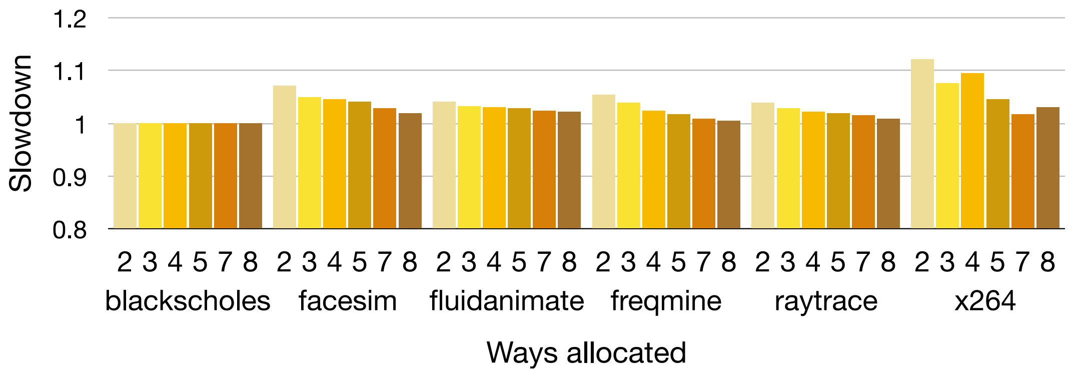


Figure 6-6: Way partitioning performance at low associativity in all caches (8-way L1, 8-way L2, and 16-way L3).

parallel graph applications from the GAP Benchmark Suite (GAPBS) [17] (the baselines described in Section 3.6, without **`milk`**). We take advantage of the graph generators in GAPBS to sweep workload sizes to demonstrate sensitivity to cache capacity and associativity.

Figure 6-6 shows DAWG partitioning of private L1 and L2 (Section 6.8.2.2) caches in addition to the L3. We explore DAWG configurations on a subset of PARSEC benchmarks on `simlarge` workloads. The cache insensitive `blackscholes` (or omitted `swaptions` with 0.001 L2 MPKI (Misses Per 1000 Instructions)) are unaffected at any way allocation. For a VM isolation policy (Section 6.8.2.1) with $^8/_{16}$ of the L3, even workloads with higher MPKI such as `facesim` show at most 2% slowdown. The $\langle ^2/_8$ L2, $^2/_{16}$ L3$\rangle$ configuration is affected by both capacity and associativity reductions, yet most benchmarks have 4–7% slowdown, up to 12% for x264. Such an extreme configuration can accommodate 4 very frequently context switched protection domains.

Figure 6-7 shows the performance of protection domains using different fractions of an L3 cache for 4-thread instances of graph applications from GAPBS. We use variable size synthetic *power law* graphs [30, 52] that match the structure of real-world social and web graphs and therefore exhibit cache locality [18]. The power law structure, however, implies that there is diminishing return from each additional L3 way. As shown, at half cache capacity ($^8/_{16}$ L3, Section 6.8.2.1), there is at most 15% slowdown (`bc` and `tc` benchmarks) at the largest simulated size (2^{20} vertices). A characteristic *eye* is formed when the performance curves of different configurations cross over the working set boundary (e.g., graph size of 2^{17}). Performance with working sets smaller or larger than the effective cache capacity is unaffected — at the largest size `cc`, `pr`, and `sssp` show 1–4% slowdown.

Reserving for the OS (Section 6.8.2.3), one way (6% of LLC capacity) adds no performance overhead to most workloads. The only exception would be a workload caught in the *eye*, e.g., PageRank at 2^{17} has 30% overhead (Figure 6-7), while at 2^{16} or 2^{18} — 0% difference.

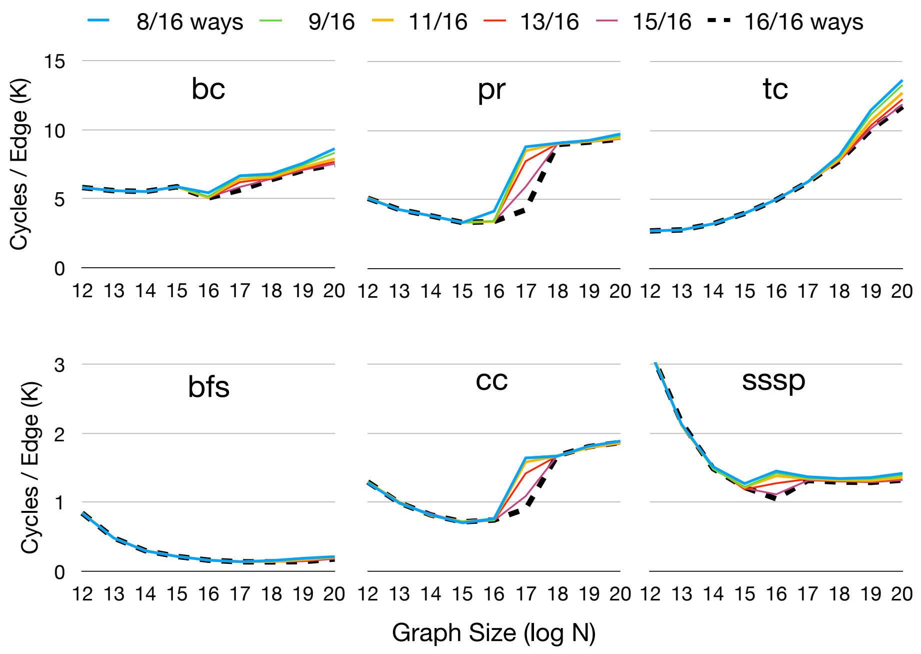


Figure 6-7: Way partitioning performance with varying working set on graph applications [17]. Simulated 16-way L3.

