Extending Memory System Semantics to Accelerate Irregular Applications

by

Guowei Zhang

B.S. in Microelectronics and B.S. in Economics
Tsinghua University (2014)

M.S. in Electrical Engineering and Computer Science
Massachusetts Institute of Technology (2016)

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

Doctor of Philosophy in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2021

Author.....
Department of Electrical Engineering and Computer Science
December 28, 2020

Certified by.....
Daniel Sanchez
Associate 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

Extending Memory System Semantics to Accelerate Irregular Applications

by

Guowei Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on December 28, 2020, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy in Electrical Engineering and Computer Science

Abstract

Computer systems are increasingly bottlenecked by data movement, and rely on sophisticated memory hierarchies to address this issue. However, conventional memory systems suffer from poor performance on many irregular access patterns. This is because memory systems use an inexpressive interface that does not convey sufficient program semantics: they organize data in fixed-sized chunks and access data with only reads and writes. As a result, memory systems incur significant performance loss on several common patterns. In this thesis, we identify three such patterns: accesses to small data fragments suffer poor locality; concurrent updates introduce excessive traffic and serialization; and dependent reads incur long latencies that are on the critical path.

To tackle these issues, this thesis proposes techniques that extend the semantics of the memory system. We apply this insight to address each of the three issues and propose solutions with different degrees of generality. COUP and COMMTM provide general architectural support by exploiting commutative updates to reduce communication and synchronization. COUP supports strict single-instruction commutativity by extending the cache coherence protocol, while COMMTM supports multi-instruction and semantic commutativity by leveraging hardware transactional memory. Whereas COUP and COMMTM are general, HTA and GAMMA target a specific data structure and a specific application, respectively. HTA addresses the inefficiencies of small fragments in the context of hash tables. It exploits the associativity in hash tables and leverages caches to reduce runtime overheads and to improve spatial locality. GAMMA is a sparse matrix-matrix multiplication accelerator. Its novel storage idiom, FIBERCACHE, combines caching and decoupled execution to ensure low latency for dependent reads with irregular reuse. This enables GAMMA to adopt an efficient dataflow, Gustavson's algorithm, to minimize off-chip traffic. In return, these techniques improve the performance and reduce the data movement of challenging applications significantly.

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

Contents

Acknowledgments

First and foremost, I owe thanks to my research advisor, Professor Daniel Sanchez. I cannot imagine how this thesis would have happened without his guidance. For me, Daniel is a perfect supervisor. It always amazes me how incredibly knowledgeable and experienced he is. Daniel guided me to the field of computer architecture, and taught me patiently how to conduct research. I learned from him how to think of big picture challenges, formulate research problems, and validate ideas with carefully conducted analysis and experiments as quickly as possible. Daniel provided me enormous freedom, yet he has always been available when I needed his help. In fact, Daniel has been supportive of me no matter what happened, which I shall never forget.

I would like to thank my thesis committee members, Professor Joel Emer and Professor Mengjia Yan. It was my privilege to collaborate with Joel in the GAMMA project. I have benefited so much from his warm and empathetic mentorship and his principled approach to conduct research. Joel taught me the importance of naming: a consistent terminology for diverse research problems and solutions is invaluable as it forces and enables reasoning about the core insights rather than the superficial implementations. Though I have not collaborated with Mengjia on a research project, she has provided me with valuable support and insightful feedback on both my thesis and my teaching experience in the course 6.823.

I am especially grateful to my great co-authors, Webb Horn, Virginia Chiu, and Nithya Attaluri. I have benefited so much from the enjoyable and memorable collaborations with them.

I owe thanks to many other professors at MIT. Professor David Perreault has been my academic advisor throughout the years. He always made sure that I was on the right path to complete a successful thesis. I thank Professors Arvind, Srini Devadas, and Vivienne Sze for their valuable feedback on my research. Additionally, I thank Professors Michael Sipser, Gregory Wornell, Stefanie Jegelka, and Samuel Madden for their excellent lectures as part of my degree.

I also would like to thank the administrative staff, Sally Lee and Mary McDavitt from system groups and Janet Fischer and Alicia Duarte from EECS administration who helped me to complete my PhD.

I am thankful to the members of Sanchez group: Maleen Abeydeera, Nathan Beckmann, Nosayba El-Sayed, Axel Feldmann, Yee Ling Gan, Mark Jeffrey, Harshad Kasture, Hyun Ryong Lee, Anurag Mukkara, Quan Nguyen, Nikola Samardzic, Suvinay Subramanian, Po-An Tsai, Cong Yan, Yifan Yang, and Victor Ying. It is my great pleasure to receive their valuable feedback on my research and discuss all kinds of fun stuff. I truly enjoy interacting with them no matter when, where, and how: during regular research meetings, random discussions in offices, travels to conferences, group outing activities, Thanksgiving dinners, or virtual Friday game nights.

I appreciate my friends in both China and the United States for making my life outside research exciting and wonderful.

Last but not least, I would love to thank my family, especially my mother and my wife. Though we are often separated by a 12- or 13-hour time difference, they have provided me the best love and encouragement I could ever imagine. I could not have completed this thesis without their support.

I am grateful for financial support from an MIT EECS Grier Fellowship; C-FAR, one of six SRC STARnet centers by MARCO and DARPA; NSF under grants CAREER-1452994 and SHF-1814969; DARPA SDH under contract HR0011-18-3-0007; and research grants from IBM and SRC. This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government.

Introduction 1

Computer systems are increasingly bottlenecked by *data movement*. For instance, in a chip manufactured at 28 nm technology, a 64-bit floating-point multiply-add, a complex arithmetic operation, consumes 20 pJ. However, shipping its operands across the chip consumes 1 nJ, 50× higher [72, 131, 231]. Shipping the operands from off-chip DRAM is even worse, and can require 16 nJ, 800× more expensive than the arithmetic operation. The gap between the cost of data movement and computation has a growing impact as the system size increases, making memory system design increasingly critical.

One key approach to reduce data movement is to exploit *temporal* and *spatial locality* in the memory system. To do this, most memory systems use a hierarchy of caches. Caches adopt sophisticated hardware mechanisms such as the cache replacement policy to exploit locality dynamically without requiring accesses to be known ahead. Such generality ensures its central role in conventional memory systems.

However, there is a large semantic gap between conventional memory systems and applications. This gap often causes a significant performance penalty. Specifically, conventional memory systems typically organize data in *fixed-sized chunks*. Examples include lines in caches and pages in virtual memory. In contrast, applications use much more diverse data structures, such as trees and hash tables. Furthermore, conventional memory systems typically support only two primitive operations: `read` and `write`. This is not the case for applications: they can access data with various other operations such as `push`, `enqueue`, and `insert`. While this narrow interface is general enough and allows for a simple implementation, this interface is overly restrictive and hurts performance on many important access patterns.

Challenges 1.1

We identify three patterns that can cause conventional systems to suffer poor performance: *small fragments*, *concurrent updates*, and *dependent reads*.

Small Fragments 1.1.1

Accesses to small fragments incur poor spatial locality and can waste significant capacity. This is caused by the fixed-sized data representation in a conventional memory system. For instance, caches organize data in fixed-sized lines, and expect all the bits in a line to be accessed at least once before an eviction. However, a small fragment typically will not occupy a full cache line. Therefore, a significant portion of the cache capacity can be wasted.

Typical examples of small fragments include the key-value pairs in a hash table and the nodes in a tree. Such pairs and nodes consist of only a few words, much smaller than a typical cache line (e.g., 64 B). Such fragments are usually accessed in a data-dependent fashion. This is especially true for the pairs in a hash table: to reduce mapping conflicts, hashing intentionally spreads the pairs uniformly across the hash table's allocated memory.

1.1.2 Concurrent Updates

Concurrent updates to shared data cause expensive communication and serialization in a conventional memory system. This happens because of the *exclusive* nature of a write. Specifically, in a conventional memory system, an update must be expressed as a *read-modify-write* sequence. While reads are *non-exclusive*, i.e., they can access the same data concurrently without communication or synchronization, writes are *exclusive*: concurrent writes must communicate and synchronize with each other to ensure serializability.

For instance, consider a counter that is updated by multiple cores in a shared memory system. On each update, the updating core first fetches an exclusive copy of the counter's cache line into its private cache, invalidating all other copies, and modifies it locally using an atomic operation such as fetch-and-add, Each update incurs significant *traffic* and *serialization*: traffic to fetch the line and invalidate other copies, causing the line to ping-pong among updating cores; and serialization because only one core can perform an update at a time.

1.1.3 Dependent Reads

Dependent reads introduce long access latencies on the critical path, limiting system performance significantly. This is because a read introduces a *roundtrip* of a request and a response. In a conventional cache, such a roundtrip is typically coupled with the long-latency data movement between the cache and the main memory. When such reads are dependent upon each other, i.e., the address of a read is computed from the return value of a previous read, e.g., A[B[i]], long roundtrip latencies accumulate, bottlenecking system performance severely.

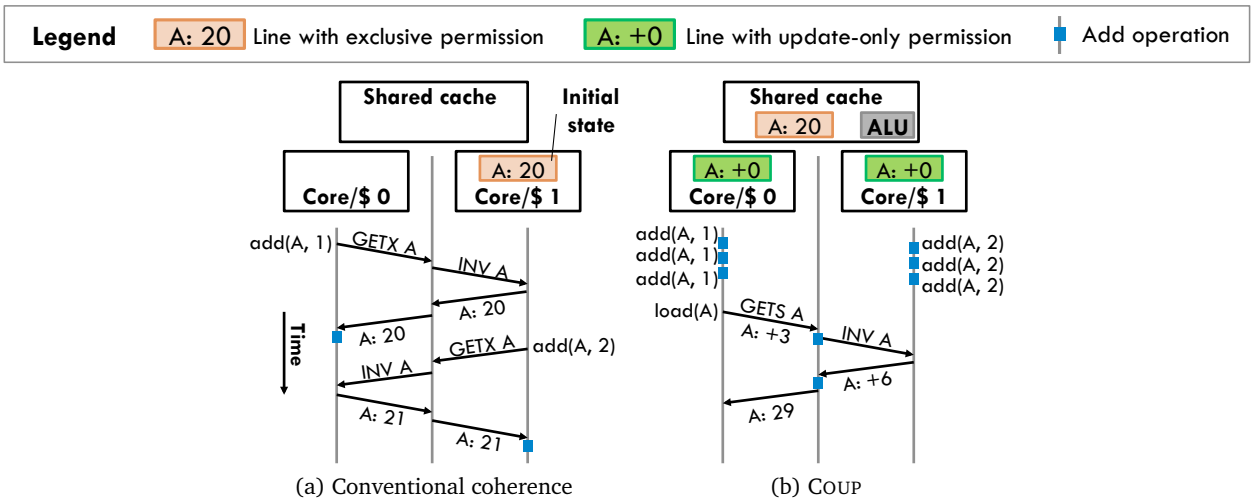
One example is accessing a compressed sparse matrix. Such an operation typically requires indirections in the form of A[B[i]]. If both the accesses to B[i] and A[B[i]] miss in the cache, the system can incur a long critical-path latency that requires expensive mechanisms to hide.

Conventional memory systems were not designed or optimized for these access patterns. While prior work has proposed techniques to address them, these techniques work for only *regular applications*, where accesses are known ahead of time. For example, in a regular application like Dense Matrix-Matrix Multiplication (MM), tiling improves spatial locality, software-based variable privatization avoids concurrent updates, and prefetching hides the long latency incurred by reads.

However, *irregular applications*, i.e., applications where accesses are not known ahead, cannot exploit these techniques. Sparse matrix-matrix multiplication (SPMSPM) is one example. SPMSPM may access small fragments with dependent reads frequently to traverse matrices, and may issue concurrent updates to produce the output. Techniques for regular applications do not work well for irregular applications because of the lack of access knowledge. What's even worse is that such irregular applications are growing rapidly in many crucial domains, such as databases [144], graph analytics [93, 136, 160, 210], bioinformatics [175], and deep learning [105, 199, 267]. Therefore, it is crucial to design architectural support to address these patterns in irregular applications.

1.2 Contributions

The key insight that this thesis develops is that the memory system can be extended with rich semantics to address the aforementioned *irregular* patterns. In this way, the memory system no longer reasons in terms of only reads and writes. Instead, by codesigning hardware and software, activities in the memory system are expressed more explicitly. This thesis applies this insight to address each of the



(a) Conventional coherence (b) COUP

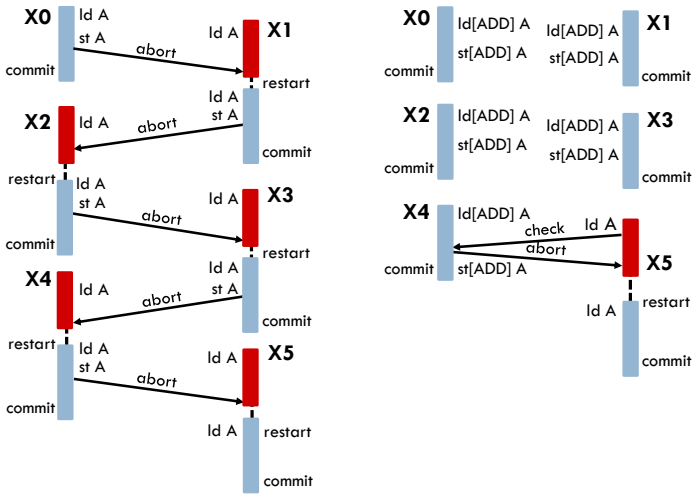
Figure 1.1: Example comparing the cost of commutative updates under two schemes. Two cores add values to a single memory location, A. (a) With conventional coherence protocols, A's fetches and invalidations dominate the cost of updates. (b) With COUP, caches buffer and coalesce updates locally, and reads trigger a reduction of all local updates to produce the actual value.

three challenges mentioned above, and proposes solutions tailored to various degrees of generality. For instance, we propose general architectural support for concurrent updates, but for dependent reads, we narrow the scope down to a specific application, SPMSPM, and build a customized memory system for it in an accelerator:

- **COUP** and **COMMTM** target the issues caused by *concurrent updates*. The key insight they exploit is that many read-modify-write updates are *commutative*. Commutative operations produce the same final result regardless of the order they are performed in. These operations need not read the data they update, so they do not introduce true data dependencies and therefore can be processed concurrently and coalesced locally before the data is read. Hence, updates can proceed concurrently, *without communication or synchronization*. COUP extends the cache coherence protocol so that single-instruction commutative updates can be performed locally and concurrently. COMMTM further exploits multi-instruction commutativity by leveraging hardware transactional memory.
- **HTA** mainly addresses the problems caused by *small fragments*. The key insight is that many such fragments are parts of an *unordered associative* container, i.e., a hash table, and can be optimized through hardware-software codesign. HTA adopts a table format that leverages characteristics such as cache line sizes and boundaries to avoid accessing disjoint small fragments. This new interface allows HTA to accelerate most operations with simple hardware, and to leave corner cases to a software path.
- **GAMMA** targets the issues of *dependent reads* in the context of SPMSPM. We show that Gustavson's algorithm is an efficient dataflow for SPMSPM, but it introduces dependent reads with irregular reuse that make conventional designs expensive. The key insight is that caching and decoupled execution can be combined: decoupled execution hides the long latency of dependent reads, while caching captures the irregular reuse patterns. GAMMA exploits this insight with a new storage idiom, FIBERCACHE. In return, GAMMA outperforms state-of-the-art SPMSPM accelerators significantly.

COUP 1.2.1

COUP (Chapter 3) is a general technique that extends coherence protocols to allow local and concurrent commutative updates. Specifically, COUP decouples read and write permissions, and introduces a limited



(a) Conventional HTM (b) COMMTM

Figure 1.2: Example comparing (a) a conventional HTM and (b) COMMTM. Transactions X0–X4 increment a shared counter, and X5 reads it. While conventional HTMs serialize all transactions, COMMTM allows commutative operations (additions in X0–X4) to happen concurrently, serializing non-commutative operations (the load in X5) only.

number of commutative-update primitive operations, in addition to reads and writes. With COUP, multiple caches can acquire a line with *update-only* permission, and satisfy commutative-update requests locally, buffering and coalescing updates. On a read request, the coherence protocol gathers all the local updates and *reduces* them to produce the correct value before granting read permission. For example, multiple cores can concurrently add values to the same counter. Unlike conventional systems that suffer from excessive traffic and serialization shown in Figure 1.1a, COUP reduces communication and synchronization by exploiting commutativity. Updates are held in their private caches as long as no core reads the current value of the counter. When a core reads the counter, all updates are added to produce the final value, as shown in Figure 1.1b.

COUP provides significant benefits at minimal cost. COUP introduces minor hardware overheads, preserves coherence and consistency, and imposes small verification costs. We identify several update-heavy parallel applications where current techniques have clear shortcomings, and discuss how COUP addresses them. Specifically, COUP supports a limited number of *single-instruction* commutative operations, such as addition and bitwise logical operations. We evaluate COUP under simulation, using single- and multi-socket systems. At 128 cores, COUP improves the performance of update-heavy benchmarks by 4%–2.4× and reduces traffic by up to 20×.

1.2.2 COMMTM

COMMTM (Chapter 4) is a Commutativity-aware Hardware Transactional Memory (HTM) that extends COUP to support an unlimited number of user-defined commutative operations that cannot be expressed as single instructions. Specifically, COMMTM supports a much broader range of *multi-instruction, semantically commutative* operations, such as set insertions and ordered puts. We find that commutativity and transactional memory are complementary: transactions benefit commutativity by guaranteeing the atomicity of multi-instruction operations, and commutativity benefits transactions by avoiding unnecessary conflicts and wasted work, as shown in Figure 1.2.

COMMTM extends the coherence protocol with a *reducible* state that generalizes COUP's update-only state. Lines in this state must be tagged with a user-defined *label*. Multiple caches can hold a given line in the reducible state with the same label, and transactions can implement commutative operations through labeled loads and stores that keep the line in the reducible state. These commutative operations proceed concurrently, without triggering conflicts or incurring any communication. A non-commutative operation (e.g., a conventional load or store) triggers a user-defined reduction that merges the different cache lines and may abort transactions with outstanding reducible updates.

We explore several variants of COMMTM that trade precision for hardware complexity. We first present a basic version of COMMTM that achieves the same precision as software *semantic locking* [149, 265]. We then extend COMMTM with *gather requests*, which allow software to distribute reducible data among caches, achieving much higher concurrency in important use cases.

We evaluate COMMTM with microbenchmarks and full TM applications. Microbenchmarks show that COMMTM scales on a variety of commutative operations, such as set insertions, reference counting, ordered puts, and top-K insertions, which allow no concurrency in conventional HTMs. At 128 cores, COMMTM improves full-application performance by up to 3.4×, lowers private cache misses by up to 45%, and reduces or even eliminates transaction aborts.

HTA 1.2.3

HTA (Chapter 5) is a Hash Table Acceleration technique that addresses the inefficiencies of hash table operations. Hash tables are widely used and consume the majority of cycles on key applications in databases [144] and genomics [175]. While hash tables have been extensively studied and optimized in software, they leave significant performance on the table in conventional systems.

Specifically, hash tables suffer from two key inefficiencies in conventional systems:

- **Poor core utilization:** Each hash table operation consists of a long sequence of instructions to compute hash values, memory accesses to keys and values, and comparisons. These instructions include hard-to-predict, data-dependent branches that add wasted cycles, and incur long-latency cache misses that limit instruction-level parallelism.
- **Poor spatial locality:** To reduce mapping conflicts, hashing spreads key-value pairs uniformly across the hash table's allocated memory. This causes poor spatial locality when key-value pairs have mixed reuse, e.g., the same-line neighbors of a frequently accessed pair are rarely accessed. This wastes a significant portion of cache capacity.

HTA address the issues through the combination of expressive ISA extensions and simple hardware changes. HTA adopts a hash table format that leverages the associative nature of caches. With this new format, multiple probes in a single lookup need access only one whole line instead of multiple small fragments spread across multiple lines. HTA accelerates most operations in hardware, and leaves rare cases to software.

We present two implementations of HTA, FLAT-HTA and HIERARCHICAL-HTA. FLAT-HTA adopts a simple, hierarchy-oblivious layout and reduces runtime overheads with simple changes to cores. HIERARCHICAL-HTA is a more complex implementation that uses a hierarchy-aware layout to improve spatial locality at intermediate cache levels. It requires some changes to caches and provides modest benefits over FLAT-HTA.

We evaluate HTA on hash table-intensive benchmarks and use it to accelerate *memoization*, a technique that caches and reuses the outputs of repetitive computations. FLAT-HTA improves the performance of the state-of-the-art hash table-intensive applications by up to 2×, while HIERARCHICAL-HTA outperforms FLAT-HTA by up to 35%. FLAT-HTA also outperforms software memoization by 2×.

1.2.4 GAMMA

GAMMA is a Gustavson-Algorithm Matrix-Multiplication Accelerator. It features architectural support for dependent reads to enable an efficient *dataflow* of SPMSPM, i.e., Gustavson's dataflow, in accelerators.

Specifically, SPMSPM is a key kernel that lies at the heart of many sparse algorithms [39, 105, 136, 140, 199, 274]. First, SPMSPM is bottlenecked by memory traffic and data movement, and admits three *dataflows* (i.e., computation schedules) with different tradeoffs: *inner-product* [109, 207], *outer-product* [196, 291], and *Gustavson's algorithm* [99]. Though Gustavson often achieves the least amount of memory traffic and requires simpler operations, prior accelerators exploit either inner-product or outer-product dataflows. This is because Gustavson has more *irregular reuse* across data structures, demanding a storage organization that can accommodate them to effectively reduce memory traffic.

GAMMA addresses this issue. GAMMA performs SPMSPM's computation using specialized processing elements (PE) with simple high-radix mergers, and performs many merges in parallel to achieve high throughput. GAMMA uses a novel on-chip storage structure that combines features of both caches and explicitly managed buffers. This structure captures Gustavson's irregular reuse patterns and streams thousands of concurrent sparse *fibers*, i.e., variable-sized rows from inputs or partial outputs, with explicitly decoupled data movement. FIBERCACHE fetches the needed fibers ahead of time so that when the processing element reads each input fiber element, the data is served from the FIBERCACHE without incurring memory accesses. This avoids PE stalls and lets the FIBERCACHE pull double duty as a latency-decoupling buffer, and saves megabytes of dedicated on-chip buffers. GAMMA features a new dynamic scheduling algorithm to achieve high utilization despite irregularity. We also present new preprocessing algorithms that boost GAMMA's efficiency and versatility.

We synthesize GAMMA and evaluate its performance on a wide range of sparse matrices. With a similar hardware budget, compared to state-of-the-art accelerators, GAMMA reduces total DRAM traffic by 2.2× on average, non-compulsory DRAM traffic by 12× on average, and achieves nearly full DRAM bandwidth utilization. Moreover, GAMMA is effective on a much broader range of sparse matrices.

In summary, the goal of this thesis is to address the patterns that used to be challenging for conventional memory systems: concurrent updates, small fragments, and dependent reads. By codesigning hardware and software, the memory system can be extended with high-level semantics. COUP and COMMTM are general support for concurrent updates, by exploiting operation-level commutativity. HTA is tailored to leverage the knowledge of a data structure, hash table, while GAMMA is tailored to an important application, SPMSPM. In return, the techniques proposed by this thesis improve performance significantly.

Background 2

Modern memory systems should allow fast (e.g, nanoseconds) accesses to large (e.g., gigabytes) volumes of data. But this is hard to achieve with a single storage structure: storage structures with large capacity incur high access latencies, and fast storage structures cannot be made sufficiently large.

Therefore, memory systems combine fast/small and slow/large storage structures in a hierarchy. The off-chip main memory made of DRAM is the root of the tree: it typically has gigabytes of capacity with an access latency of hundreds of nanoseconds. On-chip storage components are the internal nodes and the leaves of the tree. They are typically made of kilobytes to megabytes of SRAM, with access latencies ranging from one to tens of nanoseconds. As a result, most accesses are fulfilled by the fast/small on-chip structures, while others are served by the off-chip main memory.

On-chip storage can be implemented as various structures with tradeoffs in their interfaces, overheads, and capabilities. Section 2.1 presents various storage structures, and then Section 2.2 summarizes prior software and hardware techniques proposed to improve the performance of memory systems.

Memory System Architecture 2.1

Caches 2.1.1

A cache is a storage structure that receives accesses, i.e., reads and writes, from processors, and *transparently* retains recently accessed data. Subsequent accesses to data present in the cache (hits) are served directly by the cache; otherwise (misses), the cache accesses the next level of the hierarchy. Caches are widely adopted in CPUs, GPUs, and some accelerators to reduce access latency and save memory bandwidth.

Transparency: A cache can be made architecturally invisible while retaining good performance in most cases at a reasonable cost. Such transparency to software is a critical property for an on-chip structure to be widely adopted, especially in general-purpose CPUs and GPUs, as it simplifies programming and enhances compatibility significantly. To achieve this, caches support only *global accesses*, i.e., accesses based on global addresses rather than structure-specific locations, and require additional synchronization mechanisms to provide a *shared-memory* illusion when there are multiple private caches caching the same data.

Supporting global accesses: A cache maps multiple memory locations to the same cache location, and introduces a *tag* for each cache line slot to differentiate them. Such a mapping may have different levels of flexibility. A *direct-mapped* cache allows no flexibility as it maps each memory location to only one cache location. In contrast, a *fully-associative* cache enjoys the most flexibility as a line in memory may

map to all locations. An *N-way set-associative* cache sits in the middle, as it allows a line in memory to map to a *set* of *N* locations. Higher associativity leads to fewer conflict misses but higher area and energy costs.

Such flexibility of mapping in fully-associative and *N*-way set-associative caches introduces a decision problem: a cache must determine which line to evict when there is not enough capacity for a new line. Because cache accesses typically do not control evictions explicitly [1], such decisions are made in hardware by a *cache replacement policy*. The optimal replacement policy is MIN [28, 171]. But it is impractical to implement as it requires knowing future accesses. Practical replacement policies use heuristics, e.g., access history, to approximate MIN. Therefore, they do not require knowing accesses ahead, and are effective as long as the accesses exhibit good locality. Common policies include LRU, DIP [208], PDP [83], SHiP [271], and RRIP [121].

Supporting the shared-memory model: In a multicore system, it is common to have multiple private caches, each attached to a different core to improve performance. This poses challenges on providing a shared-memory programming model as these caches must synchronize on writes to shared data transparently.

Cache coherence is a property of a cache hierarchy that makes private caches invisible to applications. Coherence can be expressed as two invariants [68]: *write propagation*, that writes eventually become visible to all cores, and *write serialization*, that writes to the same address are observed in the same order by all processors.