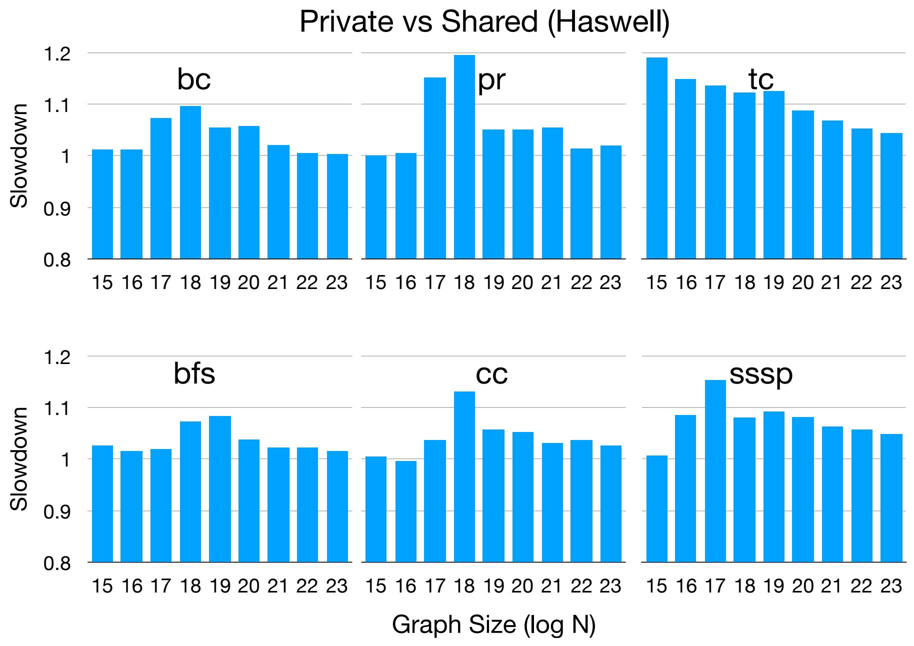


Figure 6-8: Read-only sharing effects of two instances using Shared vs Private data of varying scale (1-thread instances). Actual Haswell 20-way 30 MB L3.

6.8.4 CAT versus DAWG

We analyze and evaluate scenarios based on the degree of code and data sharing across domains.

6.8.4.1 No Sharing

There is virtually no performance difference between secure DAWG partitioning, and insecure CAT partitioning in the absence of read-sharing across domains.

DAWG reduces interference in replacement metadata updates and enforces the intended replacement strategy within a domain, while CAT may lose block history effectively exhibiting random replacement – a minor, workload-dependent perturbation. In simulations (not shown), we replicate a known observation that random replacement occasionally performs better than LRU near cache capacity. We did not observe this effect with NRU replacement.

6.8.4.2 Read-only Sharing

CAT QoS guarantees a lower bound on a workload's effective cache capacity, while DAWG isolation forces a tight upper bound. DAWG's isolation reduces cache capacity compared to CAT when cache lines are read-only shared across mutually untrusting protection domains. CAT permits hits across partitions where code or read-only data are unsafely shared. We focus on read-only data in our evaluation, as benchmarks with low L1i MPKI like GAPBS, PARSEC, or SPECCPU are poorly suited to study code cache sensitivity.

We analyze real applications using one line modifications to GAPBS to `fork` (a single-thread process) either before or after creating in-memory graph representations. The first results in a private graph for each process, while the latter simulates `mmap` of a shared graph. The shared graphs access read-only data across domains in the baseline and CAT, while DAWG has to replicate data in domain-private ways. Since `zsim` does not simulate TLBs, we ensure different virtual addresses are used to avoid false sharing. We first verified in simulation that DAWG, with memory shared across protection domains, behaves identically to CAT and the baseline with private data.

Next, we demonstrate (in Figure 6-8) that for most data sizes these benchmarks show little performance difference on real hardware (Table 6.2 shows the characteristics of the Haswell server [45]). Here Shared baseline models Shared CAT, while Private baseline models Shared DAWG. The majority of cycles are spent on random accesses to read-write data, while read-only data is streamed sequentially. Although read-only data is much larger than read-write data (e.g., 16 times more edges than vertices), prefetching and scan- and thrash- resistant policies [75, 151] further reduce the need for cache resident read-only data. Note that even at 2^{23} vertices these effects are immaterial; real-world graphs have billions of people or pages.

Table 6.2: Hardware system specifications [45].

Cores		**DRAM**	**Bandwidth**
Count	Frequency	Controllers	*Peak*
12 OoO	3.3 GHz	4 x DDR4-2133	68 GB/s

Private Caches			**Shared Cache**	
L1	L2	Organization	L3	Organization
2× 32 KB	256 KB	8-way	30 MB	20-way

6.9 Future Work

Additional performance optimizations for DAWG-aware applications can be enabled by future extensions for hardware/software co-design.

6.9.1 Streaming Data Isolation

Separate domain selectors and descriptors can be used even within the same application to take advantage of static data classification [128]. For example, in graph applications, edges are streamed with little reuse, while vertex data may have good locality; different domain descriptors can then be configured with a 1-way partition for edges, and a 3-way partition for vertex data. Vertex data is often subject to read-modify-write operations which are treated as stores, while edge reads are regular loads, therefore the domain selectors may only need to be configured infrequently. Alternatively, each instruction can have a static hint: for example, on x86 the instruction prefixes for the DS and ES segment selectors (which are unused in 64-bit mode) can be repurposed as domain selectors. By keeping fewer ways for the data with little reuse, a larger useful working set is preserved for better performance.

6.9.2 Cache-tuned Applications

The optimization techniques employed in Milk (in Chapter 3) and our high-performance graph framework Cagra [203] become even more relevant for applications under cache partitioning. Naive applications may experience dramatic performance 'cliffs' at cache boundaries. If the evaluated graph applications were taking advantage of Milk or Cagra, the observed cliffs can disappear, as the marginal utility of additional ways is even lower for cache-tuned algorithms. Cagra is tuned to fit segments of graph vertex data in L3, yet changes to segment sizes result in very slow changes to performance. Milk is insensitive to L3 size, thus only L2 partitioning would affect its performance.