Coherence is enforced by the *cache coherence protocol*. The cache coherence protocol can be implemented in various styles: write-invalidate or write-update depending on how to ensure write propagation, and snooping-based [95] or directory-based [42] depending on how to track states and serialize requests. Coherence can be implemented as hardware-managed, software-managed, or hybrid. Hardware coherence protocols [42, 95, 247, 281] are widely adopted in cache-based multicore systems. Typical examples include the MSI, MESI, and MOESI protocols [247]. Software-managed protocols [133, 148] and hybrid protocols [57, 132, 134, 135] such as DeNovo [57] and WayPoint [135] often leverage self-invalidations to avoid high-latency critical paths and are often seen in GPUs. One issue of the conventional cache coherence protocol is that it is limited by the narrow read-write interface: updates must be expressed as read-modify-write sequences and hence concurrent updates may introduce excessive traffic and serialization.

Cache coherence simplifies the implementation of common *memory consistency models*. The memory consistency model is an interface between software and hardware that precisely specifies how the memory system behaves with respect to reads and writes. Coherence concerns what values a read can return, while consistency concerns when writes become visible to reads. Coherence concerns reads/writes to a single memory location, while consistency concerns reads/writes to multiple memory locations. Common memory consistency models include sequential consistency [155], total store order (TSO), and release consistency [91].

Drawbacks: Caches may suffer from significant performance loss on several access patterns as explained in Section 1.1. Besides, caches introduce high area and energy overheads attributed to tag arrays, replacement policies, and coherence events. Therefore, when caches' costs outweigh their benefits, explicitly managed storage structures can be promising substitutes.

[1] Some processors have instructions for evictions, such as CLFLUSH in x86.

Explicitly managed storage structures 2.1.2

Explicitly managed storage structures include scratchpads [26, 193], stashes, queues, and buffets. Compared to caches, they have two characteristics. First, they adopt a more *explicit* programming interface: they grant applications more precise control of data organization and movement and typically accept only local addresses, while sacrificing the generality and simplicity of the programming model. Maintaining the shared-memory model may require expensive software coherence protocols [14]. Second, they allow simpler hardware implementations at the cost of reduced capability to exploit irregular data reuse.

Scratchpad [26, 193] is a storage structure that grants software full control over its content, by mapping its storage locations to an address range exposed to the applications. Therefore, a scratchpad avoids all the expensive hardware designed for global accesses such as the tag array and the cache replacement policy. As a result, it is a nice fit for GPUs and accelerators. Typical GPU applications bear enormous data-level parallelism and incur regular accesses that can be easily structured.

Stash [147] combines the characteristics of a scratchpad and a cache. Like a scratchpad, it unifies its content into the global address space, and allows a flexible data organization. Instead of controlling data movement *eagerly* as in a scratchpad, a stash records a mapping between a local region and a global address region to enable *lazy* data movement on misses and writebacks. Such a mapping requires more hardware but saves data movement when there is dynamic coarse-grain data reuse.

Caches, scratchpads, and stashes use the read-write interface, and incur long access latency when the data is off-chip. To overlap the long latency with other useful work, CPUs and GPUs adopt sophisticated mechanisms, e.g., out-of-order execution in CPUs and multithreading in GPUs. However, accelerators cannot afford such overheads as they are designed to be efficient and simple. Instead, accelerators typically adopt other storage structures that exploit *decoupled execution* to hide the latency.

Queue-based communication is the simplest form of decoupled execution. With a queue, the producer and the consumer can run simultaneously. Because of its simple synchronization and low hardware cost, queues are widely adopted in decoupled access-execute architectures [94, 102, 236, 244] and streaming multicores [67, 76, 250]. However, in-place updates, needed by many applications [54, 167], are hard to support in queues.

Buffet [201] is a storage structure proposed for accelerators to preserve the decoupling of a queue while supporting offset-based indexing and inexpensive in-place updates. A buffet achieves so by restricting the four operations performed on data, i.e., `fill`, `read`, `update`, and `shrink`, to form a fixed sequence and managing the sequence with a finite-state machine. Buffets and queues both exhibit good composability and simple synchronization, and allow decoupled execution to hide long access latencies.

Summary 2.1.3

As explained above, there are a diverse set of on-chip storage structures that can be used in a memory system. Pellauer et al. [201] introduced a taxonomy for *data orchestration*, i.e., transferring data into and out of a storage structure.

Specifically, for an on-chip storage structure, reusable data should *not be evicted when they are on-chip*, and should be *staged on chip early when they are off-chip*. Depending on how these two aspects are

Data Orchestration	Implicit	Explicit
Coupled	Cache	Scratchpad Stash
Decoupled	Decoupled Access-Execute (DAE) FIBERCACHE (Chapter 6)	Scratchpad (w/ DMAs) Buffet

Table 2.1: Taxonomy of data orchestration approaches as used in typical scenarios. A scratchpad can be combined with Direct Memory Access (DMA) engines to achieve decoupling. A stash explicitly maps local addresses to global addresses, though data movement may be triggered implicitly by load misses.

implemented, storage structures can be classified as either *explicit* or *implicit*, and can be used in either *coupled* or *decoupled* manner, as summarized in Table 2.1.

Explicit vs. Implicit Data Orchestration

Broadly speaking, the content of an on-chip storage structure can be managed in two styles: *explicit* or *implicit*. Explicitly orchestrated structures allow applications to directly control what to retain or remove, while implicitly orchestrated structures infer such decisions implicitly based on read/write accesses.

Based on this criteria, scratchpads, queues, and buffets are all explicit: they determine whether to keep or forfeit data based on direct operations such as `enqueue` and `dequeue`, or `fill` and `shrink`. In contrast, caches are implicit. A cache predicts what to evict based on the previous accesses, tracked and analyzed by the cache replacement policy.

Implicitly orchestrated structures are widely adopted in general-purpose computing. Though they incur high hardware overheads, implicitly orchestrated structures are typically equipped with hardware, such as the cache replacement policy, to exploit locality in irregular accesses. Compared to the resource budget of the on-chip storage, general-purpose systems are more concerned with the poor performance caused by irregular access patterns. These accesses exhibit locality that can be extracted only at runtime, and therefore cannot be exploited by explicitly orchestrated structures.

In contrast, domain-specific accelerators prefer explicitly orchestrated structures. Though explicitly orchestrated structures do not capture irregular data reuse, they require simple hardware and incur low energy consumption. Accelerators aim to be efficient in area and energy. Therefore, accelerator designers often choose applications and algorithms that avoid irregular accesses so that they can minimize area and energy overheads by adopting explicitly orchestrated structures such as queues and buffets (Section 6.1.4).

Coupled vs. Decoupled Data Orchestration

A storage structure can be used in either *coupled* or *decoupled* manner depending on whether the data needed is pre-staged ahead of processing to hide the memory access latency. They differ in whether the requester that initiates data movement from/to main memory is the same as the actual consumer of the data. With coupled data orchestration, the requester and the consumer are the same; with decoupled data orchestration, they are different units.

Coupled staging of data, commonly used in caches, stashes, and scratchpads without DMA engines, enjoys intuitive synchronization between data demand and data availability. However, the read roundtrip incurs long memory access latency, and requires sophisticated mechanisms to hide, such as those that exploit instruction-, memory-, or thread- level parallelism.

Decoupled staging of data, such as in scratchpads with DMA engines, buffets, and queues used in decoupled access-execute architectures, separates the requester that initiates data movement from memory and the data consumer. This decoupling allows the consumer to have low-latency accesses, as the data needed is staged ahead of processing. For instance, decoupled access-execute architectures are based on queues, and separate the units for sending read requests and receiving read responses. Though they may work well on regular applications, on irregular applications, they may suffer from load imbalance and lack of control-flow mechanisms [192]. Finally, decoupled data orchestration must also incorporate careful synchronization: too aggressive staging may overwrite live data, too conservative staging may result in poor efficiency.

FIBERCACHE (Chapter 6) is a storage structure with implicit and decoupled data orchestration. It orchestrates data implicitly to exploit irregular data reuse, and uses decoupled execution to hide the long memory access latencies incurred by dependent reads.

Improving Memory System Performance 2.2

Despite the diversity in organizing on-chip storage, there are common challenges in memory systems:

- Accesses that exhibit *poor locality*, e.g., accesses to small fragments, waste storage capacity.
- Concurrent updates in shared-memory systems incur *excessive communication and synchronization*.
- Dependent reads accumulate a *long access latency* that is expensive to hide in systems with implicitly orchestrated storage structures.

Improving Locality 2.2.1

The performance of an on-chip storage structure can be hurt significantly if accesses do not exhibit good locality. To address this issue, prior work has proposed many techniques. They either change the data representation or modify the compute schedule.

Changing the data representation can be effective on improving the locality of accesses. This can be performed either in software, such as graph/tensor preprocessing [16, 69, 103, 124, 203, 264, 282] and cache-aware data structures [293], or in hardware, such as compressed caches [11, 101, 209], texture caches [100], and storage structures with variable-sized lines [152] or object awareness [252]. Besides increased locality, modifying the data representation can unlock potential for SIMD operations, as data needed by similar operations are likely to be grouped together.

HTA exploits this with a table format that maps multiple key-value pairs to the same cache line. This allows multiple probes in a single lookup to be performed in parallel with only one memory access.

Changing the compute schedule is the other approach to improve locality and can be performed by either software or hardware.

Some techniques do not require changes in data representation. Instead, they schedule compute to maximize data reuse, either statically [7, 243, 275], or dynamically [3, 50, 115, 122, 234, 276]. Others are combined with modifications in data representation [90, 160, 185]. For instance, tiling [6, 107, 116, 239, 255, 290, 294] typically works on a nested loop: Based on knowledge of accesses, tiling sometimes reorganizes data in storage-fitting chunks [107, 116, 290, 294], which effectively increases the iteration levels in the loop, and then reorders the iteration levels to improve data reuse. Since it is effective on

regular accesses, many accelerators built on explicitly orchestrated storage structures rely on tiling to improve locality and avoid overflows [107].

2.2.2 Reducing Communication and Synchronization

Concurrent updates can incur excessive traffic and serialization. This is an especially serious problem for shared memory systems with cache coherence. Many software and hardware optimizations seek to reduce the cost of updates. Though often presented in the context of specific algorithms or implementations, we observe these techniques can be classified into two categories: *Privatization* [21, 62, 190] exploits commutativity to avoid communication and serialization. *Delegation* [37, 38] is more general, in that it does not require operations to be commutative, but it suffers from serialization and complexity in the memory consistency model.

Privatization lowers the cost of commutative updates by using thread-local variables [21, 62, 190]. Privatization schemes buffer updates in thread-private storage, and require reads to reduce these thread-private updates to produce the correct value. Privatization is most commonly used to implement reduction variables efficiently, often with language support (e.g., reducers in MapReduce [77], OpenMP pragmas, and Cilk Plus hyperobjects [90]). Privatization is generally used when updates are frequent and reads are rare.

Privatization is limited to commutative updates, and works best when data goes through long update-only phases without intervening reads. However, privatization has two major sources of overhead. First, software reductions are much slower, making finely-interleaved reads and updates inefficient. Second, with N threads, privatized variables increase the memory footprint by a factor of N. This makes naïve privatization impractical in many contexts (e.g., reference counting). Dynamic privatization schemes [62, 190, 278] can lessen space overheads, but add time overheads and complexity.

Delegation sends updates to a single location to reduce data movement and hence the cost of updates. Delegation does not leverage commutativity. This limits their performance benefits, but makes them applicable for non-commutative operations.

Specifically, in software, delegation schemes send updates to a single thread [37, 38]. They divide shared data among threads and send updates to the corresponding thread, using shared-memory queues [37] or active messages [225, 258]. Delegation is common in architectures that combine shared memory and message passing [225, 270] and in NUMA-aware data structures [37, 38]. Although delegation reduces data movement and synchronization, it still incurs global traffic and serialization.

Remote memory operations (RMOs) [97, 114, 228, 287] are the hardware counterpart of delegation. Rather than caching lines to be updated, update operations are sent to a fixed location, as shown in Figure 2.1. The NYU Ultracomputer [97] proposed implementing atomic fetch-and-add using adders in network switches, which could coalesce multiple requests on their way to memory. The Cray T3D [137], T3E [228], and SGI Origin [157] implemented RMOs at the memory controllers, while TilePro64 [114] and recent GPUs [268] implement RMOs in shared caches. Prior work has also proposed adding caches to memory controllers to accelerate RMOs [287] and data-parallel RMOs [9].

Although RMOs reduce update costs for both commutative and non-commutative operations, they suffer from two issues. First, while RMOs avoid ping-ponging cache lines, they still require sending every update to a shared, fixed location, causing global traffic, a serious issue in shared-memory systems. RMOs are also limited by the throughput of the single updater. For example, in Figure 2.1, frequent remote-add requests drive the shared cache's ALU near saturation. Second, consistency models must be considered because there can be concurrent users of the data, but strong consistency models are

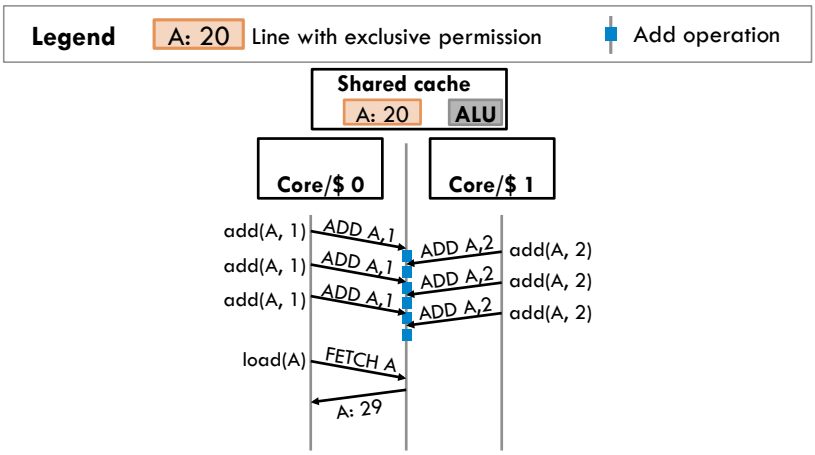


Figure 2.1: With remote memory operations, cores send updates to a fixed location, the shared cache in this case.

challenging to implement with RMOs, as it is harder to constrain memory operation order. For example, TSO requires making stores globally visible in program order, which is feasible with local store buffers, but much more complicated when stores are also performed by remote updaters. As a result, most implementations provide weakly-consistent RMOs. Timestamp-based order validation [153, §5] allows strong consistency with RMOs, but it is complicated.

Hiding Memory Access Latency 2.2.3

Irregular applications often leverage implicitly orchestrated storage structures, such as caches, to capture irregular data reuse. However, the high latency of reads that access main memory can be a serious issue. Such high latency can accumulate on dependent reads, eventually bottlenecking the system performance. Therefore, techniques that hide the memory access latency for implicitly managed storage structures are necessary.

Prior work can be classified into two categories, they either tolerate the long latency by exploiting parallelism, i.e., finding something else to do, or prefetching and specialized fetching.

Exploiting parallelism is effective as it overlaps the long memory access latency with other useful work. This works because of Little's law [163]. By allowing multiple long-latency events to happen simultaneously, the system can maintain its high throughput that would otherwise be degraded by the long latency.

Modern CPUs, GPUs, and accelerators exploit various levels of parallelism, such as instruction-level parallelism in out-of-order execution, thread-level parallelism in multicore CPUs and GPUs, operation-level parallelism in accelerators [106, 144, 151], and pipeline parallelism [67, 118, 138, 192]. However, they do not work well when there is little parallelism to begin with, e.g., on dependent reads.

Prefetching and specialized fetching are crucial to get some of the benefits of explicit data orchestration in a cache-based system. In this way, accesses from the real consumers can incur low latency since the needed data is already resident in the cache.

Prefetching allows data to be speculatively fetched into the cache ahead of accesses, so that actual accesses are likely to hit, reducing the latency drastically. Prefetching can be implemented in either

hardware [10, 27, 53, 127, 168, 187, 280] or software [182, 183]. Hardware prefetchers capture dynamic patterns at runtime. However, simple prefetchers can be limited by the access patterns they can handle, while complex prefetching techniques such as runahead execution [187] and indirect prefectures [10, 27, 280] incur high hardware overheads. Software prefetchers allow direct control from applications, but can incur high instruction overheads and poor cross-platform compatibility. Specialized fetchers [144, 151, 185, 192] are capable of fetching data non-speculatively into the cache. They rely on customized hardware and are often limited to specific access patterns and applications.

In this thesis, we propose GAMMA, a SPMSPM accelerator that adopts a novel storage idiom FIBERCACHE. FIBERCACHE is customized for an efficient SPMSPM dataflow. It combines decoupled execution and caching to both hide memory access latency and exploit irregular data reuse.

COUP 3

In this chapter, we present COUP, a general technique that extends coherence protocols to allow *local and concurrent commutative updates*. COUP allows multiple caches to acquire a line with *update-only* permission, and satisfy commutative-update requests locally, buffering and coalescing updates. Upon a read, the coherence protocol collects all the local updates and *reduces* them to produce the correct value before granting read permission.

Unlike RMOs (Section 2.2.2) that suffer from serialization and complicate memory consistency, COUP confers significant benefits over RMOs, especially when data receives several consecutive updates before being read. Moreover, COUP maintains full cache coherence and does not affect the memory consistency model. This makes COUP easy to apply to current systems and applications. Note that COUP's advantages come at the cost of a more restricted set of operations: COUP is limited to commutative updates, while RMOs support non-commutative operations such as fetch-and-add and compare-and-swap.

COUP also completes a symmetry between hardware and software schemes to reduce the cost of updates. Just as remote memory operations are the hardware counterpart to delegation, COUP *is the hardware counterpart to privatization*. COUP has two benefits over software privatization. First, transitions between read-only and update-only modes are much faster, so COUP remains practical in many scenarios where software privatization requires excessive synchronization. Second, privatization's thread-local copies increase memory footprint and add pressure to shared caches, while COUP does not.

We demonstrate COUP's utility by applying it to improve the performance of *single-word update operations*, which are currently performed with expensive atomic read-modify-write instructions.

Overall, we make the following contributions:

- We present COUP, a technique that extends coherence protocols to support concurrent commutative updates (Section 3.2 and Section 3.3). We show that COUP preserves coherence and consistency (Section 3.4), and imposes small verification costs (Section 3.5).
- We identify several update-heavy parallel applications where current techniques have clear shortcomings (Section 3.6), and discuss how COUP addresses them.
- We evaluate COUP under simulation, using single- and multi-socket systems (Section 3.7). At 128 cores, COUP improves the performance of update-heavy benchmarks by 4%–2.4×, and reduces traffic by up to 20×.

In summary, COUP shows that extending coherence protocols to leverage the semantics of commutative updates can substantially improve performance without sacrificing the simplicity of cache coherence.

Motivation 3.1

Historically, exploiting *commutativity* has been a fruitful approach to reduce the cost of updates. Commutativity [265] has been widely exploited in parallel systems, such as databases [21, 190], parallelizing compilers [204, 219], and runtimes [149, 204].

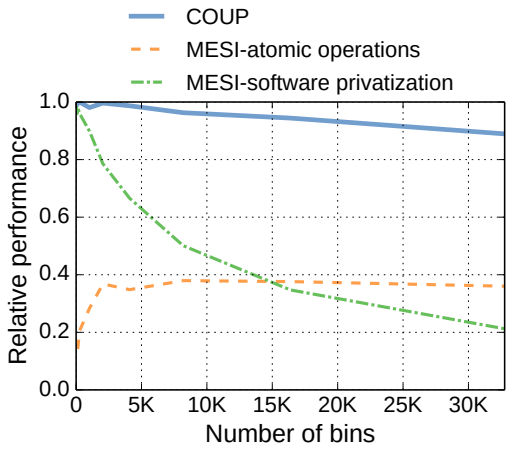


Figure 3.1: Performance of parallel histogram implementations using atomics, software privatization, and COUP. More bins reduce contention and increase privatization overheads, favoring atomics. COUP does not suffer these overheads, so it outperforms both software implementations.

Commutative updates are common in many cases. *Strictly commutative* operations produce exactly the same final state when reordered. For instance, consider two additions to a counter. Reordering them does not affect the final result at all, and therefore integer addition is strictly commutative. Besides addition, multiplication, minimization, maximization and bitwise logical operations are all strictly commutative. Such operations are typically implemented as single instructions in non-speculative parallel system.

Privatization is the software technique that exploits commutativity to reduce the costs of updates. However, it incurs significant overheads that often make privatization underperform conventional updates. For instance, Jung et al. [126] propose parallel histogram implementations using both atomic operations and privatization. These codes process a set of input values, and produce a histogram with a given number of bins. Jung et al. note that privatization is desirable with a few output bins, but works poorly with many bins, as the reduction phase dominates execution time and hurts locality. Figure 3.1 shows this tradeoff. It compares the performance of histogram implementations using atomic fetch-and-add, privatization, and COUP, when running on 64 cores (see Section 3.7 for methodology details). In this experiment, all schemes process a large, fixed number of input elements. Each line shows the performance of a given implementation as the number of output bins (x-axis) changes from 32 to 32 K. Performance is reported relative to COUP's at 32 bins (higher numbers are better). While the costs of privatization impose a delicate tradeoff between both implementations in software, COUP robustly outperforms both. In fact, COUP is the *hardware counterpart of privatization*.

3.2 COUP Example: Extending MSI

We first present the main concepts and operation of COUP through a concrete, simplified example. Consider a system with a single level of private caches, kept coherent with the MSI protocol. This system has a single shared last-level cache with an in-cache directory. It implements a single commutative-update operation, addition. Finally, we restrict this system to use single-word cache blocks. We will generalize COUP to other protocols, operations, and cache hierarchies in Section 3.3.

3.2.1 Structural changes

COUP requires modest changes to hardware structures, summarized in Figure 3.2 and described below.

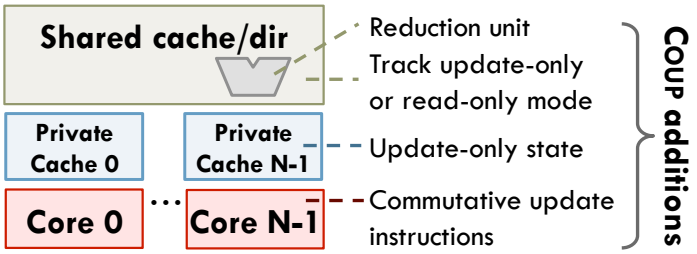


Figure 3.2: Summary of additions and modifications needed to support COUP.

Commutative-update instructions: In most ISAs, COUP needs additional instructions that let programs convey commutative updates, as conventional atomic instructions (e.g., fetch-and-add) return the latest value of the data they update. In this case, we add a *commutative-addition* instruction, which takes an address and a single input value, and does not write to any register.

Some ISAs may not need additional instructions. For instance, the recent Heterogeneous System Architecture (HSA) includes atomic-no-return instructions that do not return the updated value [2]. While these instructions were likely introduced to reduce the cost of RMOs, COUP could use them directly.

Update-only permission: COUP extends MSI with an additional state, *update-only (U)*, and a third type of request, *commutative update (C)*, in addition to conventional reads (R) and writes (W). We call the resulting protocol MUSI. Figure 3.3 shows MUSI's state-transition diagram for private caches. MUSI allows multiple private caches to hold read-only permission to a line and satisfy read requests locally (S state); multiple private caches to hold update-only permission to a line and satisfy commutative-update requests locally (U state); or at most a single private cache to hold exclusive permission to a line and satisfy all types of requests locally (M state). By allowing M to satisfy commutative-update requests, interleaved updates and reads to *private* data are as cheap as in MSI.

MUSI's state-transition diagram shows a clear symmetry between S and U: all transitions caused by R/C requests in and out of S match those caused by C/R requests in and out of U. We will exploit this symmetry in Section 3.5 to simplify our implementation.

Directory state: Conventional directories must track both the sharers of each line (using a bit-vector or other techniques [45, 223, 284]), and, if there is a single sharer, whether it has exclusive or read-only permission. In COUP, the directory must track whether sharers have exclusive, read-only, or update-only permission. The sharers bit-vector can be used to track both multiple readers or multiple updaters, so MUSI requires only one extra bit per directory tag.

Reduction unit: Though cores can perform local updates, the memory system must be able to perform reductions. Thus, COUP adds a reduction unit to the shared cache, consisting of an adder in this case.

Protocol operation 3.2.2

Performing commutative updates: Both the M and U states provide enough permissions for private caches to satisfy update-only requests. In M, the private cache has the actual data value; in U, the cache has a partial update. In either case, the core can perform the update by atomically reading the data

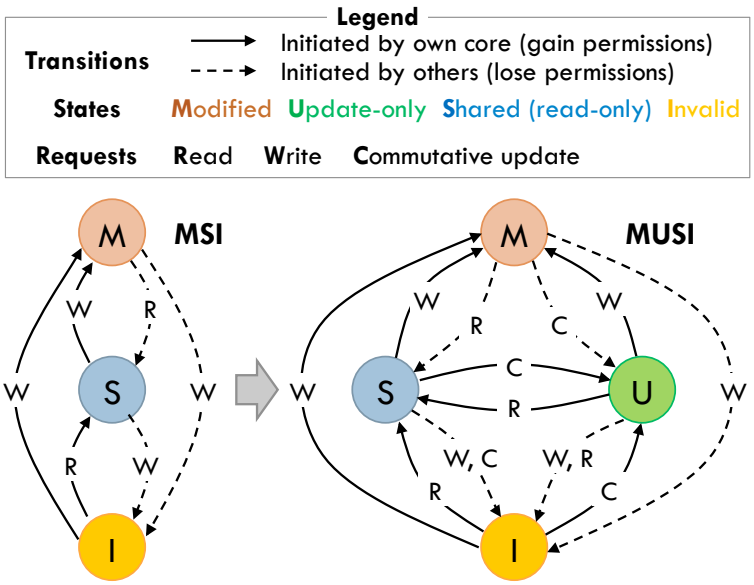


Figure 3.3: State-transition diagrams of MSI and MUSI protocols. For clarity, diagrams omit actions that do not cause a state transition (e.g., R requests in S).

from the cache, modifying it (by adding the value specified by the commutative-add instruction) and storing the result in the cache. The cache cannot allow any intervening operations to the same address between the read and the write. This scheme can reuse the existing core logic for atomic operations. We assume this scheme in our implementation, but note that alternative implementations could treat commutative updates like stores to improve performance (e.g., using update buffers similar to store buffers and performing updates with an ALU at the L1).

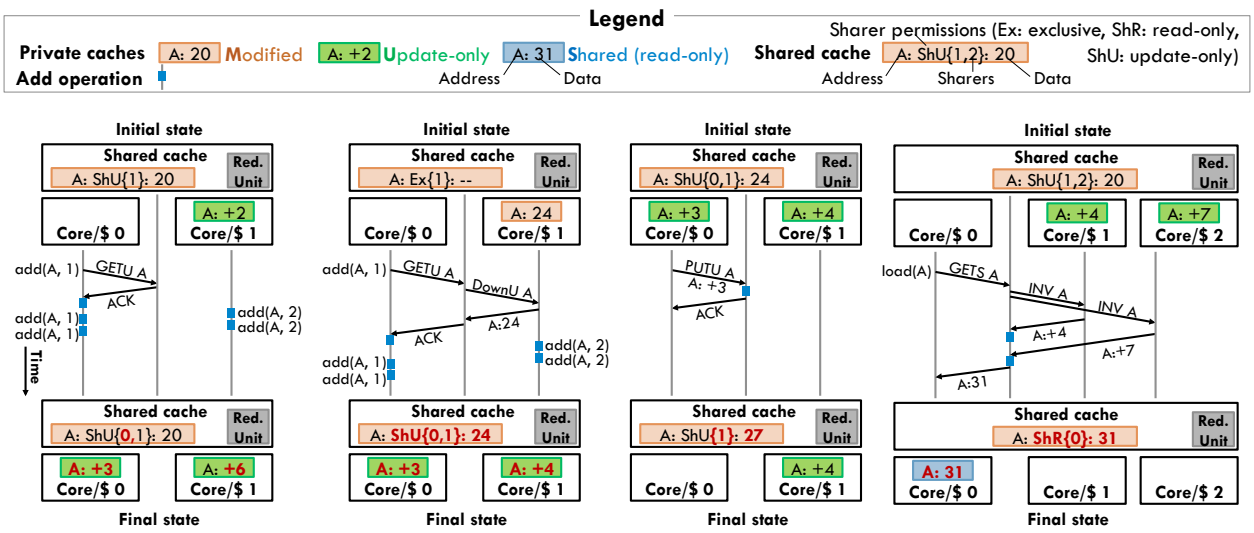
Entering the U state: When a cache has insufficient permissions to satisfy an update request (I or S states), it requests update-only permission from the directory. The directory invalidates any copies in S, or downgrades the single copy in M to U, and grants update-only permission to the requesting cache, which transitions to U. Thus, there are two ways a line can transition into the U state: by requesting update-only permission to satisfy a request from its own core, as shown in Figure 3.4a; or by being downgraded from M, as shown in Figure 3.4b.

When a line transitions into U, its contents are always initialized to the *identity element*, 0 for commutative addition. This is done even if the line had valid data. This avoids having to track which cache holds the original data when doing reductions. However, reductions require reading the original data from the shared cache.

Leaving the U state: Lines can transition out of U due to either evictions or read requests.

Evictions initiated by a private cache (to make space for a different line) trigger a *partial reduction*, shown in Figure 3.4c: the evicting cache sends its partial update to the shared cache, which uses its reduction unit to aggregate it with its local copy.

The shared cache may also need to evict a line that private caches hold in U. This triggers a *full reduction*: all caches with update-only permission are sent invalidations, reply with their partial updates, and the shared cache uses its reduction unit to aggregate all partial updates and its local copy, producing the final value.



(a) Upgrade to U caused by commutative-update request (b) Downgrade caused by commutative-update request (c) Partial reduction caused by eviction (d) Full reduction caused by read request

Figure 3.4: MUSI protocol operation: (a) granting update-only (U) state; (b) downgrade from M to U due to an update request from another core; (c) partial reduction caused by an eviction from a private cache; and (d) full reduction caused by a read request. Each diagram shows the initial and final states in the shared and private caches.

Finally, read requests from any core also trigger a full reduction, as shown in Figure 3.4d. Depending on the latency and throughput of the reduction unit, satisfying a read request can take somewhat longer than in conventional protocols. Hierarchical reductions can rein in reduction overheads with large core counts (Section 3.3). In our evaluation, we observe that reduction overheads are small compared to communication latencies.

Generalizing COUP 3.3

We now show how to generalize COUP to support multiple operations, larger cache blocks, other protocols, and deeper cache hierarchies.

Multiple operations: Formally, COUP can be applied to any *commutative semigroup* $(G, \circ)$.[1] For example, G can be the set of 32-bit integers, and $\circ$ can be addition, multiplication, and, or, xor, min, or max.

Supporting multiple operations in the system requires minor changes. First, additional instructions are needed to convey each type of update. Second, reduction units must implement all supported operations. Third, the directory and private caches must track, for each line in U state, what type of operation is being performed. Fourth, COUP must serialize commutative updates of different types, because they do not commute in general (e.g., + and $*$ do not commute with each other). This can be accomplished by performing a full reduction every time the private cache or directory receives an update request of a type different from the current one.

[1] $(G, \circ)$ is a commutative semigroup iff $\circ : G \times G \rightarrow G$ is a binary, associative, commutative operation over elements of set G, and G is closed under $\circ$.

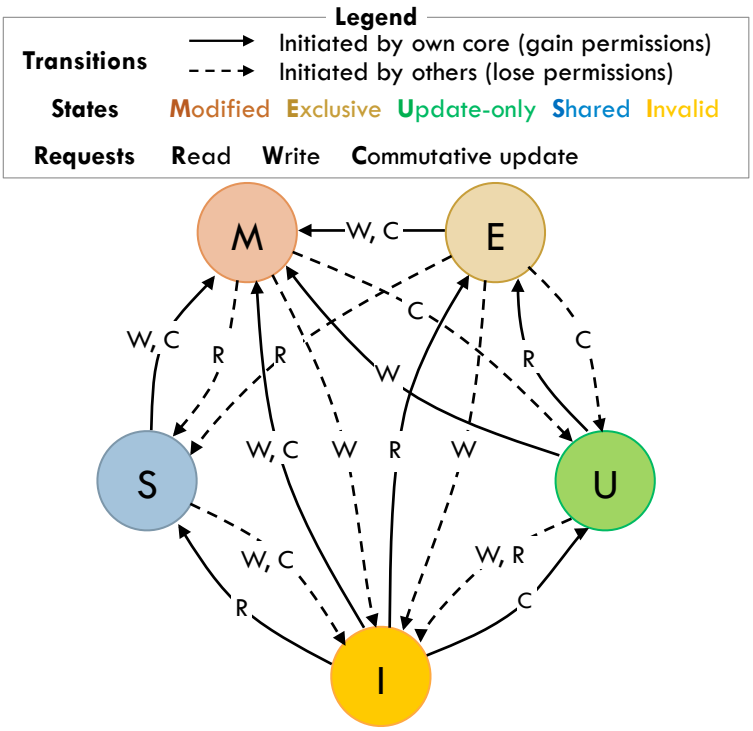


Figure 3.5: State-transition diagram of MEUSI. Just as MESI grants E to a read request if a line is unshared, MEUSI grants M to an update request if a line is unshared. For clarity, the diagram omits actions that do not cause a state transition (e.g., C requests in U).

Larger cache blocks: Supporting multi-word blocks is trivial if $(G, \circ)$ has an *identity element* (formally, this means $(G, \circ)$ is a commutative monoid). The identity element produces the same value when applied to any element in G. For example, the identity elements for addition, multiplication, *and*, and *min* are 0, 1, all-ones, and the maximum representable integer, respectively.

All the operations we implement in this work have an identity element. In this case, it is sufficient to initialize every word of the cache block to the identity element when transitioning to U. Reductions perform element-wise operations even on words that have received no updates. Note this holds *even if those words do not hold data of the same type*, because applying $\circ$ on the identity element produces the same output, so it does not change the word's bit pattern. Alternatively, reduction units could skip operating on words with the identity element.

In general, not all operations may have an identity element. In such cases, the protocol would require an extra bit per word to track uninitialized elements.

Finally, note we assume that data is properly aligned. Supporting commutative updates to unaligned data would require more involved mechanisms to buffer partial updates. If the ISA allows unaligned accesses, they can be performed as normal read-modify-writes.

Other protocols: COUP can extend protocols beyond MSI. Figure 3.5 shows how MESI [197] is extended to MEUSI, which we use in our evaluation. Note that update requests enjoy the same optimization that E introduces for read-only requests: if a cache requests update-only permission for a line and no other cache has a valid copy, the directory grants the line directly in M.

Deeper cache hierarchies: COUP can operate with multiple intermediate levels of caches and directories. COUP simply requires a reduction unit at each intermediate level that has multiple children that can issue update requests. For instance, a system with private per-core L1s and L2s and a fully shared L3 needs reduction units only at L3 banks. However, if each L2 was shared by two or more L1Ds, a reduction unit would be required in the L2s as well.

Hierarchical organizations lower the latency of reductions in COUP, just as they lower the latency of sending and processing invalidations in conventional protocols: on a full reduction, each intermediate level aggregates all partial updates from its children before replying to its parent. For example, consider a 128-core system with a fully-shared L4 and 8 per-socket L3s, each shared by 16 cores. In this system, a full reduction of a line shared in U state by all cores has $8 + 16 = 24$ operations in the critical path—far fewer than the 128 operations that a flat organization would have, and not enough to dominate the cost of invalidations.

Other contexts: We focus on single-word atomic operations and hardware cache coherence, but note that COUP could apply to other contexts. For example, COUP could be used in software coherence protocols (e.g., in distributed shared memory).

Coherence and Consistency 3.4

COUP maintains cache coherence and does not change the consistency model.

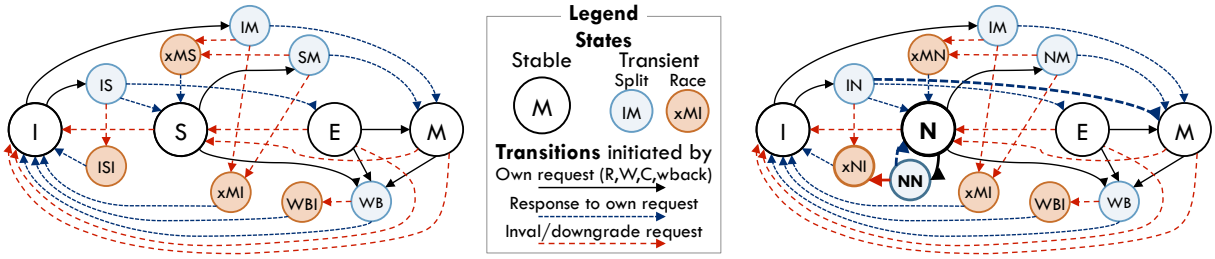
Coherence: A memory system is coherent if, for each memory location, it is possible to construct a hypothetical serial order of all operations to the location that is consistent with the results of the execution and that obeys two invariants [68, §5.1.1]:[2]

1. Operations issued by each core occur in the order in which they were issued to the memory system by that core.
2. The value returned by each read operation is the value written to that location in the serial order.

In COUP, a location can be in exclusive, read-only, or update-only modes. The baseline protocol that COUP extends already enforces coherence in and between exclusive and read-only modes. In update-only mode, multiple cores can concurrently update the location, but because updates are commutative, *any serial order* we choose produces the same execution result. Thus, the first invariant is trivially satisfied. Moreover, transitions from update-only to read-only or exclusive modes propagate all partial updates and make them visible to the next reader. Thus, the next reader always observes the last value written to that location, satisfying the second property. Therefore, COUP maintains coherence.

Consistency: As long as the system restricts the order of memory operations as strictly for commutative updates as it does for stores, COUP does not affect the consistency model. In other words, it is sufficient for the memory system to consider commutative updates as being equivalent to stores. For instance, by having store-load, load-store, and store-store fences apply to commutative updates as well, systems with relaxed memory models need not introduce new fence instructions.

[2]Others reason about coherence using the *single-writer, multiple-reader* and the *data-value* invariants [241], which are sufficient but not necessary. COUP does not maintain the single-writer, multiple-reader invariant.



(a) MESI

(b) MEUSI with generalized N state

Figure 3.6: COUP implementation: (a) full state-transition diagram for the L1 cache on the baseline two-level MESI protocol; (b) corresponding MEUSI state-transition diagram. The non-exclusive state, N, generalizes S and U, and requires only an extra transient state and four transitions over MESI.

3.5 Implementation and Verification Costs

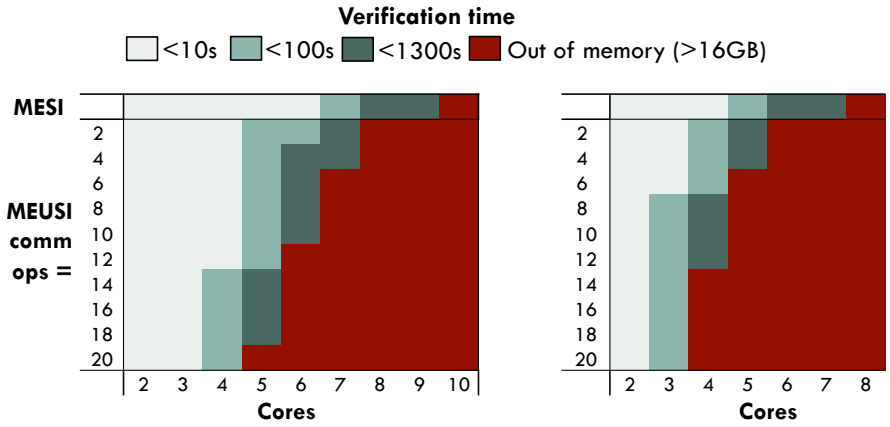
While we have presented COUP in terms of stable states, realistic protocols implement coherence transactions with additional transient states and are subject to races, which add complexity and hinder verification. By studying full implementations of MESI and MEUSI, we show that COUP requires a minimal number of transient states and adds modest verification costs.

We first implement MESI protocols for two- and three-level cache hierarchies. Our implementations work on networks with unordered point-to-point communication, and use two virtual networks without any message buffering at the endpoints. In the two-level protocol, the L1 coherence controller has 12 states (4 stable, 8 transient), and the L2 has 6 states (3 stable, 3 transient). Figure 3.6a shows the state-transition diagram of the more complex L1 cache. In the three-level protocol, the L1 has 14 states (4 stable, 10 transient), the L2 has 38 (9 stable, 29 transient), and the L3 has 6 (3 stable, 3 transient).

Generalized non-exclusive state: While we have introduced U as an additional state separate from S, both have a strong symmetry and many similarities. In fact, reads are just another type of commutative operation. We leverage this insight to simplify COUP's implementation by integrating S and U under a single, generalized *non-exclusive* state, N. This state requires minor extensions over the machinery already described in Section 3.3 to support multiple commutative updates.

Multiple caches can have a copy of the line in N, but all copies must be under the same operation type, which can be read-only or one of the possible commutative updates. An additional field per line tracks its operation type when in N. Non-exclusive and downgrade requests are tagged with the desired operation type. E and M can satisfy all types of requests; commutative updates cause an E→M transition. N can satisfy non-exclusive requests of the same type, but requests of a different type trigger an invalidation (if starting from read-only) or a reduction (if starting from a commutative-update type) and cause a type switch. Invalidations and reductions involve the same request-reply sequence, so they can use the same transient states.

Implementing two-level MEUSI this way requires 13 states in the L1 and 6 states in the L2. Compared to two-level MESI, MEUSI introduces only one extra L1 transient state. Figure 3.6b shows the L1's state-transition diagram, which is almost identical to MESI's. The new transient state, *NN*, is used when moving between operation types (e.g., from read-only to commutative-add or from commutative-and to commutative-or). Our three-level MEUSI protocol is also similar to three-level MESI: the L1 has 15 states (one more transient than MESI, *NN*), the L2 has 43 (five more transient states than MESI, which, similarly to *NN*, implement transitions between operation types), and the L3 has 6.



(a) Two-level protocols (b) Three-level protocols

Figure 3.7: COUP exhaustive verification costs for two- and three-level protocols. Costs grow much more quickly with the number of cores and levels than the number of commutative updates.

Verification costs: We use Murphi [81] to verify MESI and MEUSI. We adopt common simplifications to limit the state space, modeling caches with a single 1-bit line; self-eviction rules model a limited capacity. In three-level protocols, we model systems with a single L2 and a single L3, and simulate traffic from other L2s with L3-issued invalidation and downgrade rules. Even then, Murphi can only verify systems of up to 4-8 cores, a well-known limitation of this approach [288, 289].

MEUSI's verification costs grow more quickly with the number of cores and levels than the number of commutative operations. Figure 3.7 reports the verification times for two- and three-level MESI and MEUSI protocols supporting 2–20 commutative-update types. We run Murphi on a Xeon E5-2670, and limit it to 16 GB of memory. Murphi can exhaustively verify MESI up to 7-9 cores and MEUSI up to 3-7 cores depending on the number of levels and commutative updates. This shows that MEUSI can be effectively verified up to a large number of commutative updates. Moreover, just as protocol designers assume that modeling a few cores provide reasonable coverage, verifying up to a few commutative operations should be equally reasonable.

Motivating Applications 3.6

In this work, we apply COUP to accelerate single-word updates to shared data. To guide our design, we first study under what circumstances COUP is beneficial over state-of-the-art software techniques, and illustrate these circumstances with specific algorithms and applications.

As discussed in Section 3.1, COUP is the hardware counterpart to privatization. Privatization schemes create several replicas of variables to be updated. Each thread updates one of these replicas, and threads synchronize to reduce all partial updates into a single location before the variable is read.

In general, COUP outperforms prior software techniques if *either* of the following two conditions holds:

- Reads and updates to shared data are finely interleaved. In this case, software privatization has large overheads due to frequent reductions, while COUP can move a line from update-only mode to read-only mode at about the same cost as a conventional invalidation. Thus, privatization needs many updates per core and data value to amortize reduction overheads, while COUP yields benefits with as little as two updates per update-only epoch.
- A large amount of shared data is updated. In this case, privatization significantly increases memory footprint and puts more pressure on shared caches.

We now discuss several parallel patterns and applications that have these properties.

3.6.1 Separate Update- and Read-Only Phases

Several parallel algorithms feature long phases where shared data is either only updated or only read. Privatization techniques naturally apply to these algorithms.

Reduction variables: Reduction variables are objects that are updated by multiple iterations of a loop using a binary, commutative operator (a reduction operator) [214, 215], and their intermediate state is not read. Reduction variables are natively supported in parallel programming languages and libraries such as HPF [145], MapReduce [77], OpenMP [84], TBB [216], and Cilk Plus [90]. Prior work in parallelizing compilers has developed a wide array of techniques to detect and exploit reduction variables [125, 214, 215]. Reductions are commonly implemented using parallel reduction trees, a form of privatization. Each thread executes a subset of loop iterations independently, and updates a local copy of the object. Then, in the reduction phase, threads aggregate these copies to produce a single output variable.

Reduction variables can be small, for example when computing the mean or maximum value of an array. In these cases, the reduction variable is a single scalar, the reduction phase takes negligible time, and COUP would not improve performance much over software reductions.

Reduction variables are often larger structures, such as arrays or matrices. For example, consider a loop that processes a set of input values (e.g., image pixels) and produces a histogram of these values with a given number of bins. In this case, the reduction variable is the whole histogram array, and the reduction phase can dominate execution time [126], as shown in Figure 3.1. Yu and Rauchwerger [278] propose several adaptive techniques to lower the cost of reductions, such as using per-thread hash tables to buffer updates, avoiding full copies of the reduction variable. However, these techniques add time overheads and must be applied selectively [278]. Instead, COUP achieves significant speedups by maintaining a single copy of the reduction variable in memory, and overlapping the loop and reduction phases.

Reduction variables and other update-only operations often use floating-point data. For example, depending on the format of the sparse matrix, sparse matrix-vector multiplication can require multiple threads to update overlapping elements of the output vector [9]. However, floating-point operations are not associative or commutative, and the order of operations can affect the final result in some cases [257]. Common parallel reduction implementations are non-deterministic, so we choose to support floating-point addition in COUP. Implementations desiring reproducibility can use slower deterministic reductions in software [79].

Ghost cells: In iterative algorithms that operate on regular data, such as structured grids, threads often work on disjoint chunks of data and only need to communicate updates to threads working on neighboring chunks. A common technique is to buffer updates to boundary cells using ghost or halo cells [141], private copies of boundary cells updated by each thread during the iteration and read by neighboring threads in the next iteration. Ghost cells are another form of privatization, different from reductions in that they capture point-to-point communication. COUP avoids the overheads of ghost cells by letting multiple threads update boundary cells directly.

The ghost cell pattern is harder to apply to iterative algorithms that operate on irregular data, such as PageRank [194, 227]. In these cases, partitioning work among threads to minimize communication can be expensive, and is rarely done on shared-memory machines [227]. By reducing the cost of concurrent updates to shared data, COUP helps irregular iterative algorithms as well.

Interleaved Updates and Reads 3.6.2

Several parallel algorithms read and update shared data within the same phase. Unlike the applications in Section 3.6.1, software privatization is rarely used in these cases, as software would need to detect data in update-only mode and perform a reduction before each read. By contrast, COUP transparently switches cache lines between read-only and update-only modes in response to accesses, improving performance even with a few consecutive updates or reads.

Graph traversals: High-performance implementations of graph traversal algorithms such as breadth-first search (BFS) encode the set of visited nodes in a bitmap that fits in cache to reduce memory bandwidth [8, 55]. The first thread that visits a node sets its bit, and threads visiting neighbors of the node read its bit to find whether the node needs to be visited.

Existing implementations use atomic-*or* operations to update the bitmap [8], or use non-atomic load-*or*-store sequences, which reduce overheads but miss updates, causing some nodes to be visited multiple times [55]. In both cases, updates from multiple threads are serialized. In contrast, COUP allows multiple concurrent updates to bits in the same cache line.

Besides graph traversals, commutative updates to bitmaps are common in other contexts, such as recently-used bits in page replacement policies [66], buddy memory allocation [142], and other graph algorithms [154].

Reference counting: Reference counting is a common automatic memory management technique. Each object has a counter to track the number of active references. Threads increment the object's counter when they create a reference, and decrement and read the counter when they destroy a reference, When the reference count reaches zero, the object is garbage-collected.

Using software techniques to reduce reference-counting overheads is a well-studied problem [61, 62, 85, 176]. Scalable Non-Zero Indicators (SNZIs) [85] reduce the cost of non-zero checks. SNZIs keep the global count using a tree of counters. Threads increment and decrement different nodes in the tree, and may propagate updates to parent nodes. Readers just need to check the root node to determine whether the count is zero. SNZIs make non-zero checks fast and allow some concurrency in increments and decrements, but add significant space and time overheads, and need to be carefully tuned.

Refcache [61] delays and batches reads to reference counts, which allows it to use privatization. Threads maintain a software cache of reference counter deltas, which are periodically flushed to the global counter. When the global counter stays at zero for a sufficiently long time, the true count is known to be zero and the object is deallocated. This approach reduces reference-counting overheads, but delayed deallocation hurts memory footprint and locality.

COUP enables shared reference counters with no space overheads and less coherence traffic than shared counters. COUP also allows delayed reference counting as in Refcache without a software cache (Section 3.7.4).

Evaluation 3.7

Methodology 3.7.1

Modeled systems: We perform microarchitectural, execution-driven simulation using zsim [224]. We evaluate single- and multi-socket systems with up to 128 cores and a four-level cache hierarchy, shown in Figure 3.8. Table 3.1 details the configuration of these systems. Each processor chip has 16 cores.

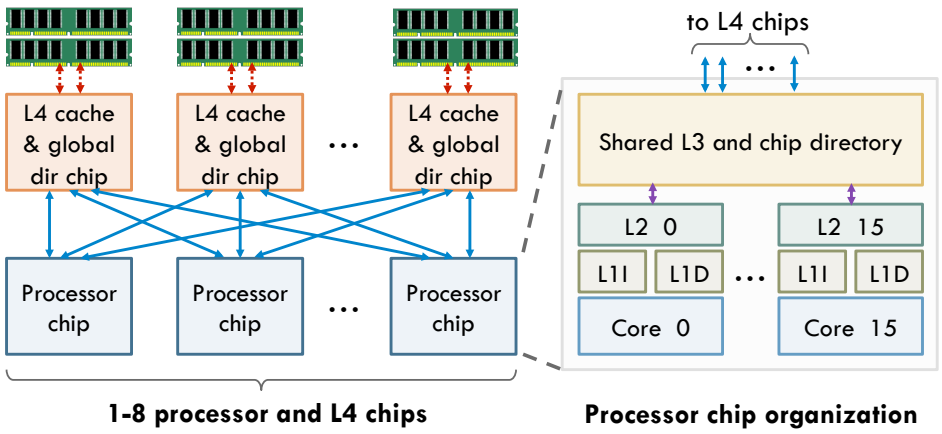


Figure 3.8: Architecture of the simulated system.

Processor chip	**Cores**	1–128 cores, 16 cores/processor chip, x86-64 ISA, 2.4 GHz, Nehalem-like OOO [224]
	L1 caches	32 KB, 8-way set-associative, split D/I, 4-cycle latency
	L2 caches	256 KB private per-core, 8-way set-associative, inclusive, 7-cycle latency
	L3 caches	32 MB, 8 banks, 16-way set-associative, inclusive, 27-cycle latency, in-cache directory
	Off-chip network	Dancehall topology, 40-cycle point-to-point links between each processor and L4 chip
	L4 & dir chip	128 MB, 8 banks/chip, 16-way set-associative, inclusive, 35-cycle latency, in-cache directory
	Coherence	MESI/MEUSI, 64 B lines, no silent drops
	Main memory	4 DDR3-1600-CL10 channels per L4 chip, 64-bit bus, 2 ranks/channel

Table 3.1: Configuration of the simulated system.

Each core has private L1s and a private L2, and all cores in the chip share a banked L3 cache with an in-cache directory. The system supports up to 8 processor chips, connected in a dancehall topology to the same number of L4 chips. Each of these chips contains a slice of the L4 cache and global in-cache directory, and connects to a fraction of main memory. This organization is similar to the IBM z13 [263].

We compare MESI and MEUSI (Figure 3.5). With MEUSI, each L3 and L4 bank has a reduction unit. We perform hierarchical reductions as described in Section 3.3: on a full reduction, each L3 bank invalidates all its children, aggregates their partial updates, and sends a single response to the L4 controller.

COUP operations and data types: We add support for eight commutative-update types:

- Addition of 16, 32, and 64-bit integers, and 32 and 64-bit floating-point values.
- AND, OR, and XOR bitwise logical operations on 64-bit words.

We observe multiplication update-only operations are rare, so we do not support multiplication. We also observe *min* and *max* are often used with scalar reduction variables (e.g., to find the extreme values of an array). COUP would provide a negligible benefit for scalar reductions, as discussed in Section 3.6.1.