The flip side of cache tuning is that the applications need to know the effective cache size allocated. Cache-oblivious algorithms automatically adapt to cache capacity changes, but are not as efficient as cache-aware algorithms. Yet, cache-aware optimized systems may experience cliffs if they are not notified that the effective cache capacity is reduced, and for CAT/DAWG way-partitioning even that the effective associativity is reduced. Adaptive solutions where applications attempt to measure the cache capacity/associativity at startup would fail to respond to allocation changes.

Cache-aware algorithms tuned to maximize hardware efficiency therefore must be aware of changes to cache way allocations. OS or CPU interfaces are needed to give applications an opportunity to query cache parameters, to receive dynamic notification when cache allocations change, or to even bid for the desired cache size.

6.10 Conclusion

Cache tag and metadata state attacks are the easiest to mount and offer very high ex-filtration bandwidth. DAWG protects against attacks that rely on a cache state-based channel, which are commonly referred to as cache-timing attacks, on speculative execution processors with reasonable overheads. The same policies can be applied to any set-associative structure, e.g., TLB or branch history tables. DAWG has its limitations and additional techniques are required to block exfiltration channels different from the cache channel.

Yet, we believe that pragmatic techniques like DAWG are needed to restore our confidence in public cloud infrastructure, while hardware and software co-design will help minimize performance overheads.

A good proxy for the performance overheads of secure DAWG is Intel's existing, though insecure, CAT hardware. Traditional QoS uses of CAT, however, differ from desired DAWG protection domains' configurations. Research on software resource management strategies can therefore commence with evaluation of large scale workloads on CAT. CPU vendors can similarly analyze the cost-benefits of increasing cache capacity and associativity to accommodate larger numbers of active protection domains. Ultimately, transforming CAT to DAWG offers a natural evolution path for both CPU and OS vendors.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 7

Conclusion

Ever increasing volumes of data need to be analyzed in current and upcoming applications. Billions of ever-connected users need to be served by more and more complex applications that need to draw on vast datasets for real-time decisions. Intelligent machines also continue to enjoy unprecedented ubiquity in our lives. Yet, Moore's law is expected to soon retreat to a distant memory, while Amdahl's law is unrelenting: we only expect diminishing returns from parallelization and vectorization.

We believe the novel techniques explored in this thesis in software, or with possible hardware co-design will reduce the bottlenecks in important classes of applications. This thesis has presented novel techniques for reducing the effective latency of indirect memory references, both by improving locality (with Milk in Chapter 3) and by improving memory parallelism (with Cimple in Chapter 4).

Public cloud computing has become the most practical platform for most data analytics. Yet, the recent speculative side-channel attacks opened new security concerns (in Chapter 5). We showed the vulnerabilities in existing mechanisms for improving indirect memory reference performance — caches, speculative execution, and SMT — and showed a hardware design (DAWG in Chapter 6) for improving security with minimal complexity and overhead. Putting it all together, we can achieve both solid security and high performance for critical applications.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Cimple: Schedulers

A.1 Simplest Dynamic Scheduler

```
template<int Width, typename CoroutineState,
         typename Refill>
void SimplestScheduler(size_t tasks,
                       Refill refill
                       )
{
  CoroutineState cs[Width];
  // fill
  for(size_t i=0; i<Width; i++)
      refill(&cs[i], i);
  size_t nextTask = Width;
  while (nextTask < tasks) {
     for(size_t i=0; i<Width; i++) {
        if (cs[i].Step() && nextTask < tasks) {
            refill(&cs[i], nextTask);
            nextTask++;
        }
     }
  }
  // drain
  for(size_t i=0; i<Width; i++) {
     while (!cs[i].Done() && !cs[i].Step()) {
         // no further refill!
     }
  }
}
```

A.2 Push and Pull Callback Scheduler

```
template<int Width, typename CoroutineState,
         typename Push, typename Pull = NoPull>
void PushPullScheduler(Push push, Pull pull = Pull())
{
    CoroutineState cs[Width];
    bool draining = false;

    // fill
    for(size_t i=0; i<Width; i++)
        if (!push(&cs[i]))
            draining = true;

    int sup = 0;
    while (!draining) {
        for(size_t i=0; i<Width; i++) {
            if (cs[i].Step()) {
                if (!pull(&cs[i])) {
                    cs[i].Reset(); // generator
                } else if (!push(&cs[i]))
                    draining = true;
            }
        }
    }

    // drain
    for(size_t i=0; i<Width; i++) {
        while (!cs[i].Done() && !cs[i].Step()) {
            if ( !pull(&cs[i]) ) {
                cs[i]._state = 0;
            }
        }
    }
}
```