	Input set	Commutative operations	Sequential runtime
hist	GRiN [1], 512 bins	32b int add	2720 Mcycles
spmv	rma10 [75]	64b FP add	94 Mcycles
fluidanimate	simlarge [31]	32b FP add	5930 Mcycles
pgrank	Wikipedia (2007) [75]	64b int add	2850 Mcycles
bfs	cage15 [75, 160]	64b OR	5764 Mcycles

Table 3.2: Benchmark characteristics.

Thus, we do not support *min* or *max*. Finally, we support a single word size for bitwise operations, because this suffices to express updates to bitmaps of any size (smaller or larger).

Commutative-update instructions: We add an instruction for each supported operation and data type. Each instruction takes two register inputs, with the address to be updated and the value to apply, and produces no register output. We encode these instructions using x86-64 no-ops that are never emitted by the compiler.

The x86 (TSO) memory model specifies that atomic instructions have an implicit store-load fence [230]; for consistency, we also add an implicit fence to commutative-update instructions. We implement conventional atomic operations and commutative updates using a four-μop sequence: load-linked, execute (in one of the appropriate execution ports), store-conditional, and store-load fence.

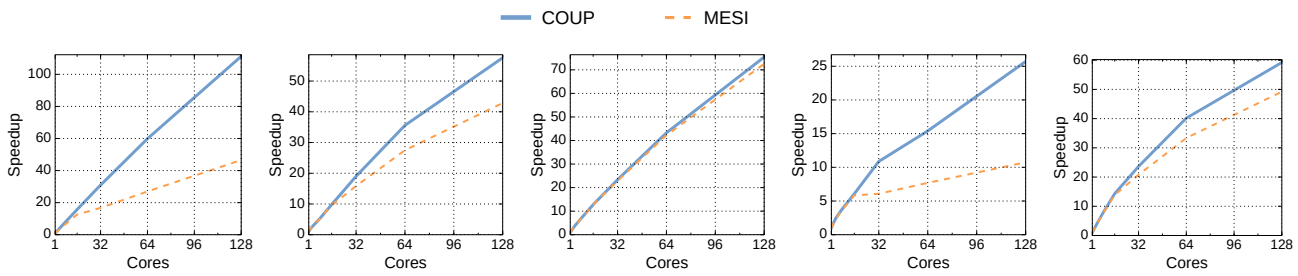
Reduction unit organization: Since functional units for the required operations are relatively simple, we assume a 2-stage pipelined, 256-bit ALU (4× 64-bit lanes). This ALU has a throughput of one full 64-byte cache line per two clock cycles, and a latency of three clock cycles per line. We explore the sensitivity to reduction unit throughput in Section 3.7.5.

Hardware overheads: In summary, our COUP implementation introduces modest overheads:

- Eight additional commutative-update instructions.
- Four bits per line to encode the non-exclusive operation type, either read-only or one of eight commutative-update types (Section 3.5).
- One reduction unit per L3 and L4 bank.

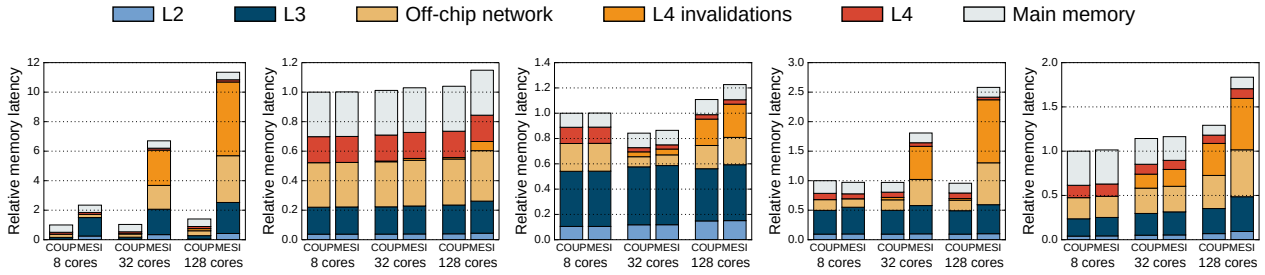
Workloads: We use a set of five multithreaded benchmarks that cover the cases described in Section 3.6:

- **hist** is the TBB-based OpenCV [35] histogramming program (version 2.4.11).
- **spmv** is a sparse matrix-vector multiplication kernel, where the matrix is encoded in compressed sparse column (CSC) format. CSC requires multiple threads to perform scattered additions to the output vector. Other input formats, such as EBE, also cause scattered adds in matrix-vector multiplication [9].
- **fluidanimate**, from the PARSEC suite [31], is a regular iterative algorithm (Section 3.6.1). We optimize the default implementation, which uses locks to guard updates to shared cells, to use atomic operations instead.
- **pgrank** is a PageRank implementation similar to the shared-memory optimized version of Satish et al. [227].
- **bfs** is a parallel breadth-first search algorithm. Our implementation extends PBFS [160] with a visited bit-vector to reduce memory traffic (Section 3.6.2), similar to state-of-the-art approaches [8, 55].



(a) hist (b) spmv (c) fluidanimate (d) pgrank (e) bfs

Figure 3.9: Per-application speedups of COUP and MESI on 1–128 cores (higher is better).



(a) hist (b) spmv (c) fluidanimate (d) pgrank (e) bfs

Figure 3.10: Breakdown of average memory access latency (AMAT) of COUP and MESI on 8, 32, and 128-core systems. AMAT is normalized to COUP's at 8 cores (lower is better).

Table 3.2 details the input sets, commutative-update operations used, and sequential runtime of each benchmark.

All the baseline benchmark implementations use atomic operations. We also compare against a privatization-based variant of `hist` (implemented using TBB reductions) in Section 3.7.3, and develop reference-counting microbenchmarks to compare COUP against SNZI and Refcache in Section 3.7.4.

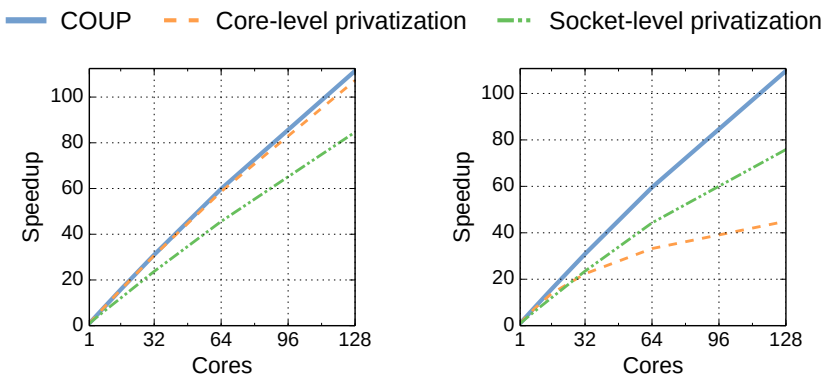
We report results on 1–128 cores. We scale the number of processor and L4 chips on runs with more cores (e.g., 1-core runs use a single processor and L4 chip, 32-core runs use two of each, and so on), which also scales the bandwidth of the memory system and L4 capacity. To achieve statistically significant results, we introduce small amounts of non-determinism as proposed by Alameldeen and Wood [12], and perform enough runs to achieve 95% confidence intervals ≤1%.

3.7.2 Comparison Against Atomic Operations

Figure 3.9 compares the performance and scalability of COUP and a conventional MESI protocol. Each graph shows results for a single application, and each line in the graph shows how performance scales for a particular scheme (MESI or COUP) as the number of cores grows from 1 to 128 (x-axis). All speedup numbers are relative to the runtime of the application on a single core under MESI. Higher numbers are better.

Figure 3.9 shows that COUP always outperforms MESI, often substantially. At 128 cores, COUP outperforms MESI by 2.4× on `hist`, 34% on `spmv`, 4.0% on `fluidanimate`, 2.4× on `pgrank`, and 20% on `bfs`. Moreover, the gap between MESI and COUP often widens as the number of cores grows, showing that COUP has better scalability than MESI.

COUP is especially beneficial for applications where shared data goes through long update-only phases. This is the case with `hist`, `spmv`, and `pgrank`. In `bfs`, where cache lines are constantly moving between U and S states as cores update and check the visited bit-vector (Section 3.6.2), COUP's advantage is lower



(a) 512 bins (b) 16K bins

Figure 3.11: Speedups of `hist` with COUP and both core- and socket-level privatization, using small (512) and large (16 K) numbers of bins.

but still significant. Finally, shared cells in `fluidanimate` experience long read-only and update-only phases, but only a fraction of cells are shared, and shared cells see few updates from neighboring threads on each update-only phase, so COUP provides a small speedup over MESI.

Figure 3.10 gives more insight into these results by showing the breakdown of average memory access latency (AMAT). Each graph shows results for a single application. Each set of two bars shows results for COUP and MESI for a given system size (8, 32, or 128 cores). The height of each bar is the average memory access latency of all loads, stores, and instruction fetches issued from the L1s, normalized to the AMAT that COUP achieves at 8 cores. Each bar is broken down into time spent at the L2, L3, off-chip network, L4, coherence invalidations from the L4, and main memory. This breakdown shows critical-path delays only (e.g., the time spent on invalidations is not the time spent on every invalidation, but the critical-path delay that L4 requests suffer because other sharers need to be invalidated or downgraded).

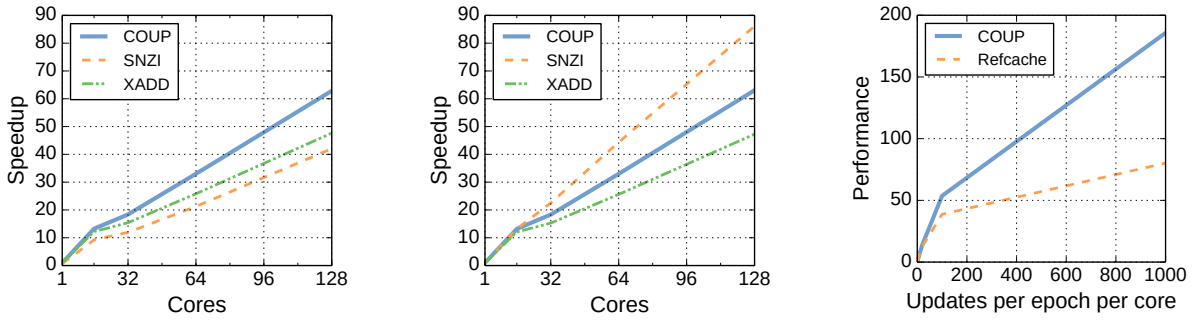
Figure 3.10 shows that COUP substantially reduces AMAT over MESI. At 128 cores, COUP's AMAT is lower than MESI's by 12.6× on `hist`, 10% on `spmv`, 12% on `fluidanimate`, 3.0× on `pgrank`, and 24% on `bfs`. COUP mainly does this by reducing invalidations and serialization. The effect of this reduction on the overall AMAT depends on how the application uses the memory system. For instance, COUP nearly eliminates invalidation traffic in `hist`, `spmv`, and `pgrank`. In `hist` and `pgrank`, invalidations are the dominant contributor to AMAT, so eliminating them has the largest impact. But AMAT in `spmv` is dominated by L4 and main memory accesses, so the overall impact of eliminating invalidations is smaller.

Beyond reducing AMAT, COUP also lowers traffic: at 128 cores, COUP incurs lower off-chip traffic than MESI by a factor of 20.2× on `hist`, 18% on `spmv`, 18% on `fluidanimate`, 4.9× on `pgrank`, and 20% on `bfs`.

Finally, even though COUP's benefits are significant, these benchmarks execute a relatively small fraction of commutative-update instructions: at 128 cores, commutative-update instructions are 1.0% of all executed instructions on `hist`, 2.4% on `spmv`, 0.96% on `fluidanimate`, 4.9% on `pgrank`, and 0.40% on `bfs`. Their impact is significant because, at large core counts, each atomic read-modify-write to a contended memory location can take several hundred cycles.

Case Study: Reduction Variables 3.7.3

All baseline benchmarks use atomic operations instead of privatization. To compare COUP with software privatization, we modify `hist` to make the histogram a reduction variable, and vary the number of bins (elements) in the histogram. We evaluate both core-level privatization, where each thread has its own variable, and socket-level privatization, where each socket has its own variable, shared and updated by



(a) Immediate dealloc, low count (b) Immediate dealloc, high count (c) Delayed dealloc

Figure 3.12: Performance of COUP on reference counting microbenchmarks: (a, b) immediate deallocation and (c) delayed deallocation.

all threads running in that socket using atomic operations. Socket-level privatization seeks to balance the overheads of the fully-shared and fully-privatized implementations.

Figure 3.11 compares the performance and scalability of COUP with core-level and socket-level privatization on `hist`. Figure 3.11a shows that, with a small number of bins, COUP outperforms core-level privatization by 3% and socket-level privatization by 38%. Core-level privatization works well in this case because each thread performs many updates to each histogram bin (128 on average), so reduction overheads are highly amortized.

In contrast, Figure 3.11b shows that, with a large number of bins, COUP outperforms core-level privatization by 2.5× and socket-level privatization by 51%. In this case, core-level privatization is dominated by the cost of reductions, as each thread performs a small number of updates to each histogram bin (2 on average).

Finally, privatization also increases footprint and adds pressure to shared caches. If we grow both the number of bins and the image size (so the number of updates per bin and thread, and thus reduction overheads, stay constant), we see an additional performance degradation of 9% in the core-level privatized version when the aggregate size of all privatized histograms overflow the L3 caches, while COUP does not suffer this degradation.

3.7.4 Case Study: Reference Counting

We use two microbenchmarks to compare COUP's performance on reference counting against the software techniques described in Section 3.6.2. The first microbenchmark models immediate-deallocation schemes, and we use it to compare against a conventional atomic-based implementation and SNZI [85]. The second microbenchmark models delayed-deallocation schemes, and we use it to compare against Refcache [61].

Immediate deallocation: In this microbenchmark, each thread performs a fixed number of increment, decrement, and read operations over a fixed number of shared reference counters. We use 1 to 128 threads, 1 million updates per thread, and 1024 shared counters. On each iteration, a thread selects a random counter and performs either an increment or a decrement and read.

SNZI uses binary trees with as many leaves as threads. The performance of SNZI depends on the number of references per object—a higher number of references causes higher surpluses in leaves and intermediate nodes, and less contention on updates. To capture this effect, we run two variants of this benchmark. In the first variant (low count), each thread keeps only 0 or 1 references per object, while in the second mode (high count), each thread keeps up to five references per object.

To achieve this, in low-count mode, when a thread randomly selects an object, it will always increment its counter if it holds no references to that object, and it will always decrement its counter if it holds one reference. In high-count mode, threads will increment with probability 1.0, 0.7, 0.5, 0.5, 0.3, and 0.0 if they hold 0, 1, 2, 3, 4, and 5 local references to that counter, respectively.

For updates, COUP and XADD use commutative-add and atomic fetch-and-add instructions, respectively.

Figure 3.12a and Figure 3.12b show the results for these experiments. In the low-count variant (Figure 3.12a), SNZI incurs high overhead when counts drop to zero, so both COUP and XADD outperform SNZI (by 50% and 17% at 128 cores, respectively). By contrast, in the high-count variant (Figure 3.12b), SNZI enjoys lower contention and outperforms COUP (by 35% at 128 cores). COUP outperforms XADD in both cases.

We conclude that, in high-contention scenarios, COUP provides the highest performance, but in specific scenarios, software optimizations that exploit application-specific knowledge to avoid contention among reads and updates can outperform COUP. We also note that it may be possible to modify SNZI to take advantage of COUP and combine the advantages of both techniques.

Delayed deallocation: In the delayed-deallocation microbenchmark, 128 threads perform increments and decrements (but not reads) on 100,000 counters. We divide the benchmark into epochs, each with a given number of updates per thread. When they finish an epoch, threads check whether counters are zero, simulating delayed-deallocation periods as in Refcache [61].

Our COUP implementation updates counters with commutative-add instructions and maintains a bitmap with a "modified" bit for each counter. The bitmap is updated with commutative-or instructions. Between epochs, cores use ordinary loads to read the value of marked counters and check whether the counters are zero. Refcache uses a per-thread software cache (a hash table) to maintain the deltas to each modified counter. Threads flush this cache when they finish each epoch.

Figure 3.12c shows the performance COUP and Refcache on the delayed deallocation microbenchmark as the number of updates per epoch (*x*-axis) grows from 1 to 1000 updates per thread and epoch. COUP outperforms refcache across the range, by up to 2.3×.

We conclude that COUP primarily helps delayed-deallocation reference counting by allowing a simpler, lower-overhead implementation to capture the low communication costs of prior software approaches (in this case, using counters and bitmaps instead of hash tables).

Sensitivity to Reduction Unit Throughput 3.7.5

COUP is barely sensitive to reduction unit throughput. We compare the default 256-bit ALU, which has a throughput of one cache line per 2 cycles, with a simpler, unpipelined 64-bit ALU, which has a throughput of one line per 16 cycles. The maximum performance degradation incurred with the slower ALU is 0.88% at 128 cores on `bfs`. Smaller systems incur somewhat lower worst-case degradations (e.g., 0.76% at 64 cores).

Additional Related Work 3.8

Loosely consistent memory (LCM) [156] is a software-controlled coherence protocol built on top of Tempest [217] that allows multiple caches to hold writable copies of the same line. These copies can become incoherent, and software must explicitly reconcile them in a later merge phase. Unlike LCM, COUP preserves cache coherence and transparently merges partial updates, requiring no software intervention.

Moreover, several cache-coherence optimizations reduce the cost of updates, though that is not their primary purpose: self-invalidations, done with either hardware predictors [159] or software protocols [57, 132], remove invalidations from the critical path; adaptive-granularity coherence schemes [152, 285, 292] reduce both false sharing and the amount of dirty data sent on invalidations; and speculation and fast networks can reduce the cost of atomic operations [89]. These schemes are orthogonal to COUP, which could be used in conjunction with them to improve performance.

While we have focused on shared-memory systems, exploiting commutativity is also common with message passing. The BlueGene/L and BlueGene/Q supercomputers feature specialized collective networks that perform reductions completely in hardware, using ALUs embedded in network routers [13, 48]. In contrast to COUP, their main advantage is minimizing the latency of scalar or short reductions across a very large number of nodes.

3.9 Summary

We have presented COUP, a technique that exploits commutativity to reduce the cost of updates in cache-coherent systems. COUP extends conventional coherence protocols to allow multiple caches to simultaneously hold update-only permission to data. We have introduced an implementation of COUP that uses this support to accelerate single-instruction commutative updates. This implementation requires minor hardware changes and, in return, substantially improves the performance of update-heavy applications.

Beyond this specific implementation, a key contribution of COUP is to recognize that it is possible to allow multiple concurrent updates without sacrificing cache coherence or relaxing the consistency model. Thus, COUP attains performance gains without complicating the parallel programming.

COMMTM 4

In this chapter, we present COMMTM, the commutativity-aware hardware transactional memory (HTM). The key idea behind COMMTM is to extend the coherence protocol and conflict detection scheme to allow multiple private caches to simultaneously hold data in a user-defined *reducible* state. Transactions can use *labeled memory operations* to read and update these private, reducible lines locally without triggering conflicts. When another transaction issues an operation that does not commute given the current reducible state and label (i.e., a normal load or store or a labeled operation with a different label), COMMTM transparently performs a user-defined reduction before serving the data. This approach *preserves transactional guarantees*: semantically-commutative operations are reordered to improve performance, but non-commutative operations cannot observe reducible lines with partial updates.

Like COUP, COMMTM modifies the coherence protocol to support new states that do not trigger coherence actions on updates, avoiding conflicts. However, COUP does not work in a transactional context (only for single-instruction atomic updates) and is restricted to a small set of *strictly commutative* operations, i.e., those that produce the same bit pattern when reordered. Instead, COMMTM supports the much broader range of multi-instruction, semantically commutative operations. Moreover, COMMTM shows that there is a symbiotic relationship between semantic commutativity and speculative execution: COMMTM relies on transactions to make commutative multi-instruction sequences atomic, so semantic commutativity would be hard to exploit without speculative execution; and COMMTM accelerates speculative execution much more than COUP does single-instruction commutative updates, since apart from reducing communication, COMMTM avoids conflicts.

Specifically, we make the following contributions:

- We present a basic version of COMMTM (Section 4.2 to Section 4.6) that achieves the same precision as software *semantic locking* [149, 265].
- We then extend COMMTM with *gather requests* (Section 4.7), which allow software to redistribute reducible data among caches, achieving much higher concurrency in important use cases.
- We evaluate COMMTM with microbenchmarks (Section 4.9) and full TM applications (Section 4.10). Microbenchmarks show that COMMTM scales on a variety of commutative operations, such as set insertions, reference counting, ordered puts, and top-K insertions, which allow no concurrency in conventional HTMs. At 128 cores, COMMTM improves full-application performance by up to 3.4×, lowers private cache misses by up to 45%, and reduces or even eliminates transaction aborts.

We first introduce the background of software and hardware transactional memory systems. We then present COMMTM's programming interface and ISA, and a concrete COMMTM implementation that extends an eager-lazy HTM baseline (Section 4.3.1). Finally, we show how to generalize COMMTM to support other coherence protocols and HTM designs.

4.1 Motivation

4.1.1 Semantic Commutativity

Semantic commutativity [265] is a broader concept than strict commutativity exploited in COUP. Semantically commutative operations produce results that are semantically equivalent when reordered, even if the concrete resulting states are different. For example, consider two consecutive insertions of different values `a` and `b` to a set `s` implemented as a linked list. If `s.insert(a)` and `s.insert(b)` are reordered, the concrete representation of these elements in set `s` will be different (either `a` or `b` will be in front). Thus, these set insertions are not strictly commutative. However, since the actual order of elements in `s` does not matter (a set is an unordered data structure), both representations are semantically equivalent, and insertions into sets semantically commute. Other examples include ordered puts and top-K insertions. Such operations are typically implemented as multiple instructions in speculative parallel systems.

4.1.2 Transactional Memory Systems

Many software and hardware techniques, such as transactional memory (TM) or speculative multithreading, rely on speculative execution to parallelize programs with atomic regions. For example, transactional memory lets programmers define transactions, regions of code that are executed atomically. For instance, in the following function, the read-modify-write sequences of account balances are placed in a transaction, which must appear atomic to ensure correctness.

```
void transfer(Account& from, Account& to, int amount) {
  atomic {
    to.balance += amount;
    from.balance -= amount;
  }
}
```

Speculative execution techniques run multiple atomic regions concurrently, and a *conflict detection* technique flags potentially unsafe interleavings of memory accesses (e.g., in transactional memory, those that may violate serializability). Upon a conflict, one or more regions are rolled back and reexecuted to preserve correctness.

Ideally, conflict detection should (1) be *precise*, i.e., allow as many safe interleavings as possible to maximize concurrency, and (2) incur minimal runtime costs. Software and hardware conflict detection techniques typically satisfy one of these properties but sacrifice the other: On the one hand, software techniques can leverage program semantics to be highly precise, but they incur high runtime overheads. On the other hand, hardware techniques incur small overheads, but leave a great amount of concurrency unexploited.

Software conflict detection schemes often exploit semantic commutativity [149, 150, 190, 204, 218, 265]. Most work in this area focuses on techniques that reason about operations to abstract data types. Prior work has proposed a wide variety of conflict detection implementations [110, 149, 204, 218, 265]. Not all commutativity-aware conflict detection schemes are equally precise: simple and general-purpose techniques, such as *semantic locking* [149, 218, 265], flag some semantically-commutative operations as conflicts, while more sophisticated schemes, like gatekeepers [149], incur fewer conflicts but have higher overheads and are often specific to particular patterns.

Specifically, semantic locking [218, 265], also known as abstract locking, generalizes read-write locking schemes (e.g., two-phase locking): transactions can acquire a lock protecting a particular object in one of a number of modes; multiple semantically-commutative methods acquire the lock in a compatible mode, and can proceed concurrently. For instance, the deposit operations to the same account

are commutative. Therefore, multiple deposit operations can hold the lock of the account in *addition* mode at the same time, and can proceed concurrently without blocking one another. Semantic locking requires additional synchronization on the actual accesses to shared data, e.g., logging or reductions. However, such software techniques incur high runtime overheads (e.g. 2-6× in software TM [41]).

Hardware can implement speculative execution at minimal costs by reusing many existing components: private caches to buffer speculative data, and the coherence protocol to detect conflicting speculative accesses. In fact, after an intensive period of research [43, 104, 111, 179, 211, 212], hardware transactional memory has been quickly adopted in commercial processors [189, 261]. Likewise, hardware already has much of the functionality that is necessary to support commutativity. However, exploiting commutativity in hardware conflict detection is still challenging because conventional coherence protocols can reason about only reads and writes. Therefore, commutative updates to the same data, e.g. deposits to the same account, trigger unnecessary conflicts, as shown in Figure 1.2a. This lack of precision can significantly limit concurrency, to the point that prior work finds that commutativity-aware software TM (STM) outperforms hardware TM (HTM) despite its higher overhead [149, 150].

COMMTM bridges the gap between software and hardware and solves this precision-overhead dichotomy. By extending the coherence protocol and the conflict detection scheme, COMMTM provides commutativity support for hardware speculation, and avoids the overheads of software techniques.

COMMTM Programming Interface and ISA 4.2

COMMTM requires simple program changes to exploit commutativity: defining a *reducible state* to avoid conflicts among commutative operations, using *labeled memory accesses* to perform each commutative operation within a transaction, and implementing *user-defined reduction handlers* to merge partial updates to the data.

In this section, we use a very simple example to introduce COMMTM's API: concurrent increments to a shared counter. Counter increments are both strictly and semantically commutative; we later show how COMMTM also supports more involved operations that are semantically commutative but not strictly commutative, such as top-K insertions. Figure 1.2 shows how COMMTM allows multiple transactions to increment the same counter concurrently without triggering conflicts.

User-defined reducible state and labels: COMMTM extends the conventional exclusive and shared read-only states with a reducible state. Lines in this reducible state must be tagged with a *label*. The architecture supports a limited number of labels (e.g., 8). The program should allocate a different label for each set of commutative operations; we discuss how to multiplex these labels in Section 4.5. Each label has an associated, user-defined *identity value*, which may be used to initialize cache lines that enter the reducible state. For example, to implement commutative addition, we allocate one label, `ADD`, to represent deltas to shared counters, and set its identity value to zero.

Labeled load and store instructions: To let the program denote what memory accesses form a commutative operation, COMMTM introduces labeled memory instructions. A labeled load or store simply includes the label of its desired reducible state, but is otherwise identical to a normal memory operation. For instance, commutative addition can be implemented as follows:

```
void add(int* counter, int delta) {
  tx_begin();
  int localValue = load[ADD](counter);
  int newLocalValue = localValue + delta;
  store[ADD](counter, newLocalValue);
  tx_end();
}
```

`load[ADD]` and `store[ADD]` inform the memory system that it may grant reducible permission with the `ADD` label to multiple caches. This way, multiple transactions can perform commutative additions locally and concurrently. Note that this sequence is performed within a transaction to guarantee its atomicity (this code may also be called from another transaction, in which case it is handled as a conventional nested transaction [180]).