A.3 Coroutine and Static Schedule for Hash table lookup

```
// Do not edit by hand.
// Automatically generated with Cimple v0.1 \
from hashprobe_linear.cimple.cpp.
// Compile with -std=c++11

// Original HashTable_find
KeyType
HashTable_find(
        KeyType k)
{
   HashType hash;
     hash = Murmur3::fmix(k);
     hash &= mask;

  while (ht[hash].key != k &&
         ht[hash].key != 0) {
       hash++;
       if (hash == size) hash = 0;
  }
     return ht[hash].key;
```

```
}

// Coroutine state for HashTable_find
struct CoroutineState_HashTable_find {
  CoroutineState_HashTable_find() : _state(0) {}
  CoroutineState_HashTable_find(KeyType k) :
        _state(0),
        k(k){ }
  int _state = 0;
  KeyType _result;
  KeyType k;
  HashType hash;
  bool Done() const { return _state == _Finished; }
  void Reset() { _state = 0; }
  KeyType Result() const { return _result ; }
  bool Step() {
    switch(_state) {
    case 0:
        hash = Murmur3::fmix(k);
        hash &= mask;
        _state = 1;
        return false;
    case 1:;
         // prefetch(&ht[hash]);
        _mm_prefetch((char*)(&ht[hash]),
                     _MM_HINT_T0);
        _state = 2;
        return false;
    case 2:;

  while (ht[hash].key != k &&
         ht[hash].key != 0) {
       hash++;
       if (hash == size) hash = 0;
  }
        _result = ht[hash].key;
        _state = _Finished;
        return true;
    } // switch
    return false;
  }
  constexpr static int InitialState = 0;
  constexpr static int _Finished = 3;
  enum class State {
    Initial  = 0,
    Final  = 3
  };
};

// Coroutine SoA state for HashTable_find x 8
template <int _Width = 8>
struct CoroutineState_HashTable_find_8 {
  int _state[_Width]  ;
  KeyType _result[_Width];
  KeyType _soa_k[_Width];
  HashType _soa_hash[_Width];
  bool SuperStep() {
  for(int _i = 0; _i < _Width ; _i++) {
  KeyType& k = _soa_k[_i];
  HashType& hash = _soa_hash[_i];
        hash = Murmur3::fmix(k);
        hash &= mask;
        }
        for(int _i = 0; _i < _Width ; _i++) {
        KeyType& k = _soa_k[_i];
        HashType& hash = _soa_hash[_i];
         // prefetch(&ht[hash]);
        _mm_prefetch((char*)(&ht[hash]),
                     _MM_HINT_T0);
        }
        for(int _i = 0; _i < _Width ; _i++) {
        KeyType& k = _soa_k[_i];
        HashType& hash = _soa_hash[_i];

  while (ht[hash].key != k &&
         ht[hash].key != 0) {
       hash++;
       if (hash == size) hash = 0;
  }
        _result[_i] = ht[hash].key;
  }
    return true;
  }
  void Init(  KeyType* k)  {
     for(int _i=0; _i<_Width; _i++) {
    _soa_k[_i] = k[_i];
     }
  }
  void Fini(KeyType*out)  {
     for(int _i=0; _i<_Width; _i++) {
        out[_i] = _result[_i];
     }
  }
 };
// End of batched HashTable_find
```

Bibliography

[1] OpenMP Application Program Interface 4.0. http://openmp.org/wp/openmp-specifications/, 2013.

[2] Intel Core i7-4790K Processor (8M Cache, up to 4.40 GHz). http://ark.intel.com/products/80807/Intel-Core-i7-4790K-Processor-8M-Cache-up-to-4_40-GHz, 2014.

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems. http://tensorflow.org/, 2015. Available at tensorflow.org.

[4] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. Massively parallel sort-merge joins in main memory multi-core database systems. *Proc. VLDB Endow.*, 5(10):1064–1075, June 2012.

[5] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Proceedings of the 4th International Conference on Supercomputing*, ICS '90, pages 1–6, New York, NY, USA, 1990. ACM.

[6] ARM. ARM Cortex-A72 MPCore processor technical reference manual, 2015.

[7] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. LinkBench: A database benchmark based on the Facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1185–1196, New York, NY, USA, 2013. ACM.

[8] AWS. Introducing Amazon EC2 high memory instances with up to 12 TB of memory. https://aws.amazon.com/about-aws/whats-new/2018/09/introducing-amazon-ec2-high-memory-instances-purpose-built-\to-run-large-in-memory-databases/, September 2018.

[9] Lars Backstrom, Paolo Boldi, Marco Rosa, Johan Ugander, and Sebastiano Vigna. Four degrees of separation. In *Proceedings of the 4th Annual ACM Web Science Conference*, WebSci '12, pages 33–42, New York, NY, USA, 2012. ACM.

[10] D. A. Bader, G. Cong, and J. Feo. On the architectural requirements for efficient execution of graph algorithms. In *Parallel Processing, 2005. ICPP 2005. International Conference on*, pages 547–556, June 2005.

[11] Rajeev Balasubramonian, Norman Jouppi, and Naveen Muralimanohar. *Multi-Core Cache Hierarchies.* Morgan & Claypool Publishers, 1st edition, 2011.

[12] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. Main-memory hash joins on modern processor architectures. *IEEE Trans. Knowl. Data Eng.*, 27(7):1754–1766, 2015.

[13] Sebastian Banescu. Cache timing attacks. 2011. [Online; accessed 26-January-2014].

[14] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the killer microseconds. *Commun. ACM*, 60(4):48–54, March 2017.

[15] S. Beamer, K. Asanović, and D. Patterson. Reducing PageRank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831, May 2017.

[16] Scott Beamer, Krste Asanović, and David Patterson. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[17] Scott Beamer, Krste Asanović, and David A. Patterson. The GAP benchmark suite. *CoRR*, abs/1508.03619, 2015.

[18] Scott Beamer, Krste Asanović, and David A. Patterson. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*, pages 56–65, 2015.

[19] Scott Beamer, Krste Asanović, and David A. Patterson. Reducing PageRank communication via propagation blocking. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 820–831, May 2017.

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

[21] Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, and Julian Shun. Internally deterministic parallel algorithms can be fast. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 181–192, 2012.

[22] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: An efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[23] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 54–65, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[24] Joseph Bonneau and Ilya Mironov. Cache-collision timing attacks against AES. In *Cryptographic Hardware and Embedded Systems-CHES 2006*, pages 201–215. Springer, 2006.

[25] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, USENIX ATC'13, pages 49–60, Berkeley, CA, USA, 2013. USENIX Association.

[26] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, pages 99–107, New York, NY, USA, 1996. ACM.

[27] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *Computer Security–ESORICS*. Springer, 2011.

[28] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 2005.

[29] Chandler Carruth. Introduce the "retpoline" x86 mitigation technique for variant #2 of the speculative execution vulnerabilities. `http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20180101/513630.html`, January 2018.

[30] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-MAT: A recursive model for graph mining. In *Proceedings of the Fourth SIAM International Conference on Data Mining, Lake Buena Vista, Florida, USA, April 22-24, 2004*, pages 442–446, 2004.

[31] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. Improving hash join performance through prefetching. In *Proceedings of the 20th International Conference on Data Engineering*, ICDE '04, pages 116–, Washington, DC, USA, 2004. IEEE Computer Society.

[32] Shimin Chen, Phillip B. Gibbons, and Todd C. Mowry. Improving index performance through prefetching. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, pages 235–246, New York, NY, USA, 2001. ACM.

[33] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 1–12, New York, NY, USA, 1999. ACM.