User-defined reductions: Finally, COMMTM requires the program to specify a per-label reduction handler that merges reducible-state cache lines. This function takes the address of the cache line and the data from a reducible cache line to merge into it. For example, the reduction operation for addition is:

```
void add_reduce(int* counterLine, int[] deltas) {
  for (int i = 0; i < intsPerCacheLine; i++) {
    int v = load[ADD](counterLine[i]);
    int nv = v + deltas[i];
    store[ADD](counterLine[i], nv);
  }
}
```

Unlike multi-instruction commutative operations done through labeled loads and stores, reduction handlers are *not transactional*. Moreover, to ease their implementation, we restrict the types of accesses they can make. Specifically, while reduction handlers can access arbitrary data with read-only and exclusive permissions, they should not trigger additional reductions (i.e., they cannot access other lines in reducible state).

4.3 COMMTM Implementation

4.3.1 Eager-Lazy HTM Baseline

To make our discussion concrete, we present COMMTM in the context of a specific eager-lazy HTM baseline. We simulate an HTM with eager conflict detection and lazy (buffer-based) version management, as in LTM [15] and Intel's TSX [277]. We assume a multicore system with per-core private L1s and L2s, and a shared L3, as shown in Figure 4.1. Cores buffer speculatively-updated data in the L1 cache; the L2 has non-speculative data only. Evicting the speculative data in L1s causes the transaction to abort. The HTM uses the coherence protocol to detect conflicts eagerly. Transactions are timestamped, and timestamps are used for conflict resolution [179]: on a conflict, the earlier transaction wins, and aborted transactions use randomized backoff to avoid livelock. This conflict resolution scheme frees eager-lazy HTMs from common pathologies [34].

4.3.2 Coherence protocol

COMMTM extends the coherence protocol with an additional state, *user-defined reducible (U)*. For example, Figure 4.2 shows how COMMTM extends MSI with the U state. Lines enter U in response to labeled loads

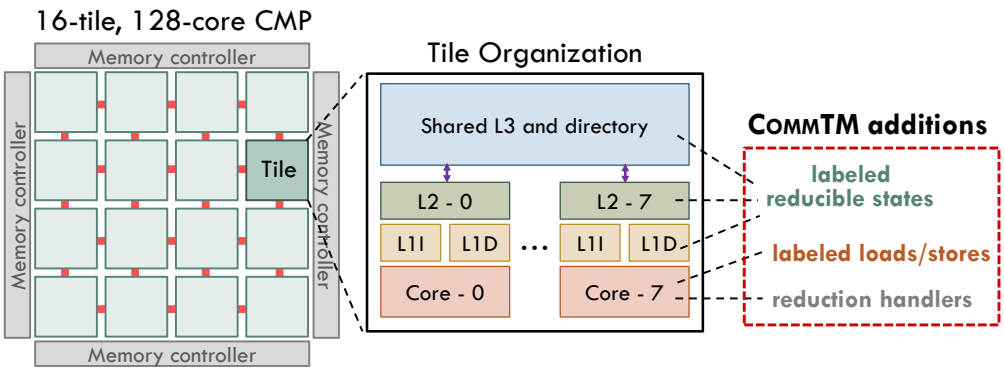


Figure 4.1: Baseline CMP and main CommTM additions.

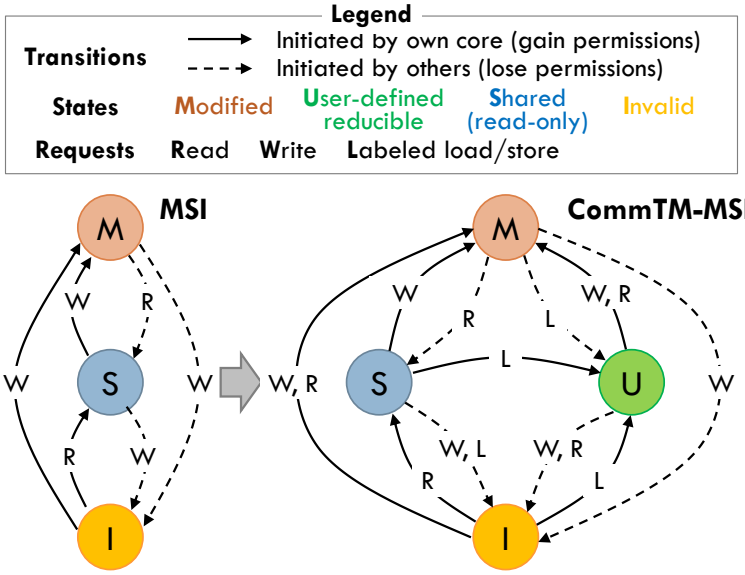
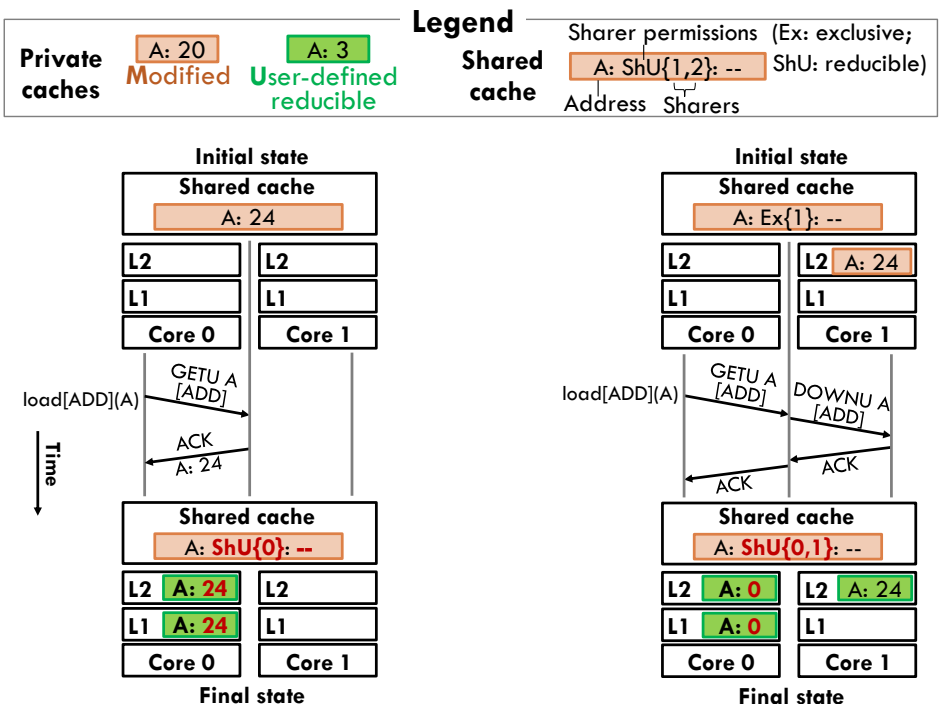


Figure 4.2: State-transition diagrams of MSI and CommTM protocols. For clarity, diagrams omit actions that do not cause a transition (e.g., R requests in S).

and stores, and leave U through reductions. Each U-state line is labeled with the type of reducible data it contains (e.g., `ADD`). Lines in U can satisfy loads and stores whose label matches the line's.

Other states in the original protocol retain similar functionality. For example, in Figure 4.2, M can satisfy all memory requests (conventional and labeled), S can only satisfy conventional loads, and I cannot satisfy any requests. In the rest of the section we will show how lines transition among these states in detail.

CommTM's U state is similar to Coup's update-only state. The transition diagram shown in Figure 4.2 is similar to that of the Coup-extended MUSI protocol shown in Figure 3.3. The major difference is the transition from U state on a read (R): Coup transitions to S state with a reduction performed by the reduction unit, while CommTM transitions to M state as the reduction is performed by the requesting core. CommTM requires substantially different support from Coup in nearly all other aspects: whereas Coup requires new update-only instructions for each commutative operation, CommTM allows programs to implement arbitrary commutative operations, exploiting transactional memory to make them atomic; whereas Coup implements fixed-function reduction units, CommTM allows arbitrary reduction functions; and whereas Coup focuses on reducing communication in a non-transactional context, CommTM reduces both transactional conflicts and communication.



(a) No other sharers

(b) Downgrade from M to U

Figure 4.3: Serving labeled memory accesses: (a) the first GETU requester obtains the data; and (b) another cache with the line in M is downgraded to U and retains the data, while the requester initializes the line with the identity value. Each diagram shows the initial and final states in the shared and private caches.

4.3.3 Transactional execution

Labeled memory operations within transactions cause lines to enter the U state. We first discuss of permissions change in the absence of transactional conflicts, then explain how conflict detection changes.

On a labeled request to a line with invalid or read-only permissions, the cache issues a GETU request and receives the line in U. There are five possible cases:

1. If no other private cache has the line, the directory serves the data directly, as shown in Figure 4.3a.
2. If there are one or more sharers in S, the directory invalidates them, then serves the data.
3. If there are one or more sharers in U with a different label from the request's, the directory asks them to forward the data to the requesting core, which performs a reduction to produce the data. Reductions are discussed in detail in Section 4.3.4.
4. If there are one or more sharers in U with the same label, the directory grants U permission, but does not serve any data.
5. If there is an exclusive sharer in M, the directory downgrades that line to U and grants U to the requester without serving any data, as shown in Figure 4.3b.

In cases 1–3, the requester receives both U permission and the data; in cases 4 and 5, the requester does not receive any data, and instead initializes its local line with the user-defined identity element (e.g., zeros for ADD). Labeled operations must be aware that data may be scattered across multiple caches. In all cases, COMMTM preserves a key invariant: reducing the private versions of the line produces the right readable value.

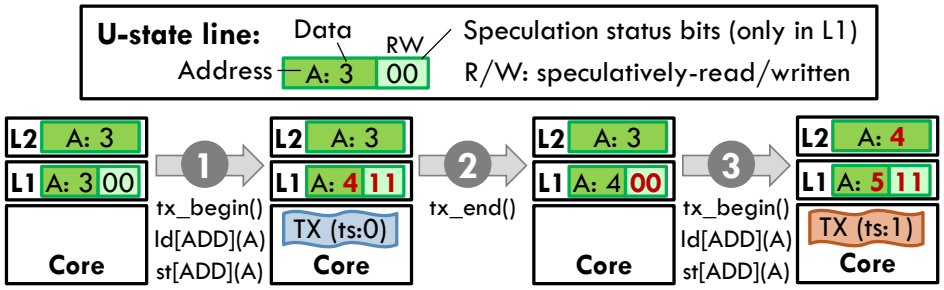


Figure 4.4: Value management for U-state lines is similar to M. L1 tag bits record whether the line is speculatively read or written (using the state and label to infer whether from labeled or unlabeled instructions). Upon commit, spec-R/W bits are reset to zero. Before being written by another transaction, dirty U-state lines are written back to the L2.

Speculative value management: Value management for lines in U that are modified is nearly identical to that of lines in M. Figure 4.4 shows how a line in U is read, modified, and, in the absence of conflicts, committed: ① Both normal and labeled writes are buffered in the L1 cache, and non-speculative values are stored in the private L2. ② When the transaction commits, all dirty lines in the L1 are marked as non-speculative. ③ Before a dirty line in the L1 is speculatively written by a new transaction, its value is forwarded to the L2. Thus, if the transaction is aborted, its speculative updates to data in both M and U can be safely discarded, as the L2 contains the correct value.

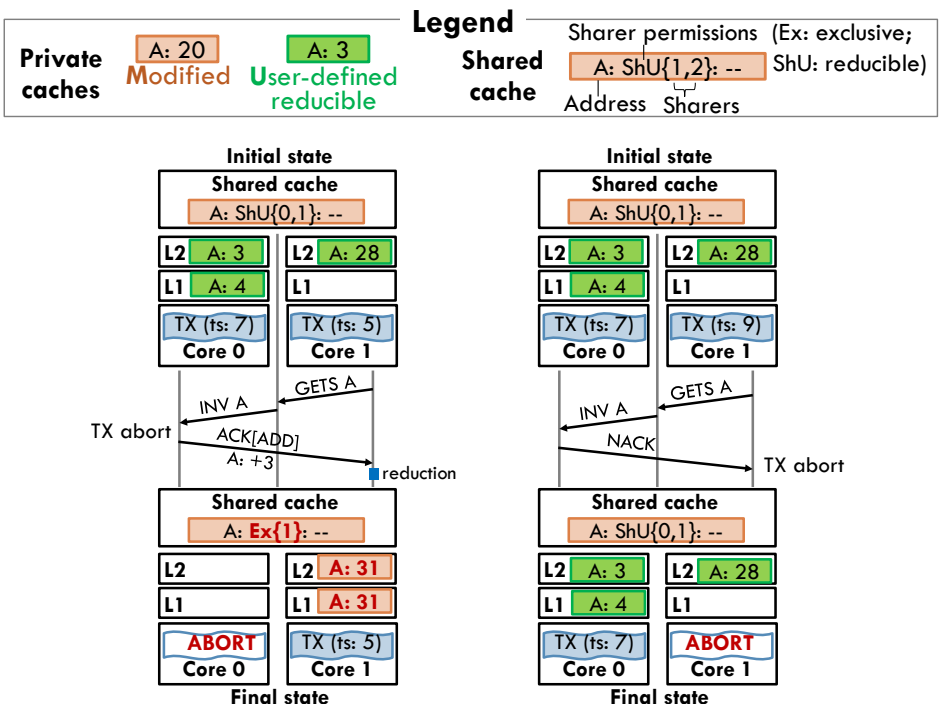
Conflict detection and resolution: COMMTM leverages the coherence protocol to detect conflicts. In our baseline, conflicts are triggered by invalidation and downgrade requests to lines read or modified by the current transaction (i.e., lines in the transaction's read- or write-sets). Similarly, in COMMTM, invalidations to lines that have received a labeled operation from the current transaction trigger a conflict. We call this set of lines transaction's *labeled set*. We leverage the existing L1's status bits to track the labeled set, as shown in Figure 4.4.

COMMTM is orthogonal to the conflict resolution protocol. We leverage our baseline's timestamp-based resolution approach: each transaction is assigned an unique timestamp, and requests from each transaction include its timestamp. On an invalidation to a line in the transaction's read, write, *or labeled set*, the core compares its transaction's timestamp and the requester's. If the receiving transaction is younger (i.e., has a higher timestamp), it honors the invalidation request and aborts; if it is older than the requester, it replies with a `NACK`, which causes the requester to abort. Figure 4.5 shows both of these cases in detail for a line in the labeled set.

Reductions 4.3.4

COMMTM performs reductions transparently to satisfy non-commutative requests. There is a wide range of implementation choices for reductions, as well as important considerations for deadlock avoidance.

We choose to perform reductions at the core that issues the reduction-triggering request. Specifically, each core features a *shadow hardware thread* dedicated to perform reductions. Figure 4.6 shows the steps of a reduction in detail: ① When the directory receives a reduction-triggering request, it sends invalidation requests to all the cores with U-state permissions. ② Each of the cores receiving the invalidation forwards the line to the requester. ③ When each forwarded line arrives at the requester, the shadow thread runs the reduction handler, which merges it with the current line (if the requester does not have the line in U yet, it transitions to U on the first forwarded line it receives). ④ After all lines



(a) INV from lower-timestamp transaction

(b) INV from higher-timestamp transaction

Figure 4.5: Core 0 receives an invalidation request to a U-state line in its transaction's labeled set. (a) if requester has a lower timestamp, abort and forward data; and (b) if requester has a higher timestamp, NACK invalidation.

have been received and reduced, the requester transitions to M, ④ notifies the directory, and ⑤ serves the original request.

Dedicating a helper hardware context to reductions ensures that they are performed quickly, but adds implementation cost. Alternatively, we could handle reductions through user-level interrupts of the main thread [170, 225, 270], or use a low-performance helper core [17, 51].

NACKed reductions: When a reduction happens to a line that has been speculatively updated by a transaction, the core receiving the invalidation may NACK the request, as shown in Figure 4.5b. In this case, the requesting core still reduces the values it receives, but aborts its transaction afterwards, retaining its data in the U state. When re-executed, the transaction will retry the reduction, and will eventually succeed thanks to timestamp-based conflict resolution.

For simplicity, non-speculative requests have no timestamp and cannot be NACKed. Finally, even though the request they seek to serve may come from within a transaction, *reductions are not speculative*: reduction handlers always operate on non-speculative data, and have no atomicity guarantees. Transactional reductions would be more complex, and they are unnecessary in all the use cases we study (Section 4.9 and Section 4.10).

Deadlock avoidance: Because the memory request that triggers the reduction blocks until the reduction is done, and reduction handlers may themselves issue memory accesses, there are subtle corner cases that may lead to deadlock and must be addressed. First, as mentioned in Section 4.2, we enforce

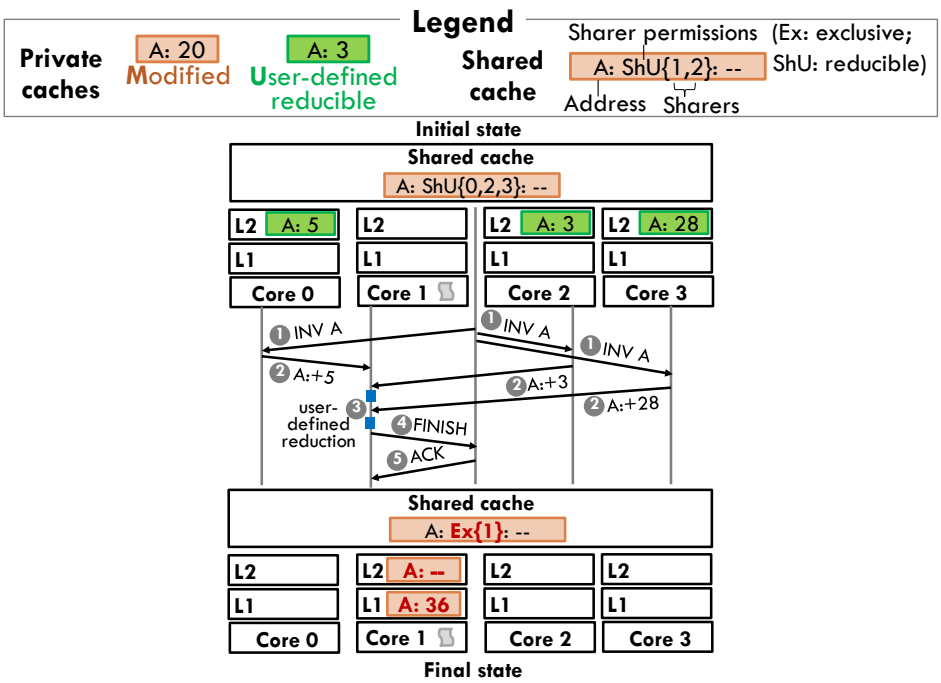


Figure 4.6: Core 0 issues unlabeled or differently-labeled request, causing a full reduction of A's U-state data, held in several private caches.

that reduction handlers cannot trigger reductions themselves (this restriction is easy to satisfy in all the reduction handlers we study). Second, to avoid a protocol deadlock caused by reductions, we dedicate an extra virtual network for forwarded U-state data. This adds moderate buffering requirement to on-chip network routers [200], which must already support 3-6 virtual networks in conventional protocols [32, 184, 232]. Third, we reserve a way in all cache levels for data with permissions other than U. Misses from reductions always fill data in that way, which ensures that they will not evict data in U, which would necessitate a reduction.

With these provisos, memory accesses caused by reductions cannot cause a cyclic dependence with the access they are blocking, avoiding deadlock. We should note that both the corner cases and the deadlock-avoidance strategies we adopt are similar to those in architectures with hardware support for active messages, where these topics are well studied [5, 170, 225, 259] (a forward response triggered by a reduction is similar to an active message).

Handling unlabeled operations to speculatively-modified labeled data: Finally, COMMTM must handle a transaction that accesses the same data through labeled and unlabeled operations (e.g., it first adds a value to a shared counter, and then reads it). Suppose that an unlabeled access to data in U causes a reduction (i.e., if the core's U-state line was not the only one in the system). If the data was speculatively modified by our own transaction, we cannot simply incorporate this data to the reduction, as the transaction may abort, leaving COMMTM unable to reconstruct the non-speculative value of the data. For simplicity, in this case we abort the transaction and perform the reduction with the non-speculative state, re-fetched from the core's L2. When restarted, labeled loads and stores are performed as conventional loads and stores, so the transaction does not encounter this case again. Though we could avoid this abort through more sophisticated schemes (e.g., performing speculative and non-speculative reductions), we do not observe this behavior in any of our use cases.

4.3.5 Evictions

Evictions of lines in U from private caches are handled as follows: if no other private caches have U permissions for the line apart from the one that initiates the eviction, the directory treats this as a normal dirty writeback. When there are other sharers, the directory forwards the data to one of the sharers, chosen at random, which reduces it with its local line. If the chosen core is performing a transaction that touches this data, for simplicity, the transaction is aborted. Finally, evictions of lines in U from the shared cache cause a reduction at one of the cores sharing the line. Since the last-level cache is inclusive, this eviction aborts all transactions that have accessed the line.

4.4 Putting it all Together: Overheads

In summary, our COMMTM implementation introduces moderate hardware overheads:

- Labeled load and store instructions in ISA and cores.
- Cache at all levels need to store per-tag label bits. Supporting eight labels requires 3 bits/line, introducing 0.6% area overhead for caches with 64-byte lines.
- Extended coherence protocol and cache controllers. While we have not verified COMMTM's protocol extensions, they are similar to COUP's, which has reasonable verification complexity (requiring only 1–5 transient states by merging S and U)
- One extra virtual network for forwarded U data, which adds few KBs of router buffers across the system [73].
- One shadow hardware thread per core to perform reductions. In principle, this is the most expensive addition (an extra thread increases core area by about 5% [113]). However, commercial processors already support multiple hardware threads, and the shadow thread can be used as a normal thread if the application does not benefit from COMMTM.

4.5 Generalizing COMMTM

COMMTM can be applied to other contexts beyond our particular implementation.

Other protocols: While we have used MSI for simplicity, COMMTM can easily extend other invalidation-based protocols, such as MESI or MOESI, with the U state. In fact, we use and extend MESI in our evaluation.

Multiplexing labels: Large applications with many data types may have more semantically-commutative operations than hardware provides. In this case, we can assign the same label to two or more operations under two conditions. First, it should not be possible for both commutative operations to access the same data. There are many cases where this is naturally guaranteed, for instance, on operations on different types (e.g., insertions into sets and lists). Second, U-state lines need to have enough information (e.g., the data structure's type) to allow reduction handlers to perform the right operation. This allows COMMTM to scale to large applications with a small number of labels in hardware.

Lazy conflict detection: While we focus on eager conflict detection, COMMTM applies to HTMs with lazy (commit-time) conflict detection, such as TCC [44, 104] or Bulk [43, 205]. This would simply require acquiring lines in S or U without restrictions (triggering non-speculative reductions if needed,

but without flagging conflicts), holding speculative updates (both commutative and non-commutative), and making them public when the transaction commits. Commits then abort all executing transactions with non-commutative updates. For example, a transaction that triggers a reduction and then commits would abort all transactions that accessed the line while in U, but transactions that read and update the line while in U would not abort each other.

Other contexts: COMMTM's techniques could be used in other contexts beyond TM where speculative execution is required, e.g., thread-level speculation.

COMMTM vs Semantic Locking 4.6

Just as eager conflict detection is the hardware counterpart to two-phase locking [18, 112], COMMTM as described so far is the hardware counterpart to semantic locking (Section 4.1). In semantic locking, each lock has a number of modes, and transactions try to acquire the lock in a given mode. Multiple transactions can acquire the lock in the same mode, accessing and updating the data it protects concurrently [149] (with some other synchronization to arbitrate low-level accesses, e.g., logging updates and performing reductions later). An attempt to acquire the lock in a different mode triggers a conflict. Each label in COMMTM can be seen as a locking mode, and just like reads and writes implicitly acquire read and write locks to the cache line, labeled accesses implicitly acquire the lock in the mode specified by the label, triggering conflicts if needed. Furthermore, COMMTM is architected to reduce communication by holding commutative updates to the line in private caches.

Avoiding Needless Reductions with Gather Requests 4.7

While semantic locking is general, not all semantically-commutative operations are amenable to semantic locking, and more sophisticated software conflict detectors allow more operations to commute [149]. Similarly, we now extend COMMTM to allow more concurrency than semantic locking. The key idea is that many operations are *conditionally commutative*: they only commute when the reducible data they operate on meets some conditions. With COMMTM as presented so far, these conditions require normal reads, resulting in frequent reductions that limit concurrency. To solve this problem, we introduce *gather requests*, which allow moving partial updates to the same data across different private caches *without leaving the reducible state*.

Motivation: Consider a *bounded non-negative counter* that supports `increment` and `decrement` operations. `increment` always succeeds, but `decrement` returns a failure when the initial value of the counter is already zero. `increment` always commutes, but `decrement` only commutes if the counter has a positive value. Bounded counters have many use cases, such as reference counting and resizable data structures.

In COMMTM, we can exploit the fact that if the local value is positive, the global value must be positive. In this case, `decrement` can safely decrement the local value. However, if the local value is zero, `decrement` must perform a reduction to check whether the value has reached zero, as shown in this implementation:

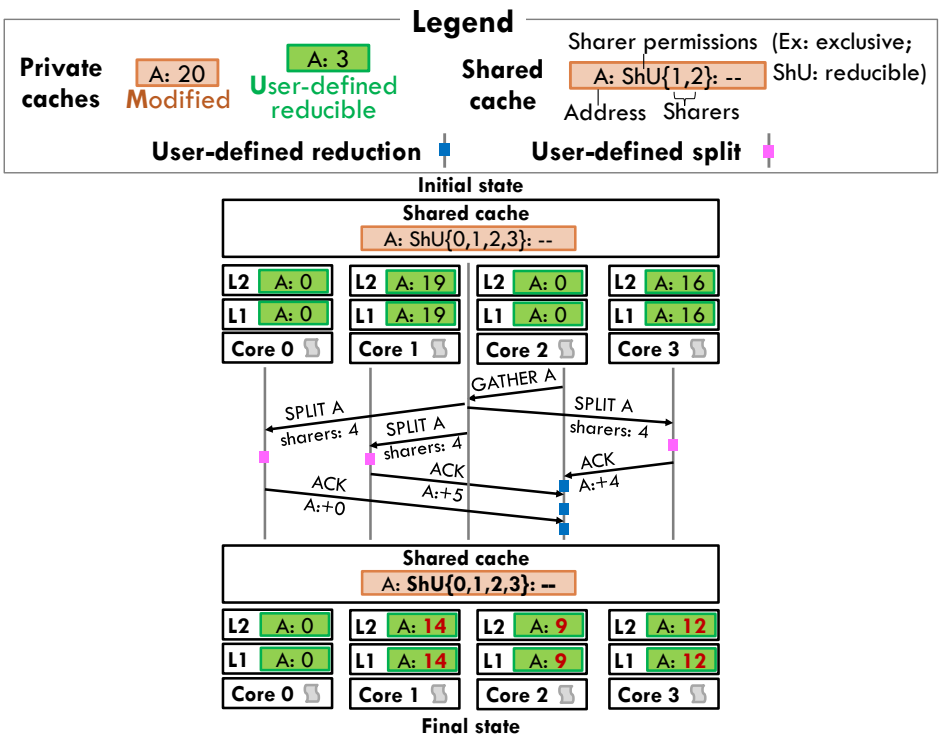


Figure 4.7: Gather requests collect and reduce U-state data from other caches. In this example, core 2 initiates a gather to satisfy a local decrement. User-defined splitters at other cores donate part of their local deltas to core 2. For instance, core 3 splits its initial value, 16, into 12, which it retains, and 4, which it donates.

```
bool decrement(int* counter) {
  tx_begin();
  int value = load[ADD](counter);
  if (value == 0) { // trigger a reduction
    if (load(counter) == 0) {
      tx_end();
      return false;
    }
  }
  store[ADD](counter, value - 1);
  tx_end();
  return true;
}
```

With frequent decrements, reductions will serialize execution even when the actual value of the counter is far greater than zero. Gather requests avoid this by allowing programs to observe partial updates in other caches and redistribute them without leaving U.

Gather requests: Figure 4.7 depicts the steps of a gather request in detail. Gather requests are initiated by a new instruction, `load_gather`, which is similar to a labeled load. If the requester's line is in U, `load_gather` issues a gather request to the directory and reduces forwarded data from other sharers before returning the value.

The directory forwards the gather request to each (U-state) sharer. The core executes a *user-defined*

splitter, a function analogous to a reduction handler, that inspects its local value and sends a part of it to the requester. In our implementation, the directory forwards the number of sharers in gather requests, which splitters can use to rebalance the data appropriately.

Splitters reuse all the machinery of reduction handlers: they run on the shadow thread, are non-speculative, and split requests may trigger conflicts if their address was speculatively accessed.

Our bounded counter example can use gather requests as follows. First, we modify the `decrement` operation to use `load_gather`:

```
bool decrement(int* counter) {
  tx_begin();
  int value = load[ADD](counter);
  if (value == 0) {
    value = load_gather[ADD](counter);
    if (value == 0)
      if (load(counter) == 0) {
        tx_end();
        return false;
      }
  }
  store[ADD](counter, value - 1);
  tx_end();
  return true;
}
```

Second, we implement a user-defined splitter that gives a fraction $1/numSharers$ of its counter values, which, over time, will maintain a balanced distribution of values:

```
void add_split(int* counterLine, int* fwdLine, int numSharers) {
  for (int i = 0; i < intsPerCacheLine; i++) {
    int value = load[ADD](counterLine[i]);
    int donation = ceil(v / numSharers);
    fwdLine[i] = donation;
    store[ADD](counterLine[i], v - donation);
  }
}
```

Figure 4.7 shows how a gather request rebalances the data and allows a `decrement` operation to proceed while maintaining lines in U. Note how, after the gather request, the requester's local value (9) allows it to perform successive `decrements` locally. In general, we observe that, although gather requests incur global traffic and may cause conflicts, they are rare, so their cost is amortized across multiple operations.

There are many options to enhance the expressiveness of gather operations. For example, we could enhance `load_gather` to query a subset of sharers, or to provide user-defined arguments to splitters. However, we have not found a need for these mechanisms for the operations we evaluate. We leave an in-depth exploration of these and other mechanisms to enhance COMMTM's precision to future work.

Experimental Methodology 4.8

As in Chapter 3, we perform microarchitectural, execution-driven simulation using zsim. We evaluate a 16-tile CMP with 128 simple cores and a three-level memory hierarchy, shown in Figure 4.1, with parameters given in Table 4.1. Each core has private L1s and a private L2, and all cores share a banked L3 cache with an in-cache directory.

Cores	128 cores, x86-64 ISA, 2.4 GHz, IPC-1 except on L1 misses
L1 caches	32 KB, private per-core, 8-way set-associative, split D/I
L2 caches	128 KB, private per-core, 8-way set-associative, inclusive, 6-cycle latency
L3 cache	64 MB, fully shared, 16 4 MB banks, 16-way set-associative, inclusive, 15-cycle bank latency, in-cache directory
Coherence	MESI/COMMTM, 64 B lines, no silent drops
NoC	4×4 mesh, 2-cycle routers, 1-cycle 256-bit links
Main mem	4 controllers, 136-cycle latency

Table 4.1: Configuration of the simulated system.

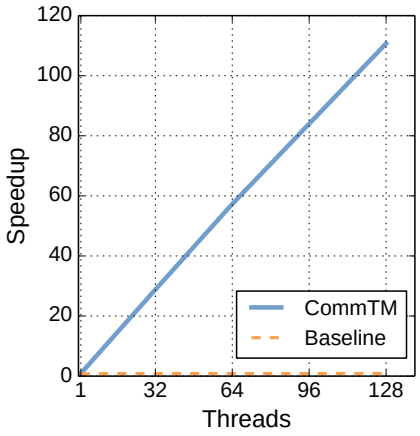


Figure 4.8: Speedup of counter microbenchmarks.

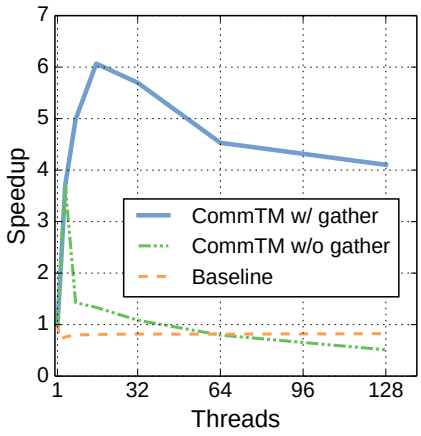


Figure 4.9: Speedup of reference-counting microbenchmark.

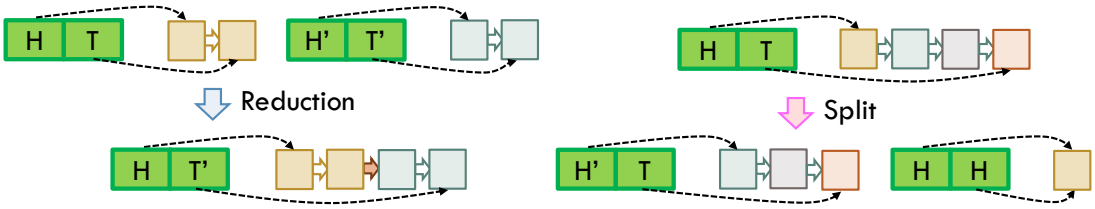
We compare the baseline HTM and COMMTM. Both HTMs use Intel TSX [277] as the programming interface, but do not use the software fallback path, which the conflict resolution protocol makes unnecessary. We add encodings for `labeled_load`, `labeled_store`, and `load_gather`, with labels embedded in the instructions.

We evaluate COMMTM under microbenchmarks (introduced in Section 4.9) and full-blown TM applications (discussed in Section 4.10). We run each benchmark to completion, and report results for its parallel region. To achieve statistically significant results, we introduce small amounts of non-determinism [12], and perform enough runs to achieve 95% confidence intervals $\leq 1\%$ on all results.

4.9 COMMTM on Microbenchmarks

We use microbenchmarks to explore COMMTM's capabilities and its impact on update-heavy operations.

Counter increments: In this microbenchmark, threads perform 10 million increments to a single counter, implemented as presented in Section 4.3. Figure 4.8 shows that COMMTM achieves linear scalability, while the baseline HTM serializes all transactions. While counters are our simplest case, prior work reports that counter updates are a major cause of aborts in real applications [63, 221].



(a) Reducing a list descriptor (b) Splitting a list descriptor

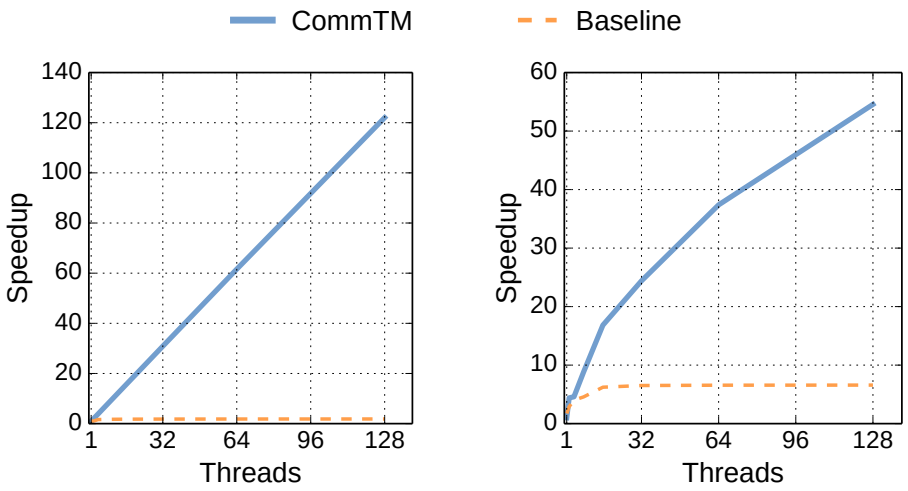
Figure 4.10: A linked-list descriptor contains its head and tail pointers, and can be shared in U states by multiple caches. Each U-state copy represents a partial linked list. A reduction merges all partial lists and generate the resulting descriptor, and a split divides the partial list into two: one only containing the previous head element and the other containing the rest.

Reference counting: We implement a reference counter using the non-negative bounded counter described in Section 4.7, with and without gather requests. Threads acquire and release 1 million references in total, incrementing and decrementing the counter. Each thread starts with three references to the object and holds up to five references. Threads behave probabilistically: each thread increments the counter with probability 1.0, 0.7, 0.5, 0.5, 0.3, and 0.0 if it holds 0, 1, 2, 3, 4, and 5 local references, respectively, and decrements it otherwise. Figure 4.9 shows that the baseline HTM achieves no speedup, and COMMTM without gather requests provides some speedup with few threads, but frequent reductions caused by threads having zero in their U-state line result in serialized transactions. By contrast, COMMTM with gather requests scales to 3.7× at 128 threads. The sub-linear scalability is due to more frequent gather requests and splits at high thread counts.

Linked lists: In this microbenchmark, threads enqueue and dequeue elements from a singly-linked list. When order is unimportant (e.g., if the list is used as a set, a hash table bucket, or a work-sharing queue), these operations are semantically (but not strictly) commutative. Figure 4.10a shows how COMMTM makes these operations concurrent. Only the *descriptor* of a linked list, which contains its head and tail pointers, is accessed with labeled loads and stores (accesses to elements use normal loads and stores). This way, each reducible, local descriptor has its own tail pointer, and threads can enqueue/dequeue elements locally. Figure 4.10a shows how the user-defined reduction handler merges two linked-list descriptors. Dequeues use `load_gather` if their local descriptor is empty, and each splitter donates the head element of its local list, as shown in Figure 4.10b.

Figure 4.11 compares the baseline HTM and COMMTM. In the baseline HTM, to avoid false sharing, head and tail pointers are allocated on different cache lines. Threads perform 10 million operations: all enqueues in Figure 4.11a, or 50% enqueues and 50% dequeues (randomly interleaved) in Figure 4.11b. The baseline HTM scales poorly in both cases, while COMMTM scales near-linearly on enqueues, and by 55× on mixed enqueues/dequeues (limited again by frequent gathers).

Ordered puts: Ordered puts or priority updates are frequent in databases [190] and are key in challenging parallel algorithms [233]. This semantically-commutative operation replaces an existing key-value pair with a new input pair if the new pair has a lower key. In COMMTM, we simply access the key-value pair with a labeled accesses, and define a reduction handler that merges key-value pairs by keeping the lowest one. Threads perform 10 million ordered puts using randomly-generated 64-bit keys and values. These fit within a cache line, but arbitrarily large key-value pairs are possible by using indirection (i.e., keeping pointers to the key and value in the reducible line). Figure 4.13a shows that COMMTM scales near-linearly, while the baseline is 3.8× slower (in this case, the baseline scales to 31× because only smaller keys cause conflicting writes).



(a) 100% enqueues (b) 50% enqueues, 50% dequeues

Figure 4.11: Speedup of linked list microbenchmark under baseline HTM and CommTM.

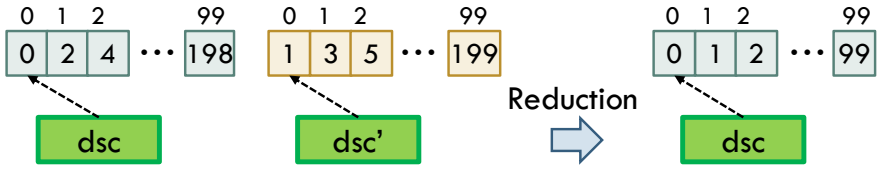


Figure 4.12: A top-K set descriptor with $K = 100$.

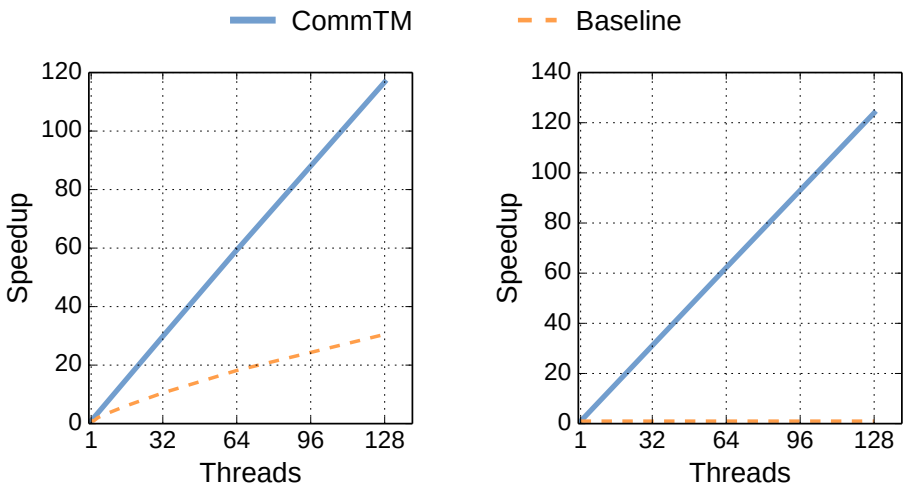
Top-K: A *top-K set*, common in databases, contains the K highest elements of a set [190]. We implement insertions to a top-K set similarly to the linked-list: a descriptor contains a pointer to the top-K data (stored as a heap), and only the descriptor uses reducible states. Threads build up local top-K heaps, and reads trigger a reduction that merges all local heaps, as shown in Figure 4.12.

Figure 4.13b shows the performance of inserting 10 million elements to a top-1000 set. While the baseline HTM suffers significant serialization introduced by unnecessary read-write dependencies, CommTM scales top-K set insertions linearly, yielding 124× speedup at 128 threads.

4.10 CommTM on Full Applications

We evaluate CommTM on several TM benchmarks: `boruvka` [149], and `genome`, `kmeans`, `ssca2`, and `vacation` from STAMP [178]. Table 4.2 details their input sets and main characteristics. `boruvka` computes the minimum spanning tree of a graph. It utilizes several commutative operations: `OPUT` to record the minimum-weight edges connecting separate graph components, `MIN` to union two components, `MAX` to mark edges added to the minimum spanning tree, and `ADD` to calculate the weight of the resulting tree. `kmeans` performs commutative additions to shared cluster centroids. `ssca2` spends little time in commutative updates to shared, global graph metadata. We compile `genome` and `vacation` with resizable hash tables (similar to Blundell el al.[33]), which use conditionally-commutative updates to a bounded counter to determine when to resize.

Figure 4.14 compares the performance and scalability of CommTM and the baseline HTM. Each graph shows the speedups of the baseline HTM and CommTM for a single application from 1–128 threads (x-axis). As before, all speedups are relative to the performance of a sequential execution in the baseline HTM. Figure 4.14 shows that CommTM always outperforms baseline HTM, often significantly. At 128



(a) Ordered put (b) Top K

Figure 4.13: Speedups of (a) an ordered put benchmark, and (b) a top-K insertion benchmark.

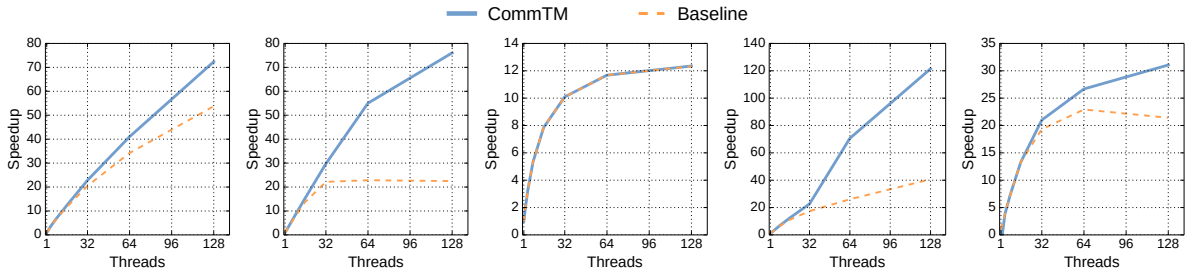
	Input set	Uses gather?	Commutative operations
boruvka	usroads [75]	✗	Updating min-weight edges (64b-key `OPUT`) Unioning components (64b `MIN`) Marking edges (64b `MAX`) Calculating weight of MST (64b `ADD`)
kmeans	-m15 -n15 -t0.05 -i random-n16384-d24-c16 [178]	✗	Updating cluster centers (32b `ADD`, 32b `FP ADD`)
ssca2	-s16 -i1.0 -u1.0 -l9 -p9 [178]	✗	Modifying global information for a graph (32b `ADD`)
genome	-g4096 -s64 -n640000 [178]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)
vacation	-n4 -q60 -u90 -r32768 -t8192 [178]	✓	Remaining-space counter of a resizable hash table (bounded 64b `ADD`)

Table 4.2: Benchmark characteristics.

threads, COMMTM outperforms the baseline by 35% on `boruvka`, 3.4× on `kmeans`, 0.2% on `ssca2`, 3.0× on `genome`, and 45% on `vacation`. Moreover, the gap between baseline HTM and COMMTM often widens as the number of threads grows, demonstrating the better scalability of COMMTM.

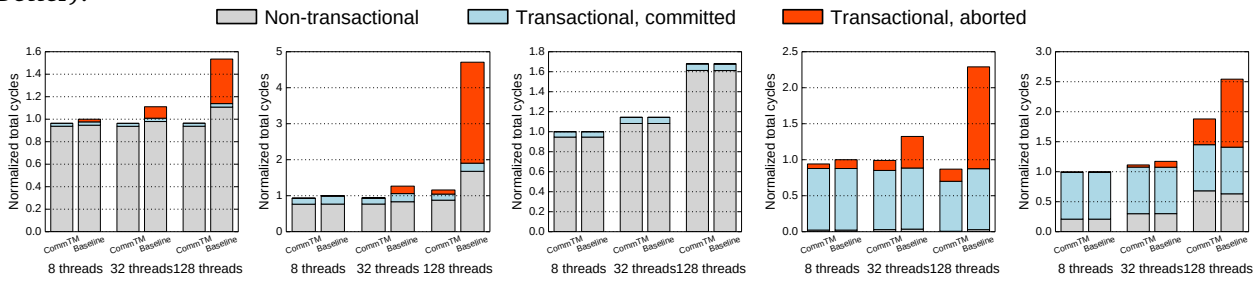
COMMTM is especially beneficial on update-heavy applications. For instance, `kmeans` introduces a large number of commutative updates within transactions. With conventional HTMs, these updates must be serialized. Thus, as the number of threads increases, serialized updates bottleneck the whole application. COMMTM, however, makes these updates local and concurrent, achieving significant speedup. As the update contention decreases, the benefit of COMMTM decreases. For applications such as `ssca2`, where there is little concurrent modification to shared data, COMMTM yields a negligible improvement over the baseline HTM.

Figure 4.15 gives more insight into these results by showing the breakdown of total cycles spent by all threads for each application. Each cycle is either non-transactional or transactional, and transactional cycles are divided into useful (committed) and wasted (aborted) cycles. Each graph shows the breakdown of cycles for both COMMTM and the baseline HTM on 8, 32, and 128 threads for a single application. Cycles are normalized to the baseline's at 8 threads. Lower bars are better.



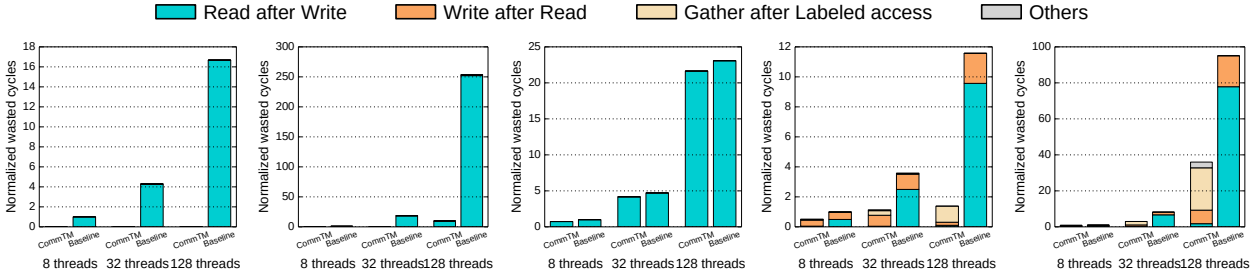
(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

Figure 4.14: Per-application speedups of CommTM and baseline HTM on 1–128 threads (higher is better).



(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

Figure 4.15: Breakdown of total cycles for CommTM and baseline HTM for 8, 32, and 128 threads (lower is better).



(a) boruvka (b) kmeans (c) ssca2 (d) genome (e) vacation

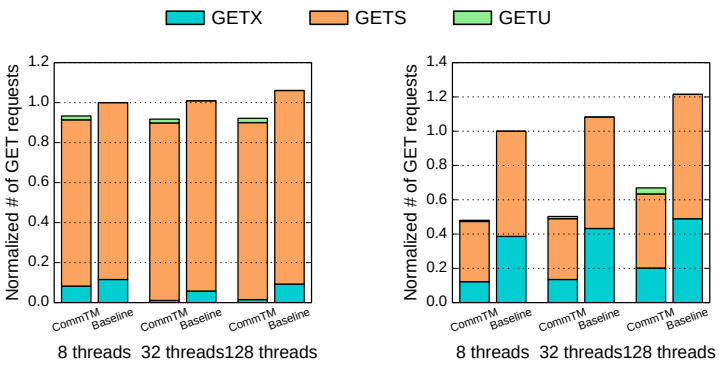
Figure 4.16: Breakdown of wasted cycles for CommTM and baseline HTM for 8, 32, and 128 threads (lower is better).

Figure 4.15 shows that CommTM substantially reduces wasted transactional cycles. At 128 threads, CommTM's wasted cycles is lower than the baseline's by 25× on `kmeans`, 6.6% on `ssca2`. 8.3× on `genome`, and 2.6× on `vacation`. In `boruvka`, CommTM eliminates all aborts and hence eliminates all wasted transactional cycles.

The breakdown of total cycles explains why CommTM has little impact on performance of `ssca2`: contention is rare and therefore only a small fraction of cycles are spent on aborted transactions.

Figure 4.16 further details the cause of wasted cycles. In the baseline HTM, wasted cycles are almost always caused by read-after-write dependency violations. For applications with ample semantic commutativity, such as `boruvka` and `kmeans`, most of these dependencies are superfluous and CommTM avoids them entirely.

Beyond improving concurrency, CommTM also reduces traffic, as applications with significant data reuse benefit substantially from buffering updates in private caches. Figure 4.17 shows the breakdown of `GET` requests between L2s and L3 for `boruvka` and `kmeans`, the two applications with a significant



(a) boruvka (b) kmeans

Figure 4.17: Breakdown of total number of GET requests between L2s and L3 for COMMTM and conventional HTM on 8, 32 and 128 threads (lower is better).

reduction in traffic. At 128 threads, COMMTM reduces L3 GET requests by 13% on boruvka and 45% on kmeans. This also explains why non-transactional cycles are lower in Figure 4.15 (15% lower on boruvka and 48% on kmeans).

Finally, though COMMTM improves performance significantly, labeled memory operations are relatively rare. At 128 threads, the fraction of all labeled instructions, including labeled loads, stores and gather requests, over all executed instructions are 0.13% on boruvka, 1.2% on kmeans, 0.000059% on ssca2, 0.042% on genome, and 0.057% on vacation. Though rare, their impact is substantial: on conventional HTMs, these operations cause conflicts that abort whole transactions, which include many other (conventional) instructions, wasting a large amount of cycles.

Additional Related Work 4.11

Prior work in hardware speculation, especially HTM, has proposed a wide set of techniques to reduce the number of conflicts and their impact. These techniques are orthogonal to COMMTM, as they do not leverage commutativity, and detect conflicts through reads and writes.

Several HTMs, such as DATM [213], SONTM [18], Wait-n-GoTM [119], and OmniOrder [206], reduce aborts by letting transactions continue execution after they conflict and trying to commit them in the order imposed by the data dependence that caused the conflict. These designs can substantially improve performance when dependences are acyclic, but semantically-commutative updates often consist of read-modify-write chains that cause cyclic dependencies.

SI-TM [164] relaxes serializability and implements snapshot isolation, which only flags write-write dependences as conflicts. SI-TM, like other schemes that weaken serializability [4, 235], can allow more concurrency on reads and writes to the same data but requires programs to be rewritten to work under a less intuitive concurrency model. SI-TM also relies on an expensive multiversioned main memory. Finally, SI-TM also cannot handle conflicting read-modify-write operations, which cause write-write conflicts (e.g., unlike COMMTM, SI-TM bottlenecks on kmeans [164]).

Other techniques focus on reducing the cost of mispeculation. ReSlice [226] reexecutes only the conflicting load and its dependent instructions, and RetCon [33] performs symbolic reexecution of simple, conflicting auxiliary updates (e.g., updates to shared counters that are not used elsewhere in the transaction). Unlike these schemes, COMMTM does not trigger conflicts to begin with, avoiding superfluous communication and serialization. COMMTM is also much cheaper than ReSlice and allows a broader range of operations than RetCon.