[34] Avery Ching, Sergey Edunov, Maja Kabiljo, Dionysios Logothetis, and Sambavi Muthukrishnan. One trillion edges: Graph processing at Facebook-scale. *Proceedings of the VLDB Endowment*, 8(12):1804–1815, 2015.

[35] Cloudera. Inside Cloudera Impala: Runtime Code Generation. `http://blog.cloudera.com/blog/2013/02/inside-cloudera-impala-runtime-code-generation/`, 2013.

[36] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Commun. ACM*, 47(3):45–47, March 2004.

[37] Jonathan Corbet. KAISER: hiding the kernel from user space. `https://lwn.net/Articles/738975/`, November 2017.

[38] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 3–3, Berkeley, CA, USA, 2003. USENIX Association.

[39] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[40] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1243–1254, New York, NY, USA, 2013. ACM.

[41] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *ACM SIGPLAN Notices*, volume 34, pages 229–241. ACM, 1999.

[42] Wei Ding, Mahmut Kandemir, Diana Guttman, Adwait Jog, Chita R. Das, and Praveen Yedlapalli. Trading cache hit rate for memory performance. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 357–368, New York, NY, USA, 2014. ACM.

[43] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Transactions on Architecture and Code Optimization (TACO)*, 2012.

[44] Tom Duff. Duff's Device. `http://doc.cat-v.org/bell_labs/duffs_device`, 1988.

[45] E5v3. Intel Xeon Processor E5-2680 v3(30M Cache, 2.50 GHz). `http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz`.

[46] Nosayba El-Sayed, Anurag Mukkara, Po-An Tsai, Harshad Kasture, Xiaosong Ma, and Daniel Sanchez. KPart: A hybrid cache partitioning-sharing technique for commodity multicores. In *Proceedings of the 24th international symposium on High Performance Computer Architecture (HPCA-24)*, February 2018.

[47] M. Anton Ertl and David Gregg. Optimizing indirect branch prediction accuracy in virtual machine interpreters. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 278–288, New York, NY, USA, 2003. ACM.

[48] C. W. Fletcher, L. Ren, X. Yu, M. Van Dijk, O. Khan, and S. Devadas. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 213–224, Feb 2014.

[49] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 212–223, New York, NY, USA, 1998. ACM.

[50] Rahul Garg and Yogish Sabharwal. Optimizing the HPCC RandomAccess benchmark on Blue Gene/L supercomputer. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '06/Performance '06, pages 369–370, New York, NY, USA, 2006. ACM.

[51] Vinodh Gopal, Wajdi Feghali, Jim Guilford, Erdinc Ozturk, Gil Wolrich, Martin Dixon, Max Locktyukhin, and Maxim Perminov. Fast cryptographic computation on Intel architecture processors via function stitching. `https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/communications-ia-cryptographic-paper.pdf`, 2010.

[52] Graph500. Graph 500 benchmark. `http://www.graph500.org/specifications`.

[53] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[54] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 279–299. Springer, 2016.

[55] Niklas Gustafsson, Deon Brewis, and Herb Sutter. Resumable functions. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3858.pdf`, 2014.

[56] Pablo Halpern, Arch Robison, Hong Hong, Artur Laksberg, Gor Nishanov, and Herb Sutter. Task block (formerly task region) r4. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4411.pdf`, 2015.

[57] Hwansoo Han and Chau-Wen Tseng. Improving compiler and run-time support for irregular reductions using local writes. In *Languages and Compilers for Parallel Computing*, pages 181–196. Springer, 1998.

[58] Hwansoo Han and Chau-Wen Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, July 2006.

[59] A. Herdrich, E. Verplanke, P. Autee, R. Illikkal, C. Gianos, R. Singhal, and R. Iyer. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 657–668, March 2016.

[60] Maurice Herlihy, Yossi Lev, Victor Luchangco, and Nir Shavit. A provably correct scalable concurrent skip list. In *Conference On Principles of Distributed Systems (OPODIS)*, 2006.

[61] C. Hermsmeyer, H. Song, R. Schlenk, R. Gemelli, and S. Bunse. Towards 100G packet processing: Challenges and technologies. *Bell Labs Technical Journal*, 14(2):57–79, Summer 2009.

[62] R. Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 66–77, New York, NY, USA, 1990. ACM.

[63] Jann Horn. Reading privileged memory with a side-channel. `https://googleprojectzero.blogspot.com/2018/01/`, January 2018.

[64] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: Providing memory performance feedback in modern processors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, ISCA '96, pages 260–270, New York, NY, USA, 1996. ACM.

[65] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual. `http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html`, 2017. [Online; accessed 11-February-2018].

[66] Intel. Intel Memory Latency Checker v3.4. `https://software.intel.com/en-us/articles/intelr-memory-latency-checker`, 2017.

[67] Intel. Speculative Execution Side Channel Mitigations. Revision 2.0. `https://software.intel.com/sites/default/files/managed/c5/63/336996-Speculative-Execution-Side-Channel-Mitigations.pdf`, May 2018.

[68] Intel. Deep dive: Intel Analysis of Microarchitectural Data Sampling. `https://software.intel.com/security-software-guidance/insights/deep-dive-intel-analysis-microarchitectural-data-sampling`, May 2019.

[69] Intel Corp. Improving real-time performance by utilizing Cache Allocation Technology. April 2015.

[70] Gorka Irazoqui, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Wait a minute! a fast, cross-VM attack on AES. In *International Workshop on Recent Advances in Intrusion Detection*, pages 299–319. Springer, 2014.

[71] Raj Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley & Sons, 1990.

[72] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer. High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 343–353, Feb 2015.

[73] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C Steely Jr, and Joel S. Emer. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware TLA cache management policies. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 151–162. IEEE Computer Society, 2010.

[74] Aamer Jaleel, William Hasenplaugh, Moinuddin Qureshi, Julien Sebot, Simon Steely, Jr., and Joel Emer. Adaptive insertion policies for managing shared caches. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 208–219, New York, NY, USA, 2008. ACM.