Finally, open-nested transactions [180, 181] can provide some of the benefits of commutativity. Unlike conventional (closed) nested transactions, which remain speculative until their parent commits, open-nested transactions commit when they end, and specify an abort handler to undo their effects if their parent later aborts. While open-nested transactions make their parents less vulnerable, the nested transactions still suffer from conflicts and serialization. By contrast, COMMTM can support truly concurrent and communication-free updates to the same data. Moreover, open nesting is only practical when operations are easy to undo, which commutative operations may lack (e.g., top-K in Section 4.9).

4.12 Summary

We have presented COMMTM, an HTM that exploits semantic commutativity to allow multiple transactions updating shared data concurrently and without conflicts. COMMTM extends the coherence protocol and the conflict detection scheme and preserves transactional guarantees. Moreover, COMMTM's basic scheme allows as much concurrency as semantic locking. Gather requests allow COMMTM to reduce conflicts even further.

COMMTM bridges the precision-overhead dichotomy of hardware vs software conflict detection: As a result, COMMTM scales many operations that serialize in conventional HTMs, such as set insertions, reference counting, and top-K insertions, while retaining the low overhead of HTMs. At 128 cores, COMMTM outperforms an eager-lazy HTM by up to 3.4× and reduces or even eliminates aborts.

HTA 5

In this chapter, to address the inefficiency of small fragments, we propose HTA. Whereas COUP and COMMTM provide general architectural support for commutative updates, HTA is tailored to a specific data structure: hash tables.

Hash tables are widely used, but they are inefficient in current systems: they use core resources poorly and suffer from limited spatial locality in caches. HTA is a technique that accelerates hash table operations through a combination of expressive ISA extensions and simple hardware changes (Section 5.2). HTA adopts a hash table format that leverages the associative nature of caches. HTA introduces new instructions to perform hash table lookups and updates. These instructions are designed to leverage existing core structures and prediction mechanisms. For example, hash table lookups have branch semantics and thus leverage the core's branch predictors to avoid control-flow stalls. With a simple HTA function unit, these instructions consume far fewer pipeline resources than conventional hash table operations, allowing more instruction-level and memory-level parallelism to be exploited. HTA accelerates most hash table operations, leaving rare cases to a *software path* that allows overflowing to conventional software hash tables.

We present two implementations of HTA, FLAT-HTA (Section 5.3) and HIERARCHICAL-HTA (Section 5.4). Both implementations introduce simple changes to cores to reduce runtime overheads. FLAT-HTA adopts a simple, hierarchy-oblivious layout that works well for hash tables with uniform reuse. HIERARCHICAL-HTA adopts a multi-level, hierarchy-aware layout that lets fast caches hold more frequently accessed key-value pairs, improving spatial locality when hash tables have mixed reuse. HIERARCHICAL-HTA requires changing cache controllers and provides modest benefits over FLAT-HTA. These implementations do not reserve space in caches. Instead, they dynamically share cache capacity with non-HTA data.

We evaluate HTA on hash table-intensive benchmarks and use it to accelerate *memoization*, a technique that caches the results of repetitive computations, allowing the program to skip them (Section 5.5). FLAT-HTA accelerates hash table-intensive applications by up to 2×, while HIERARCHICAL-HTA outperforms FLAT-HTA by up to 35%. Moreover, HTA outperforms software memoization by 2× and achieves comparable performance to conventional hardware memoization but without the need for specialized on-chip storage.

Motivation 5.1

Hash Tables 5.1.1

Hash tables are unordered associative containers. They hold *key-value pairs* and support three operations: *lookups* to retrieve the data associated with a particular key, and *insertions* and *deletions* to add or remove key-value pairs. Hash tables perform these operations with amortized constant-time complexity. They are

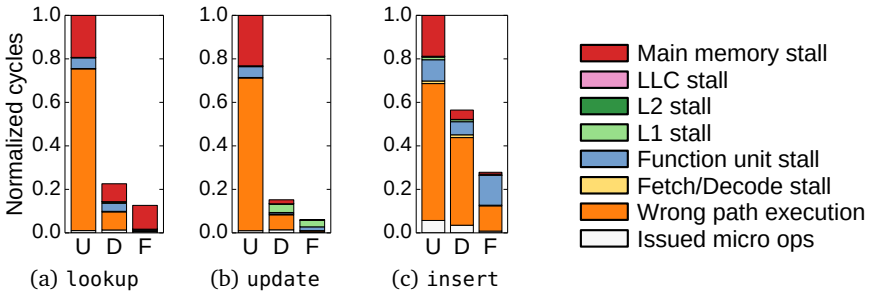


Figure 5.1: Execution time and cycle breakdown of three hash table microbenchmarks using three hash table implementations: libstdc++'s C++11 Unordered map, Google's Dense hash map, and Flat-HTA.

heavily used in many domains, like databases [144], key-value stores [158, 161, 162], networking [240], genomics [175], and memoization [177, 245].

A hash table is typically implemented using an array to hold key-value pairs, which is indexed by a *hash* of the key. A *collision* happens when multiple key-value pairs map to the same array index. Collisions become more common as array utilization grows. To support high utilization, hash tables include a collision resolution strategy, such as probing additional locations, and resize the table when its utilization reaches a certain threshold.

Implementations vary in several aspects, like hash function selection, collision resolution, and resizing mechanisms. Simple hash functions such as XOR-folding and bit selection [120] are fast but are prone to hotspots, while more complex hash functions such as universal hashing [40] distribute key-value pairs more uniformly but incur more overheads. Basic collision resolution strategies include chaining and open addressing [172]. Upon a collision, chaining appends the new key-value pair to the existing ones, forming a linked list, while open addressing probes other positions in the hash table. Resizing can be performed all-at-once or incrementally.

There is a wide range of hash table implementations with different algorithmic tradeoffs, e.g., trading space efficiency for lookup efficiency [195]. For instance, Cuckoo hashing [195] improves space efficiency and worst-case lookup performance at the cost of increasing average-case lookup complexity. Its variants further focus on either reducing the memory accesses per lookup [36] or improving locality [88].

5.1.2 Hash table performance analysis

Despite the wide range of hash table implementations, we observe that state-of-the-art designs suffer from two issues: *poor core utilization* and *poor spatial locality*:

1. Poor core utilization adds overheads that limit the performance of many hash table-intensive applications [144]. To analyze the causes of high overheads, we evaluate three common hash table operations under different hash table implementations using detailed simulation (see Section 5.6 for methodology details). We use two state-of-the-art software baselines, `libstdc++`'s C++11 `unordered_map` and Google's `dense_hash_map`, as well as FLAT-HTA.

Figure 5.1 compares the execution time (lower is better) of each implementation under three cases: (a) lookups, (b) updates, and (c) insertions. In (a) and (b), each hash table is initialized with 1 million randomly generated key-value pairs, and has a footprint of about 64 MB. Then, the program performs back-to-back lookups or updates to existing, randomly chosen keys. In (c), each hash table starts empty and 1 million distinct, randomly chosen key-value pairs are inserted into it. Over time, the hash table grows to accommodate the inserted pairs.

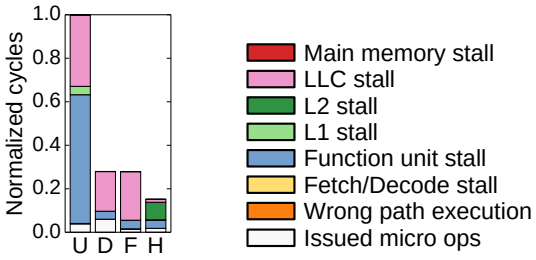


Figure 5.2: Execution time and cycle breakdown of the mixed-reuse microbenchmark for the previous hash tables and Hierarchical-HTA.

Figure 5.1 breaks down execution time into the cycles cores spend on different activities, following the CPI stack methodology [87]. Specifically, each bar shows the cycles cores spend *(i)* issuing committed instructions; *(ii)* executing wrong-path instructions due to a branch misprediction; *(iii)* stalled (i.e., unable to issue) due to the frontend (fetch or decode); and *(iv)* stalled due to different backend events: functional units, L1 cache, L2 cache, LLC, or main memory.

Figure 5.1 reveals two key sources of overhead in software hash tables: hard-to-predict branches and underutilized backend parallelism.

First, Figure 5.1 shows that hard-to-predict branches in hash table probing add many cycles: up to 74% of cycles are spent on wrong-path execution. This is because, in both `unordered_map` and `dense_hash_map`, such branches direct the control flow to either the end of an operation or another hash table probing. These branches depend on data loaded from memory, so they take a long time to resolve and are challenging for branch predictors.

Second, hash table operations make poor use of backend resources to exploit instruction-level and memory-level parallelism. Each hash table operation takes a sequence of instructions including hash calculation, memory accesses, comparisons, and branches. These instructions occupy tens to hundreds of micro-op (μop) slots, comparable to the reorder buffer size. As shown in Figure 5.1, this limits memory-level parallelism significantly: most backend stalls are spent waiting for main memory responses, and the reorder buffer does not have enough resources to overlap multiple misses.

FLAT-HTA effectively reduces these overheads and improves performance by up to 2.5×. First, its design avoids hard-to-predict branches, reducing or even eliminating wrong-path execution. Second, each hash table operation takes far fewer μop slots, improving memory-level parallelism and reducing backend stalls by up to 5.6×.

2. Poor spatial locality is the other issue in software hash tables [36]. Hash tables spread key-value pairs uniformly across the table's allocated memory. This hinders spatial locality, as the neighboring pairs of a frequently-accessed pair are usually not frequently accessed. This wastes a significant portion of cache capacity.

To illustrate this, we design a microbenchmark similar to the previous ones. First, we size the hash tables to occupy 256 MB and pre-insert 1 million key-value pairs. By sizing the hash tables to 256 MB instead of their natural 64 MB, we keep hash table load artificially low to reduce branch mispredictions in the software versions (at this low load, the first probe almost always succeeds). This is, however, space-inefficient. Then, the benchmark performs a series of dependent lookups to a subset of 8,000 keys.

Figure 5.2 shows the execution time and cycle breakdown for the previous hash table implementations plus HIERARCHICAL-HTA. Since there are no branch mispredictions and lookups depend on each other, the benchmark has limited parallelism. As a result, FLAT-HTA outperforms the best software implementation by only 1%, as it spends 80% of cycles waiting for LLC responses.

By contrast, by adopting a multi-level hierarchy-aware hash table layout, HIERARCHICAL-HTA densely packs frequently accessed key-value pairs in lower-level caches, and therefore reduces misses. As shown in Figure 5.2, HIERARCHICAL-HTA serves most of the lookups from the L2 instead of the LLC, outperforming FLAT-HTA by 84%.

5.1.3 Prior work in accelerating hash tables

Prior work has introduced hardware support to reduce hash table overheads. In databases, prior work [106, 144] leverages the inter-key parallelism of database operators such as join to exploit data-level parallelism. These techniques optimize the throughput but not the latency of hash table accesses.

Near-memory [166] and near-storage acceleration [249, 273] bypass the cache hierarchy entirely, and are a sensible choice when operating on large hash tables with no locality. They do avoid the spatial locality problems of caching hash tables. However, they incur high latency and work poorly when hash table accesses have locality.

Other hardware techniques introduce specialized hardware units for hashing and comparisons instead of using the processor pipeline, and allocate dedicated on-chip storage for hash tables. They are typically specialized for applications such as PHP processing [96] and distributed key-value stores [46, 162]. Like HTA, these techniques do reduce the latency of hash table operations. Unlike HTA, these techniques introduce large storage structures that rival or exceed the area of the L1 cache. For example, Da Costa et al.'s proposal to accelerate memoization [71] consumes 98 KB. However, not all applications can benefit from this storage. In these cases, this dedicated storage not only wastes area that could otherwise be devoted to caches, but also hurts energy consumption [58].

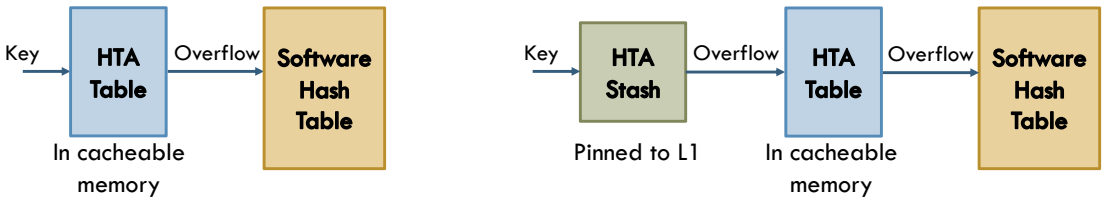
By contrast, HTA is general and optimizes both the throughput and the latency of hash table operations. HTA avoids specialized on-chip storage by storing hash tables in caches, so they share scarce on-chip memory capacity with other program data.

5.1.4 Memoization

Memoization is a technique to improve performance and energy efficiency. Memoization caches the results of repetitive computations, allowing the program to skip them. Memoized computations must be pure and depend on few, repetitive inputs. Memoization is the cornerstone of many important algorithms, such as dynamic programming, and is widely used in many languages [173, 174], especially functional ones. It was first introduced by Michie in 1968 [177]. Since then, it has been implemented using software and hardware, but both have significant drawbacks, which we address with HTA.

Software memoization relies on hash tables to memoize input-output pairs. The high runtime overheads of hash tables hamper software memoization significantly. As we later show, many memoizable functions are merely 20 to 150 instructions long, comparable to or even cheaper than hash table lookups. Memoizing them is harmful. For example, Citron and Feitelson [60] show that software memoization incurs significant overheads on short functions: when memoizing mathematical functions indiscriminately, software memoization incurs a 7% performance loss, while a hardware approach yields a 10% improvement. To avoid poor performance, software schemes must apply memoization selectively, relying on a careful cost-benefit analysis of memoizable regions, done by either compilers [82, 245], profiling tools [78], or programmers [246].

Prior work has proposed hardware support to accelerate memoization [60, 245, 253], and thus can unlock more memoization potential. Much prior work on hardware memoization focuses on automating the detection of memoizable regions at various granularities [71, 117, 237, 253], while others rely on ISA and program changes to select memoizable regions [60, 64, 245]. However, all prior hardware techniques require dedicated storage for memoization tables. Such tables require similar or even larger sizes than



(a) FLAT-HTA (b) HIERARCHICAL-HTA

Figure 5.3: Overview of HTA implementations.

L1 caches. Therefore, they incur significant area and energy overheads [58], especially for programs that cannot exploit memoization.

Other prior work has proposed architectural [254] or runtime [220] support to track implicit inputs/outputs of memoized functions, enabling memoizing impure functions. This is orthogonal to the acceleration of hash tables, which is the focus of our work. HTA could be easily combined with them.

HTA Hardware/Software Interface 5.2

HTA is a Hash Table Acceleration technique. HTA handles most hash table accesses in hardware, and leaves rare cases such as overflows and table resizing to a slow software path. HTA introduces two key software-visible features to accelerate hash table operations in hardware.

First, HTA adopts a format for hash tables that exploits the characteristics of caches to make lookups and updates fast. HTA stores hash tables in cacheable memory. This avoids the large costs of specialized hardware caches used by prior hardware techniques. HTA does not statically partition cache capacity between hash table data and normal program data. Instead, both types of data are managed by the unified cache replacement policy, and share cache capacity dynamically based on access patterns.

Second, HTA introduces hash table instructions for lookups and updates that are amenable to a fast and simple implementation. Whereas software hash table lookups use multiple instructions and hard-to-predict branches, HTA hash table lookups are done through a single instruction with branch semantics. The outcome of a lookup (resolved or not) can be predicted accurately by the core's existing branch predictors, avoiding most control-flow stalls.

Figure 5.3 gives an overview of how both HTA implementations, FLAT-HTA and HIERARCHICAL-HTA, use these features. FLAT-HTA (Figure 5.3a) stores key-value pairs across an HTA *table* and a software hash table. The HTA table is stored in cacheable memory, and may be spread across multiple caches or main memory. The HTA table is sized to hold most key-value pairs, and the software hash table is used as a victim cache, to hold pairs that overflow the HTA table.

HIERARCHICAL-HTA (Figure 5.3b) extends FLAT-HTA by letting cache levels retain individual key-value pairs rather than cache lines. Specifically, they cache key-value pairs of the HTA table in small, cache-level-specific regions called HTA *stashes*. A pair that overflows an HTA stash is handled by the next level. This improves spatial locality at intermediate caches levels, as their lines fill up with frequently-accessed pairs. However, HIERARCHICAL-HTA does not improve spatial locality at the last-level cache (doing so would complicate the interface with main memory), so its benefits over FLAT-HTA are modest. Figure 5.3b shows an example of HIERARCHICAL-HTA with one HTA stash pinned to the L1.

We now describe the ISA changes common to FLAT-HTA and HIERARCHICAL-HTA, then describe their implementations.

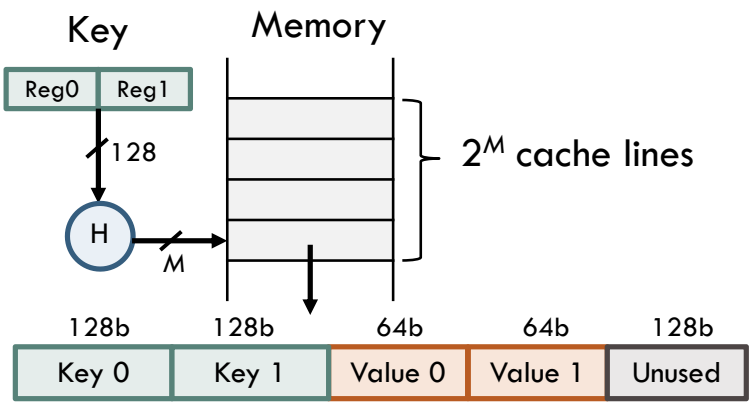


Figure 5.4: HTA table format.

5.2.1 HTA hash table format

The HTA table is stored in a contiguous, fixed-size region of memory, as shown in Figure 5.4.

HTA uses a storage format designed to leverage the characteristics of caches. Each cache line stores a fixed number of key-value pairs. For example, Figure 5.4 shows the format of a 64-byte cache line for a hash table with 128-bit keys and 64-bit values. A given entry can map to a single cache line, but can be stored in any position within the line. A lookup thus requires hashing the input data to produce a cache line index, fetching the line at that index (as shown in Figure 5.4), and comparing all keys in the line. This design requires accessing a single cache line, but retains associativity within a line to reduce collisions. To avoid the need for valid bits, HTA initializes each line's entries with *invalid* key values, which are simply keys that hash to a different line.

HTA leaves collision resolution to software. Specifically, there may be *overflowed* key-value pairs that cannot be stored in a line due to capacity constraints. These overflowed pairs are handled by the software path, which stores them in the software hash table.

5.2.2 HTA ISA extensions

HTA stores a small number of HTA *table descriptors* in architectural registers. Each descriptor holds the table's starting address and its size. Our implementations support four HTA table descriptors. If the program uses more than four hash tables, it should manage their descriptors accordingly, loading them into registers before operating on the hash table.

HTA adds four instructions to perform hash table operations: `hta_lookup`, `hta_update`, `hta_swap`, and `hta_delete`. These instructions have branch semantics. Figure 5.5 and Figure 5.6 show sample code that uses them to implement single-threaded lookups and insertions. (Section 5.3.4 describes how these instructions are used to implement thread-safe hash tables for multithreaded applications.)

1. `hta_lookup` performs a lookup in the HTA hash table whose descriptor is specified by table_id. `hta_lookup` supports keys with up to four integer or floating-point words and a single integer or floating-point value, all stored in registers. As shown in Figure 5.5, `hta_lookup` stores the number of integer and floating-point key registers, and the core decodes them to a fixed set of registers. We choose the same register mappings as the ISA's calling convention. For instance, in x86-64, num_int_keys = 2 means that the 64-bit values in registers `rdi` and `rsi` are used as a 128-bit key. Similarly, the is_int_value indicates whether the value is integer or floating-point. In x86-64, either `rax` or `xmm0` is used accordingly.

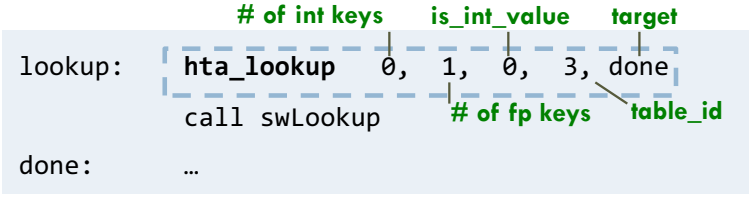


Figure 5.5: Example showing how `hta_lookup` is used to implement a singled-threaded hash table lookup.

```
insert:   hta_swap  0,  1,  0,  3, done
          call swHandleInsert
done:     …
```

Figure 5.6: Example showing how `hta_swap` is used to implement a single-threaded hash table insertion.

If the lookup is resolved, i.e., the key is found or the line is not full, `hta_lookup` acts as a taken branch. It jumps to the **target** PC encoded in the instruction (in PC-relative format), sets the overflow flag to indicate whether the lookup succeeds, and also updates the result register with the corresponding value. If the lookup is not resolved, i.e., the key is not found *and* the line is full, `hta_lookup` acts as a non-taken branch, and continues to execute the next instruction.

2. `hta_update` is used to update the HTA hash table. Like `hta_lookup`, `hta_update` encodes the key and value registers, and the table id. If the key is found or the line is not full, `hta_update` updates the pair in the cache line and jumps to the **target** PC. Otherwise, if the key is not found and the line is already full, `hta_update` does not modify anything and continues to execute the next instruction.

3. `hta_swap` attempts to insert a pair more aggressively than `hta_update`. Similar to `hta_update`, `hta_swap` encodes the key and value registers, and the table id. Upon a `hta_swap`, if the key is found or the line is not full, `hta_swap` performs the same operations as `hta_update`: it updates the pair in the cache line and jumps to the **target** PC. However, if the key is not found and the line is full, `hta_swap` selects a victim pair randomly to make space for the update. The victim's key and value are placed in registers and `hta_swap` acts as a non-taken branch, letting the software path finish the update, e.g., by inserting the victim pair to the software hash table. Such distinction between `hta_update` and `hta_swap` is useful for thread-safe hash tables (Section 5.3.4).

4. `hta_delete` removes a key-value pair with a matching key. Its format is identical to `hta_lookup`. If the key is found, it is replaced with a special *deleted* key, and the instruction acts as a taken branch. Otherwise, `hta_delete` acts as a non-taken branch to take the software path.

Deleted key values must be different from invalid key values, as `hta_lookup` should not interpret a deleted key as empty space (so that lookups do not miss pairs that overflowed to the software table), but `hta_update` and `hta_swap` should interpret a deleted key as empty space. In all, HTA uses four pre-specified key values: it chooses two small key values that do not map to line 0 as line 0's invalid and deleted key values, and two small key values that map to line 0 as the invalid and deleted key values for all other lines.

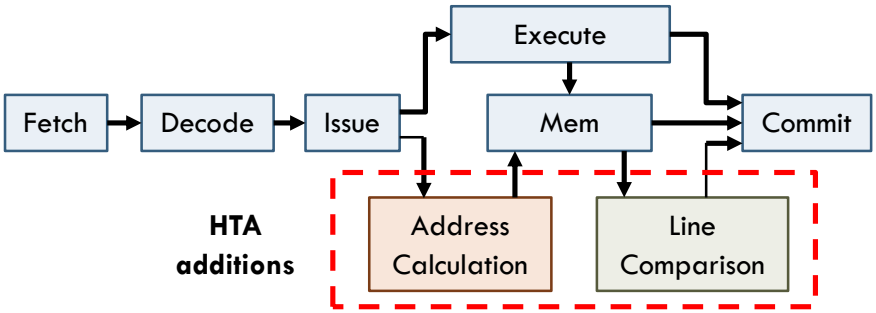


Figure 5.7: HTA core pipeline changes.

Single-threaded lookups: Figure 5.5 shows the implementation of a hash table lookup with HTA instructions. The lookup begins with the `hta_lookup` instruction. A resolved `hta_lookup` jumps to `done` and continues program execution. Otherwise, the program goes through the software path to perform a lookup in the software hash table. In this way, the HTA hash table behaves as an *exclusive cache* of the conventional software hash table, allowing most of the accesses to be handled quickly. The software path is rarely executed, and therefore introduces little performance impact.

Single-threaded insertions: Similarly, Figure 5.6 illustrates how to implement an insertion with HTA (if a pair with the same key already exists, an insertion updates its value). Either `hta_swap` or `hta_update` can be used. A resolved `hta_swap` instruction jumps to `done`, skipping the software path. An unresolved `hta_swap` runs through the software path, which *(i)* inserts the victim pair to the software hash table, and *(ii)* checks whether the software hash table has a pair with the same key as the newly inserted pair, and removes it if so, as this old, overflowed pair is now stale.

5.2.3 ISA design alternatives

We have designed the HTA ISA to integrate well in x86 processors: HTA instructions are encoded in a compact format and are decoded into multiple µops upon execution (Section 5.3.1). An alternative RISC-style implementation is also possible, e.g., by exposing the different µops as instructions. However, this design choice is not important: as shown in Figure 5.1 and Figure 5.2, the time spent on frontend stalls and issuing µops is negligible, so using CISC- vs. RISC-style instructions would not significantly change the results. The key benefit of HTA is to reduce wrong-path execution and backend stalls.

5.3 FLAT-HTA Implementation

As shown in Figure 5.3a, FLAT-HTA uses a single-level HTA table stored in cacheable memory. FLAT-HTA substantially reduces overheads over software hash tables, but still suffers from poor spatial locality.

5.3.1 Core pipeline changes

FLAT-HTA requires simple changes to cores, shown in Figure 5.7. We add a simple functional unit that executes lookup, update, swap, and delete instructions. This unit is fed the values of key registers, possibly over multiple cycles, as well as the table id.

For an `hta_lookup` instruction, the unit first hashes the input values and table size to find the line index. We restrict the system to use power-of-2 sizes for each HTA table. We use the x86 CRC32

Circuit	Address calculation	Line comparison	Total
Area(μm^2)	6,173	9,176	15,349
Area(%core)	0.022	0.033	0.055

Table 5.1: Area of the HTA functional unit on a 45 nm process.

instruction to compute the hash value; other ISAs could implement CRC or a different cheap hash [120]. We find that CRC produces good distributions in practice.

After hashing, the HTA functional unit loads the appropriate cache line, compares all the keys, and outputs whether the the software path can be skipped, whether there's a match, as well as the corresponding result if so. `hta_update` and `hta_swap` are similar, but the functional unit also takes the value to update, and stores the pair in the appropriate line. `hta_delete` is also similar, but does not return a value.