[75] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely Jr., and Joel S. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *37th International Symposium on Computer Architecture (ISCA 2010), June 19-23, 2010, Saint-Malo, France*, pages 60–71, 2010.

[76] JEDEC. DDR SDRAM Standard. `https://www.jedec.org/standards-documents/docs/jesd-79f`, 2000.

[77] JEDEC. DDR2 SDRAM Standard. `http://www.jedec.org/standards-documents/docs/jesd-79-2e`, 2003.

[78] JEDEC. DDR3 SDRAM Standard. `http://www.jedec.org/standards-documents/docs/jesd-79-3d`, 2007.

[79] JEDEC. DDR4 SDRAM Standard. `http://www.jedec.org/standards-documents/docs/jesd-79-4a`, 2014.

[80] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the "killer nanoseconds". *In Proceedings of the 44th International Conference on Very Large Data Bases (VLDB'18), August 2018, Rio de Janeiro, Brazil, VLDB Endowment*, 11(11):1702–1714, 2018.

[81] Sang-Woo Jun, Andy Wright, Sizhuo Zhang, Shuotao Xu, and Arvind. GraFboost: Using accelerated flash storage for external graph analytics. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 411–424, Piscataway, NJ, USA, 2018. IEEE Press.

[82] Wookeun Jung, Jongsoo Park, and Jaejin Lee. Versatile and scalable parallel histogram construction. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 127–138, New York, NY, USA, 2014. ACM.

[83] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 158–169. ACM, June 2015.

[84] M. Kayaalp, K. N. Khasawneh, H. A. Esfeden, J. Elwell, N. Abu-Ghazaleh, D. Ponomarev, and A. Jaleel. RIC: Relaxed inclusion caches for mitigating LLC side-channel attacks. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[85] Alfons Kemper, Thomas Neumann, Jan Finis, Florian Funke, Viktor Leis, Henrik Mühe, Tobias Mühlbauer, and Wolf Rödiger. Processing in the hybrid OLTP & OLAP main-memory database system hyper. *IEEE Data Eng. Bull.*, 36(2):41–47, 2013.

[86] Richard E Kessler and Mark D Hill. Page placement algorithms for large real-indexed caches. *Transactions on Computer Systems (TOCS)*, 1992.

[87] Paul-Virak Khuong and Pat Morin. Array layouts for comparison-based searching. *J. Exp. Algorithmics*, 22:1.3:1–1.3:39, May 2017.

[88] Changkyu Kim, Tim Kaldewey, Victor W. Lee, Eric Sedlar, Anthony D. Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endow.*, 2(2):1378–1389, August 2009.

[89] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '18, October 2018.

[90] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. DAWG: A defense against cache timing attacks in speculative execution processors. Cryptology ePrint Archive, Report 2018/418, May 2018.

[91] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *ArXiv e-prints*, July 2018.

[92] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. Cimple: Instruction and memory level parallelism. *ArXiv e-prints*, July 2018.

[93] Vladimir Kiriansky, Haoran Xu, Martin Rinard, and Saman Amarasinghe. Cimple: Instruction and memory level parallelism: A DSL for uncovering ILP and MLP. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 30:1–30:16, New York, NY, USA, 2018. ACM.

[94] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. Optimizing indirect memory references with milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 299–312, New York, NY, USA, 2016. ACM.

[95] Onur Kocberber, Babak Falsafi, and Boris Grot. Asynchronous memory access chaining. *Proc. VLDB Endow.*, 9(4):252–263, December 2015.

[96] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.

[97] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology (CRYPTO)*. Springer, 1996.

[98] Nicholas Kohout, Seungryul Choi, Dongkeun Kim, and Donald Yeung. Multi-chain prefetching: Effective exploitation of inter-chain memory parallelism for pointer-chasing codes. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, PACT '01, pages 268–279, Washington, DC, USA, 2001. IEEE Computer Society.

[99] Jingfei Kong, Onur Aciicmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *workshop on Computer security architectures*. ACM, 2008.

[100] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. Impala: A modern, open-source SQL engine for Hadoop. In *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*, 2015.

[101] D. Koufaty and D. T. Marr. Hyperthreading technology in the NetBurst microarchitecture. *IEEE Micro*, 23(2):56–65, March 2003.

[102] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture*, ISCA '81, pages 81–87, Los Alamitos, CA, USA, 1981. IEEE Computer Society Press.

[103] Jens Krueger, Changkyu Kim, Martin Grund, Nadathur Satish, David Schwalb, Jatin Chhugani, Hasso Plattner, Pradeep Dubey, and Alexander Zeier. Fast updates on read-optimized databases using multi-core CPUs. *Proc. VLDB Endow.*, 5(1):61–72, September 2011.

[104] Junjie Lai and Andre Seznec. Performance upper bound analysis and optimization of SGEMM on Fermi and Kepler GPUs. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[105] Butler Lampson. ACM Turing Award lectures. chapter Principles for Computer System Design. ACM, New York, NY, USA, 1993.

[106] Samuel Larsen and Saman Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 145–156, New York, NY, USA, 2000. ACM.

[107] Per-Åke Larson, Adrian Birka, Eric N. Hanson, Weiyun Huang, Michal Nowakiewicz, and Vassilis Papadimos. Real-time analytical processing with SQL server. *PVLDB*, 8(12):1740–1751, 2015.

[108] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. CGO '04, San Jose, CA, USA, Mar 2004.

[109] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012.

[110] Viktor Leis, Alfons Kemper, and Thomas Neumann. The adaptive radix tree: Artful indexing for main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 38–49, Washington, DC, USA, 2013. IEEE Computer Society.

[111] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51(3):244–257, 2010.

[112] Justin Levandoski, David Lomet, Sudipta Sengupta, Adrian Birka, and Cristian Diaconu. Indexing on modern hardware: Hekaton and beyond. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2014*. ACM, June 2014.

[113] Sheng Li, Hyeontaek Lim, Victor W. Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G. Andersen, O. Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to achieve a billion requests per second throughput on a single key-value store server platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[114] Fredrik Liljeros, Christofer R Edling, Luis A Nunes Amaral, H Eugene Stanley, and Yvonne Åberg. The web of human sexual contacts. *Nature*, 411(6840):907–908, 2001.

[115] 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 *HPCA*. IEEE, 2008.

[116] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *ArXiv e-prints*, January 2018.

[117] John D. C. Little. A proof for the queuing formula: L = λ W. *Oper. Res.*, 9(3):383–387, June 1961.

[118] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. CATalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, Mar 2016.

[119] Fangfei Liu and Ruby B Lee. Random fill cache architecture. In *Microarchitecture (MICRO)*. IEEE, 2014.

[120] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *Security and Privacy*. IEEE, 2015.

[121] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M. Hellerstein. Distributed GraphLab: A framework for machine learning and data mining in the cloud. *Proc. VLDB Endow.*, 5(8):716–727, April 2012.

[122] Stefan Manegold, Peter Boncz, and Martin Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. on Knowl. and Data Eng.*, 14(4):709–730, July 2002.

[123] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11:1–13, September 2017.

[124] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *J. Algorithms*, 49(1):114–152, October 2003.

[125] Nicholas Mitchell, Larry Carter, and Jeanne Ferrante. Localizing non-affine array references. In *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pages 192–202. IEEE, 1999.

[126] Chuck Moore and Pat Conway. General-purpose multi-core processors. In Stephen W. Keckler, Kunle Olukotun, and H. Peter Hofstee, editors, *Multicore Processors and Systems*, pages 173–203. Springer US, Boston, MA, 2009.

[127] Todd C. Mowry, Monica S. Lam, and Anoop Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, pages 62–73, New York, NY, USA, 1992. ACM.

[128] Anurag Mukkara, Nathan Beckmann, and Daniel Sanchez. Whirlpool: Improving dynamic cache management with static data classification. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 113–127, New York, NY, USA, 2016. ACM.

[129] Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '15, pages 291–305, Berkeley, CA, USA, 2015. USENIX Association.

[130] Nervana. SGEMM. https://github.com/NervanaSystems/maxas/wiki/SGEMM, 2017.

[131] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[132] A. Newell and H. Simon. The logic theory machine–a complex information processing system. *IRE Transactions on Information Theory*, 2(3):61–79, September 1956.

[133] A. Newell and F. M. Tonge. An introduction to information processing language v. *Commun. ACM*, 3(4):205–211, April 1960.

[134] M. E. J. Newman. Detecting community structure in networks. *The European Physical Journal B - Condensed Matter and Complex Systems*, 38(2):321–330, 2004.

[135] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. A lightweight infrastructure for graph analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 456–471, New York, NY, USA, 2013. ACM.

[136] Gor Nishanov and Jim Radigan. Resumable functions v.2. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4134.pdf, 2014.

[137] OpenMP. OpenMP Application Program Interface 4.5. http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf, 2015.

[138] Yossef Oren, Vasileios P Kemerlis, Simha Sethumadhavan, and Angelos D Keromytis. The spy in the sandbox – practical cache attacks in Javascript. *arXiv preprint arXiv:1502.07373*, 2015.

[139] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *Topics in Cryptology–CT-RSA 2006*, pages 1–20. Springer, 2006.

[140] G. Ottoni and B. Maher. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 233–244, Feb 2017.

[141] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. Automatic thread extraction with decoupled software pipelining. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 105–118, Washington, DC, USA, 2005. IEEE Computer Society.

[142] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999.

[143] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. *SIGARCH Comput. Archit. News*, 12(3):348–354, January 1984.

[144] I.E. Papazian, S. Kottapalli, J. Baxter, J. Chamberlain, G. Vedaraman, and B. Morris. Ivy Bridge server: A converged design. *Micro, IEEE*, 35(2):16–25, Mar 2015.

[145] Binh Pham, Ján Veselý, Gabriel H. Loh, and Abhishek Bhattacharjee. Large pages and lightweight memory management in virtualized environments: Can you have it both ways? In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 1–12, New York, NY, USA, 2015. ACM.

[146] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[147] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: a practical approach for robust index joins. *In Proceedings of the 44th International Conference on Very Large Data Bases (VLDB'18), August 2018, Rio de Janeiro, Brazil, VLDB Endowment*, 11(2):230–242, 2017.

[148] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *The VLDB Journal*, Dec 2018.

[149] Christian Queinnec and Bernard Serpette. A dynamic extent control operator for partial continuations. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 174–184, New York, NY, USA, 1991. ACM.

[150] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: demand-based associativity via global replacement. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 544–555, June 2005.

[151] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 381–391, New York, NY, USA, 2007. ACM.

[152] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[153] Easwaran Raman, Guilherme Ottoni, Arun Raman, Matthew J. Bridges, and David I. August. Parallel-stage decoupled software pipelining. In *Proceedings of the 6th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, pages 114–123, New York, NY, USA, 2008. ACM.

[154] Jun Rao and Kenneth A. Ross. Making B+-Trees cache conscious in main memory. *SIGMOD Rec.*, 29(2):475–486, May 2000.

[155] Richard Grisenthwaite. Cache Speculation Side-channels, January 2018.

[156] RocksDB. RocksDB. `http://rocksdb.org/`, 2017.

[157] Erven Rohou, Bharath Narasimha Swamy, and André Seznec. Branch prediction and the performance of interpreters: Don't trust folklore. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 103–114, Washington, DC, USA, 2015. IEEE Computer Society.

[158] Joel Saltz, Kathleen Crowley, Ravi Michandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8(4):303–312, 1990.

[159] Samsung. DDR3L SDRAM 240pin Registered DIMM M393B2G70BH0 datasheet:. `http://www.samsung.com/global/business/semiconductor/file/product/ds_ddr3_4gb_b-die_based_1_35v_rdimm_rev16-5.pdf`, 2012.

[160] Samsung. DDR4 SDRAM 288pin Registered DIMM M393A2G40DB1 datasheet. `http://www.samsung.com/semiconductor/global/file/product/DS_8GB_DDR4_4Gb_D_die_RegisteredDIMM_Rev15.pdf`, 2015.

[161] Daniel Sanchez and Christos Kozyrakis. Vantage: Scalable and efficient fine-grain cache partitioning. In *38th Annual International Symposium on Computer Architecture (ISCA)*, pages 57–68, June 2011.

[162] Daniel Sanchez and Christos Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture-ISCA*, volume 13, pages 23–27. Association for Computing Machinery, 2013.

[163] Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Jiwon Seo, Jongsoo Park, M. Amber Hassaan, Shubho Sengupta, Zhaoming Yin, and Pradeep Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 979–990, New York, NY, USA, 2014. ACM.

[164] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[165] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. PALM: parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *PVLDB*, 4(11):795–806, 2011.

[166] Stephen Shankland. Itanium: A cautionary tale. *CNET News*, Dec 2005. [Online; accessed 11-February-2015].

[167] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[168] Rami Sheikh, James Tuck, and Eric Rotenberg. Control-Flow Decoupling. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, pages 329–340, Washington, DC, USA, 2012. IEEE Computer Society.

[169] Yossi Shiloach and Uzi Vishkin. An O(log n) parallel connectivity algorithm. *Journal of Algorithms*, 3(1), 1982.

[170] Julian Shun and Guy E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 135–146, New York, NY, USA, 2013. ACM.

[171] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1 - the fault-tolerant distributed RDBMS supporting Google's ad business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.

[172] B. Sinharoy, R. Kalla, W. J. Starke, H. Q. Le, R. Cargnoni, J. A. Van Norstrand, B. J. Ronchetti, J. Stuecheli, J. Leenstra, G. L. Guthrie, D. Q. Nguyen, B. Blaner, C. F. Marino, E. Retter, and P. Williams. IBM POWER7 multicore server processor. *IBM J. Res. Dev.*, 55(3):191–219, May 2011.

[173] B J Smith. Advanced computer architecture. chapter A Pipelined, Shared Resource MIMD Computer, pages 39–41. IEEE Computer Society Press, Los Alamitos, CA, USA, 1986.

[174] Read Sprabery, Konstantin Evchenko, Abhilash Raj, Rakesh B. Bobba, Sibin Mohan, and Roy H. Campbell. A novel scheduling framework leveraging hardware cache partitioning for cache-side-channel elimination in clouds. *CoRR*, abs/1708.09538, 2017.

[175] Stefan Sprenger, Steffen Zeuch, and Ulf Leser. Cache-sensitive skip list: Efficient range queries on modern cpus. In *International Workshop on In-Memory Data Management and Analytics*, pages 1–17. Springer, 2016.

[176] Jimmy Su and Katherine Yelick. Automatic support for irregular computations in a high-level language. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Papers - Volume 01*, IPDPS '05, Washington, DC, USA, 2005. IEEE Computer Society.

[177] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proceedings of the VLDB Endowment*, 8(3):245–256, 2014.

[178] George Taylor, Peter Davies, and Michael Farmwald. The TLB slice - a low-cost high-speed address translation mechanism. *SIGARCH Computer Architecture News*, 1990.

[179] Jens Teubner, Gustavo Alonso, Cagri Balkesen, and M. Tamer Ozsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering*, ICDE '13, pages 362–373, Washington, DC, USA, 2013. IEEE Computer Society.

[180] Linus Torvalds. Re: Page colouring. `http://yarchive.net/comp/linux/cache_coloring.html`, 2003.

[181] K. A. Tran, T. E. Carlson, K. Koukos, M. Själander, V. Spiliopoulos, S. Kaxiras, and A. Jimborean. Clairvoyance: Look-ahead compile-time scheduling. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 171–184, Feb 2017.

[182] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[183] James Tuck, Luis Ceze, and Josep Torrellas. Scalable cache miss handling for high memory-level parallelism. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, pages 409–422, Washington, DC, USA, 2006. IEEE Computer Society.

[184] Paul Turner. Retpoline: a software construct for preventing branch-target-injection. `https://support.google.com/faqs/answer/7625886`, January 2018.

[185] Neil Vachharajani, Ram Rangan, Easwaran Raman, Matthew J. Bridges, Guilherme Ottoni, and David I. August. Speculative decoupled software pipelining. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 49–59, Washington, DC, USA, 2007. IEEE Computer Society.

[186] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[187] Vasily Volkov. Better performance at lower occupancy. In *Proceedings of the GPU technology conference, GTC'10*, 2010.

[188] Carl A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the 5th Symposium on Operating Systems Design and implementation-Copyright Restrictions Prevent ACM from Being Able to Make the PDFs for This Conference Available for Downloading*, OSDI '02, pages 181–194, Berkeley, CA, USA, 2002. USENIX Association.

[189] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 74:1–74:6, New York, NY, USA, 2016. ACM.

[190] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[191] Jan Wassenberg and Peter Sanders. Engineering a multi-core radix sort. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 160–169, Berlin, Heidelberg, 2011. Springer-Verlag.

[192] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. The RISC-V instruction set manual, volume i: User-level ISA, version 2.0. Technical Report UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[193] Q. Xu, H. Jeon, and M. Annavaram. Graph processing on GPUs: Where are the bottlenecks? In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 140–149, Oct 2014.

[194] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy*, pages 640–656, May 2015.

[195] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (sharp): Defending against cache-based side channel atacks. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 347–360, New York, NY, USA, 2017. ACM.

[196] F. Yao, M. Doroslovacki, and G. Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 168–179, Feb 2018.

[197] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[198] A. Yasin, Y. Ben-Asher, and A. Mendelson. Deep-dive analysis of the data analytics workload in cloudsuite. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 202–211, Oct 2014.

[199] K.C. Yeager. The MIPS R10000 Superscalar Microprocessor. *IEEE Micro*, 1996.

[200] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache Spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[201] M. Zhang and K. Asanovic. Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 336–345, June 2005.

[202] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, pages 89–102, New York, NY, USA, 2009. ACM.

[203] Y. Zhang, V. Kiriansky, C. Mendis, S. Amarasinghe, and M. Zaharia. Making caches work for graph analytics. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 293–302, Dec 2017.