HTA leverages existing core mechanisms to improve performance. We integrate these instructions into an out-of-order, superscalar x86-64 core similar to Intel's Haswell (see Section 5.6). The frontend treats HTA instructions as branches. This way, HTA instructions leverage the existing branch target buffer and branch predictor to predict whether the code following each instruction can be skipped. Thus, the core overlaps the execution of lookups and updates with either the execution of the software path (if a resolution is not predicted) or its continuation (if a resolution is predicted). We find that this effectively hides the latency of HTA instructions.

In our implementation, the backend executes `hta_lookup` using multiple RISC μops. The decoder produces one or more μops that feed each input register to the HTA functional unit, an HTA μop that instructs the functional unit to start, a branch-resolution μop, and, if the lookup is predicted to be resolved, a μop to move the lookup result into its destination register. The other instructions use a similar implementation.

Hardware costs 5.3.2

We implement the HTA functional unit in RTL and synthesize it using yosys [269] and the 45 nm FreePDK45 standard cell library [143]. The functional unit meets a target 3 GHz frequency. The address calculation circuit mainly consists of a 64-bit adder, 64 AND gates, and registers to store the four HTA table descriptors. The line comparison circuit includes comparators to search for a given key and an empty slot in parallel. Table 5.1 reports the area consumed by these components. Overall, the functional unit takes just 0.055% of the area of a Nehalem core [86], which was manufactured in a 45 nm process as well. Thus, HTA's area overheads are negligible.

Software path 5.3.3

The software path performs lookups and updates to a conventional software hash table. It handles the rare overflowed accesses to HTA. Besides, the software path also resizes the HTA table dynamically. The resizing algorithm HTA adopts is based on comparing the fraction of HTA accesses that take the software path with a threshold (e.g., 1%). If the fraction is above the threshold, the software path doubles the size of the HTA table and reinserts all existing elements in both the HTA table and the software hash table.

To keep track of this fraction, each HTA table is assigned a counter that is stored in memory at one word above its starting address. The counter is incremented rarely (every 100 HTA accesses in our implementation), and hence approximately monitors the number of HTA accesses of the table.

The software path also maintains a counter recording the number of software path invocations. The software path uses these counters to calculate the fraction of accesses that overflow, and decides whether to resize the HTA table.

```
lookup:   hta_lookup   0,  1,  0,  3, done
          call swLockLine
          hta_lookup   0,  1,  0,  3, release
          call swLookup
release:  call swUnlockLine
done:     …
```

Figure 5.8: Example showing how `hta_lookup` is used to implement a thread-safe hash table lookup. The software path uses fine-grain locks.

```
insert:   hta_update   0,  1,  0,  3, done
          call swLockLine
          hta_swap     0,  1,  0,  3, release
          call swHandleInsert
release:  call swUnlockLine
done:     …
```

Figure 5.9: Example showing how HTA instructions are used to implement a thread-safe insert. The software path uses fine-grain locks.

5.3.4 Parallel hash table implementation

With multiple threads, the simple hash table operations shown in Figure 5.5 and Figure 5.6 need some refinement to be thread-safe. We leverage that HTA instructions are atomic (cores already have the machinery to ensure this for all instructions, such as line locking or verification loads). This guarantees the atomicity of operations that do not invoke the software path.

If the software path is invoked, a synchronization strategy is needed to guarantee atomicity. We use fine-grain locks, each of which protects a few lines (four in our implementation). However, HTA is orthogonal to the synchronization technique used by the software path, and can use other techniques. For example, it could be combined with transactional memory.

Thread-safe lookups: Figure 5.8 shows our thread-safe implementation of lookups. The software path involves acquiring the line's lock; executing the `hta_lookup` instruction again; if needed, accessing the software hash table; and finally releasing the lock. `hta_lookup` must be invoked again after locking to avoid races with insertions.

Thread-safe insertions: Figure 5.9 shows code for thread-safe updates. This code shows why `hta_update` and `hta_swap` are both needed: `hta_update` does not modify HTA table state if the software path is invoked. This is important to avoid races: by using `hta_swap` only after locking, all modifications are properly synchronized.

5.4 HIERARCHICAL-HTA Implementation

HIERARCHICAL-HTA extends FLAT-HTA to cache individual key-value pairs of the HTA table in cache-specific regions called HTA *stashes* (Figure 5.3b). A stash's lines can only be stored in a specific cache

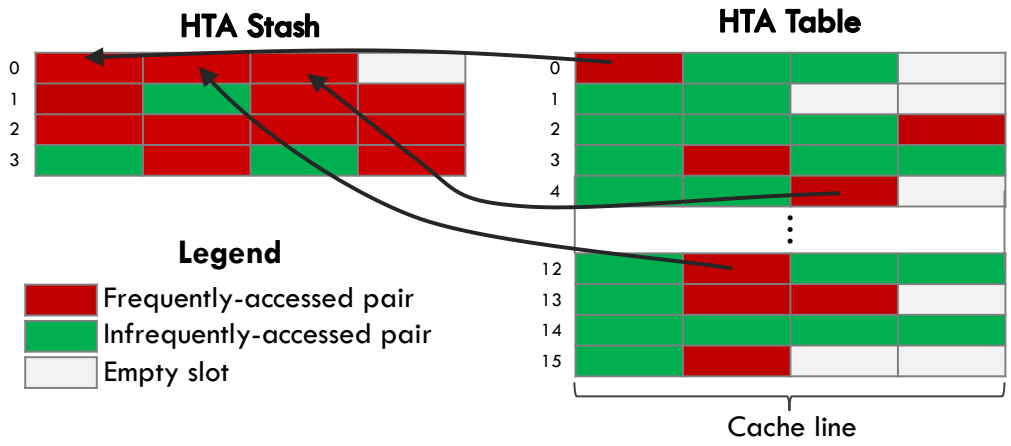


Figure 5.10: HTA stash format.

level. Stashes do not reserve any capacity in their cache (i.e., they do not partition the cache). Instead, similar to FLAT-HTA, each stash shares capacity with normal program data, and the actual capacity a stash consumes depends on the workload's access pattern.

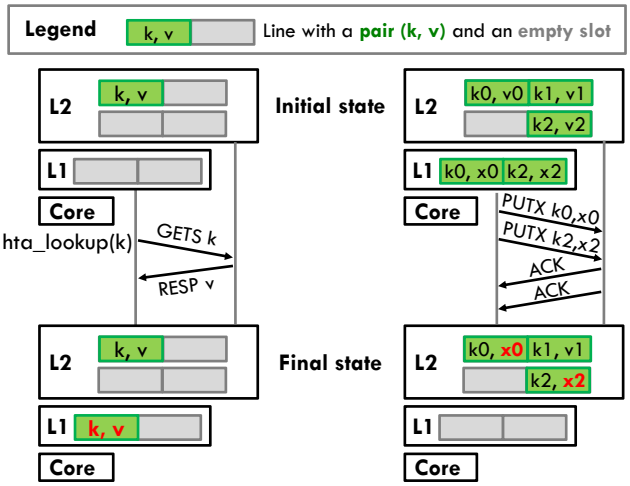
This hierarchy-aware layout improves spatial locality on intermediate cache levels, improving cache utilization and reducing misses. Whereas FLAT-HTA only requires changes to the core, HIERARCHICAL-HTA also modifies cache controllers so that they can fetch and serve key-value pairs rather than cache lines. However, HIERARCHICAL-HTA does not improve spatial locality at the LLC, as making the LLC manage key-value pairs rather than lines would complicate the interface main memory (which is optimized for wide transfers). Therefore, HIERARCHICAL-HTA yields only modest gains over FLAT-HTA.

HTA table restrictions: For simplicity, we introduce some restrictions on the backing HTA table: it must be in a contiguous region of *physical* memory, must be power-of-two sized, and must be size-aligned. (FLAT-HTA tables live in pageable virtual memory so they do not have these restrictions.) These restrictions let us operate on physical addresses, avoiding the need for TLBs on caches, and simplify addressing.

HTA stash format: Figure 5.10 shows an example layout of an HTA stash and its corresponding HTA table. For simplicity, each HTA stash uses a contiguous range of 2^K cache lines. 2^K can be greater than the number of lines in the cache that the stash lives in. Suppose the HTA table is 2^M lines large. Then, given the HTA table address of a particular key, its HTA stash address is computed by zeroing the highest $M-K$ bits of its offset within the HTA table. Key-value pairs will map to line X in the HTA stash if they map to lines $X, X+2^K, X+2\cdot 2^K, ..., X+(2^M-2^{M-K})$ in the HTA table.

Cache controllers store some information about each of their HTA stashes: the starting address and size of their corresponding HTA table, and the key-value pair format. This limits the number of HIERARCHICAL-HTA hash tables that each cache may hold (to four hash tables in our implementation).

Per-pair management: Cache controllers are extended to manipulate and communicate individual key-value pairs within each level: they perform shared fetches (GETS), exclusive fetches (GETX), and dirty writebacks (PUTX) on key-value pairs, analogous to the usual requests for line fetches and evictions in conventional caches. Each HTA operation checks the hash table's HTA stashes in sequence, and the next-level HTA stash (and eventually the HTA table) is accessed only when the current stash cannot resolve the operation.



(a) `hta_lookup` (b) Eviction

Figure 5.11: Pair-grain memory ops in HIERARCHICAL-HTA.

Figure 5.11 illustrates HIERARCHICAL-HTA's operation on a system with a two-level cache hierarchy and an L1-pinned HTA stash. Suppose the L1 starts empty. An `hta_lookup` triggers a `GETS` request with the key and HTA table line from the L1, as shown in Figure 5.11a. The L2 accesses the right HTA table line (fetching it from memory if needed), and responds with its associated value. The L1 allocates space for the HTA stash line and installs the key-value pair there.

The HTA table is inclusive of HTA stashes. Updates are similar to lookups but issue GETX (exclusive) requests. On an update, if the HTA table does not have a pair with the same key, the pair is first inserted into the HTA table.

Caches can evict HTA stash lines as shown in Figure 5.11b. Individual key-value pairs are written back if the line is marked as modified, and are simply dropped if the line is clean.

Overflows: Overflows in an HTA stash are transparent to software: the cache evicts a randomly-chosen pair to the next level to make space for a new one. Overflows in the HTA table are treated the same way as in FLAT-HTA, by invoking the software fallback path for updates. Note that, since the HTA table is inclusive of HTA stashes, overflows or evictions in HTA stashes never cause HTA table overflows.

Coherence: Finally, we maintain coherence conservatively. Coherence is tracked at the shared last-level cache, for each line in the HTA table. When an LLC line in the HTA table is evicted, or when the line is shared and an exclusive request is received, all the sharers of the line are sent invalidations. At smaller caches that contain HTA stashes, an exclusive request (due to an update) that falls on an HTA stash line with shared permission (due to lookups) causes the pairs in the line to be dropped. These policies let us reuse line-level coherence metadata, though they are less precise than if we performed pair-by-pair coherence.

5.5 HTA-Accelerated Memoization

Memoization improves performance and saves energy by caching and reusing the outputs of repetitive computations. As discussed in Section 5.1.4, prior software and hardware memoization techniques have significant drawbacks. Software memoization suffers from high runtime overheads, and is thus limited to

```
memo_exp: hta_lookup  0,  1,  0,  3, done
          call exp
          hta_swap    0,  1,  0,  3, done
done:     …
```

Figure 5.12: Example showing how HTA instructions are used to memoize the `exp()` function.

long computations. Prior hardware techniques achieve low overheads and can memoize short functions, but they rely on large, special-purpose memoization caches that waste significant area and energy.

We leverage HTA to accelerate memoization cheaply, matching the performance of prior hardware memoization techniques without dedicated on-chip storage.

Memoization tables are allocated for memoizable functions. Each memoization table is a hash table that stores arguments as the key and return values as the value. Since memoization tables are small, we use FLAT-HTA to implement them. These FLAT-HTA tables *do not have conventional software paths* that manipulate software hash tables. Instead, on an HTA table miss the software path simply calls the memoizable function. This is a good tradeoff: executing the short function is cheaper than a software hash table lookup.

Memoization operations are implemented using HTA instructions. Figure 5.12 shows example code that leverages HTA instructions to memoize a single-argument, single-result function (`exp`). We place an `hta_lookup` before the call to the memoized function. If the key is found, then the corresponding value is returned in the return value register and the function call is skipped. Otherwise, the function is executed and its result is memoized using `hta_swap`.

Since memoization tables do not grow to accommodate extra items, insertions simply replace one of the line's entries. As a result, there is no software path for `hta_swap` (i.e., **target** equals next PC), because the victim pair is simply dropped. This does not affect the correctness of the program. This is the right tradeoff when memoizing short functions; longer functions could use a full-blown HTA-accelerated hash table.

Exploiting memoizable regions: We have developed a pintool [169] to identify memoizable (i.e., pure) functions [286]. A function is defined as memoizable if it satisfies two conditions. First, its memory reads are either to read-only data or to its stack. Second, its memory writes are restricted to its own stack. Then, we manually added `hta_lookup` and `hta_swap` instructions to these functions' callsites. Due to its low overheads, HTA does not need to perform selective memoization based on cost-benefit analysis as in software techniques. Therefore, we memoize every function that our tool identifies as memoizable. We memoize both application and standard-library functions.

Methodology 5.6

We perform microarchitectural, execution-driven simulation using zsim [224]. We evaluate 1-core and 16-core chips with parameters shown in Table 5.2. These systems use out-of-order cores modeled after Haswell, and a 3-level cache hierarchy with a shared, inclusive LLC that has 2 MB per core.

Our HTA implementation includes registers for four HTA table descriptors. HTA instructions incur the cost of a cache-line load/store (in two 256-bit accesses), plus one cycle to perform key comparisons. We

Core	x86-64 ISA, 3.0 GHz, Haswell-like OOO: 16B-wide ifetch, 2-level bpred with 2 K×18-bit BHSRs + 4 K×2-bit PHT, 4+1+1+1 decoders, 6 execution ports, 4-wide commit
L1 cache	32 KB, 4-way set-associative, 3-cycle latency, split D/I
L2 cache	256 KB, 8-way set-associative, 7-cycle latency, inclusive
L3 cache	2 MB per core, 16-way set-associative, 15-cycle latency, inclusive
Main mem	DDR3-1600, 4 channels (16 cores) or 1 channel (1 core)

Table 5.2: Configuration of the simulated system.

	Input set	Baseline hash table	# of `hta_lookups`	# of `hta_swaps` & `updates`
bfcounter	ENCSR687ZCM.fastq [251]	C++11	0	26960049
lzw	the Bible	`unordered_map`	4364173	765632
hashjoin	–r-size=16777216 –s-size =268435456 –skew=1.5	Google	268435456	23335399
ycsb-read	-z0.6 -r1.0 -w0.0	`dense_hash_map`	95998531	0
ycsb-write	-z0.6 -r0.0 -w1.0		0	95998531

Table 5.3: Hash table benchmark characteristics.

model an L1 that supports wide accesses (256 bits per cycle), which is common due to SIMD instructions (e.g., 256-bit AVX). We encode `hta_lookup`, `hta_update`, `hta_swap`, and `hta_delete` using x86-64 no-ops that are never emitted by the compiler.

We evaluate HTA using two sets of workloads: one set uses hash tables as a key part of their implementation, and the other set leverages memoization to improve performance. To achieve statistically significant results, we introduce small amounts of non-determinism [12], and perform enough runs to achieve 95% confidence intervals $\leq 1\%$ on all results.

5.6.1 Hash table workloads

We analyze four applications that use hash tables heavily:

- `bfcounter` [175] is a memory-efficient software to count k-mers in DNA sequence data, which is essential for many methods in bioinformatics, including genome and transcriptome assembly. `bfcounter` uses a heavily-updated hash table to hold k-mers. We use a DNA sequence from ENCODE [251] as the input.
- `lzw` is a text compression benchmark based on the LZW algorithm [266], a widely-used lossless data compression technique. A hash table is used to hold the dictionary. We use the Bible as the input text file.
- `hashjoin` [24] is a single-threaded implementation of the hash join algorithm. `hashjoin` joins two synthetic tables. There are two phases in the program: in the first phase, the inner table is scanned to build a hash table; and then in the second phase, the outer table is scanned while the hash table is probed to produce output tuples.
- `ycsb` [65] is an implementation of *Yahoo! Cloud Serving Benchmark* that runs on DBx1000 [279]. Hash tables are used for hash indexes. We evaluate `ycsb` with two configurations: 100% read queries and 100% write queries.

Table 5.3 details these applications and their characteristics.

	Lang.	Benchmark suite	Input Set	Memoizable functions	Memoization table per function
bwaves	Fortran	SPECCPU2006	ref	slowpow, pow, halfulp, exp1	32 KB
bscholes	C++	PARSEC	native	CDNF, exp, logf	32 KB
equake	C	SPECOMP2001	ref	phi0, phi1, phi2	4 KB
water	C	SPLASH2	1061208	exp	32 KB
bf semphy	C++	BioParallel	220	suffStatGlobalHomPos::get	4 KB
nab	C	SPECOMP2012	ref	exp, slowexp_avx	2 KB

Table 5.4: Memoization benchmark characteristics.

We modify each application to support multiple hash table implementations (using template metaprogramming to do so without runtime overheads). We compare the following implementations:

- **Baseline:** Because no single hash table design works best for all applications, we use the best of `libstdc++`'s C++11 `unordered_map` and Google's `dense_hash_map` as the baseline implementation. The best of both either matches or outperforms the application's existing hash tables.
- **FLAT-HTA and HIERARCHICAL-HTA:** To evaluate HTA, we use a hash table implementation with HTA hash tables accessed through `hta_lookup/update/swap/delete` instructions. The HTA hash table starts empty and is resized as elements are inserted. Specifically, if the fraction of software path invocations over total HTA accesses is above 1%, the size of HTA table is doubled. This involves allocating a new HTA table that is twice as large, then inserting all the pairs in both the previous HTA table and the software hash table into the new HTA table. For each application, HTA uses the same software hash table as the baseline. Since HTA rarely uses the software hash table, its performance is insensitive to the choice of software hash table.
- **HTA-SW:** To further analyze HTA and illustrate where performance differences comes from, we implement a software scheme, HTA-SW, that implements the same algorithm as HTA but without any hardware support. HTA-SW uses the same table format, the same software hash tables, and the same resizing algorithm. HTA-SW does not rely on any hardware support: all the steps in hash table operations, including hashing, key comparison, memory accesses, and branches, are implemented purely in software. HTA-SW stores the keys within a cache line in a contiguous format, so that comparisons can be implemented with SIMD load and comparison instructions [293]. Specifically, we use Intel AVX vector load, compare, and mask instructions to exploit the parallelism in key lookups.

We fast-forward each application to skip initialization (e.g., data loading in `ycsb`) and simulate them to completion.

Memoization workloads 5.6.2

We analyze programs from six benchmark suites and choose one application with high memoization potential from each suite. Table 5.4 details these applications and their characteristics. For each application, we use the same memoization table size for all memoized functions. We report the table size that yields the best performance. Section 5.7.5 provides more insight on the effect of table size.

We fast-forward each application for 50 billion instructions. We instrument each program with *heartbeats* that report application-level progress (e.g., when each timestep or transaction finishes), and run it for as many heartbeats as the baseline system (without memoization) completes in 5 billion instructions. This lets us compare the same amount of work across schemes, since memoization changes the instructions executed.

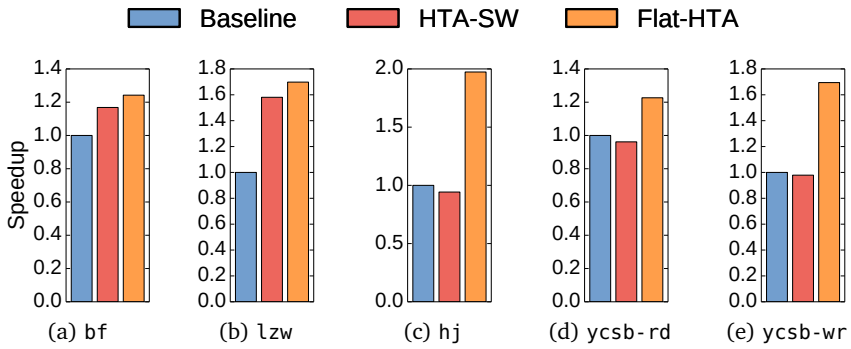


Figure 5.13: Speedups of FLAT-HTA and HTA-SW over the software baseline on single-threaded apps.

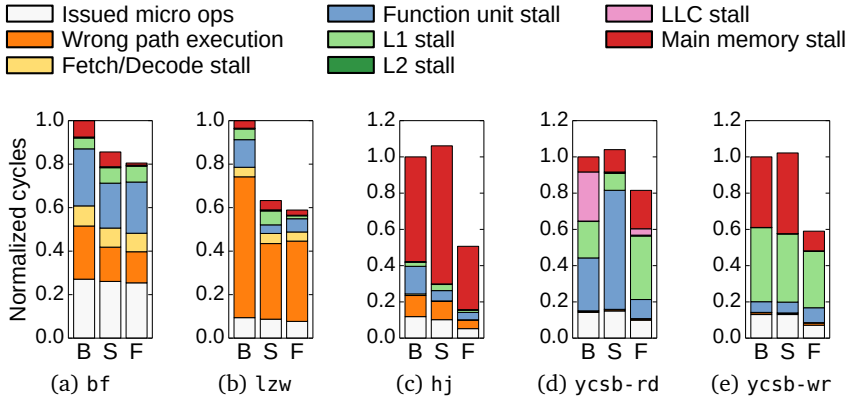


Figure 5.14: Cycle breakdowns for the Baseline, HTA-SW, and Flat-HTA.

5.7 Evaluation

5.7.1 HTA on single-threaded applications

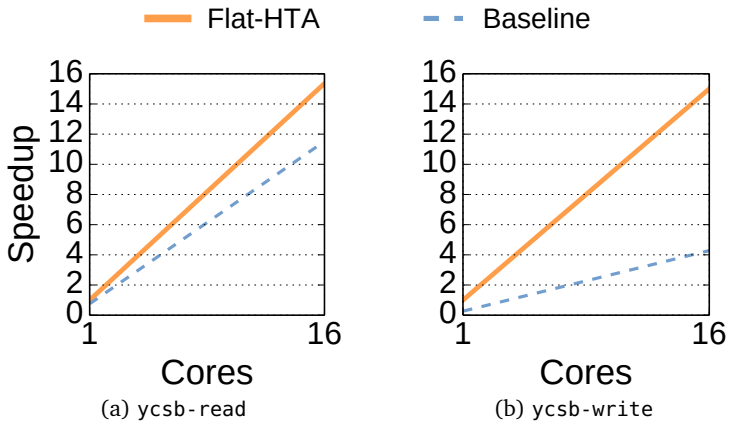
Figure 5.13 compares the performance of the baseline, HTA-SW, and FLAT-HTA. FLAT-HTA outperforms the baseline by 24% on `bfcounter`, 70% on `lzw`, 2.0× on `hashjoin`, 23% on `ycsb-read`, and 69% on `ycsb-write`. Figure 5.14 shows the breakdown of core cycles following the same format as Section 5.1.2, and shows the same trends. `bfcounter` and `lzw` benefit mainly from reduced mispredicted branches, while `hashjoin` and `ycsb` gain mostly from better backend parallelism.

Figure 5.13 also shows that FLAT-HTA outperforms HTA-SW substantially, by 6.3% on `bfcounter`, 7.4% on `lzw`, 2.1× on `hashjoin`, 28% on `ycsb-read`, and 73% on `ycsb-write`. Figure 5.14 shows these benefits stem from reduced wrong-path execution and backend stalls. Specifically, though FLAT-HTA incurs the same cache misses as HTA-SW, applications with abundant operation-level parallelism, like `hashjoin` and `ycsb`, benefit from HTA significantly by using the reorder buffer better: since each hash table operation uses far fewer µops, more operations are overlapped, reducing backend stalls. `ycsb-write` benefits more than `ycsb-read` because FLAT-HTA improves updates more than lookups (as Section 5.1.2 showed).

Moreover, HTA-SW does not consistently improve performance. HTA-SW outperforms the baseline substantially on `bfcounter` and `lzw`, by 17% and 58% respectively, showing that the HTA design can sometimes outperform state-of-the-art hash tables even when implemented entirely in software. However, HTA-SW causes small performance degradations (up to 6%) on `hashjoin`, `ycsb-read` and `ycsb-write`. On these applications, the performance improvement of FLAT-HTA comes from hardware acceleration.

	Baseline		FLAT-HTA and HTA-SW	
	Type	Table	HTA table	Software path
bfcounter	`unordered`	275 MB	512 MB	2 MB
lzw	`_map`	8 MB	16 MB	117 KB
hashjoin	`dense`	512 MB	256 MB	512 B
ycsb-read	`_hash`	512 MB	256 MB	0 B
ycsb-write	`_map`	512 MB	256 MB	0 B

Table 5.5: Memory usage of the baseline and FLAT-HTA.



(a) `ycsb-read` (b) `ycsb-write`

Figure 5.15: Speedups of FLAT-HTA and the baseline on `ycsb` when scaling from 1 to 16 cores. Speedups are relative to the single-threaded baseline.

Beyond performance, space efficiency is an important consideration for hash tables. Table 5.5 reports the memory consumption of the different implementations. HTA-SW uses the same memory layout as FLAT-HTA, and hence has exactly the same memory consumption. Overall, results show that HTA does not cause undue storage overheads—there are differences of 2× in all cases, but note that hash tables grow exponentially over time and small differences in resizing thresholds can cause 2× size differences. On `bfcounter` and `lzw`, which use `unordered_map` as the baseline, the FLAT-HTA table is 2× larger than the baseline's. On `hashjoin` and `ycsb`, which use `dense_hash_map` as the baseline, the FLAT-HTA table is 2× smaller than the baseline's.

HTA on multithreaded applications 5.7.2

We now evaluate FLAT-HTA on multithreaded applications. Since `bfcounter`, `lzw`, and `hashjoin` are single-threaded, we use the multithreaded implementations of `ycsb-read` and `ycsb-write`.

As shown by Figure 5.15, at 16 cores, FLAT-HTA outperforms the baseline by 33% on `ycsb-read` and by 3.5× on `ycsb-write`. These speedups are higher than in the serial version (23% and 69%) because most HTA operations are performed without acquiring locks.

HTA with hierarchy-aware layout 5.7.3

We now compare the performance of FLAT-HTA and HIERARCHICAL-HTA. In this experiment, HIERARCHICAL-HTA uses a 32 KB HTA stash pinned to the L1 cache, followed by a 256 KB HTA stash pinned to the L2 cache. The HTA table is cached in the LLC.

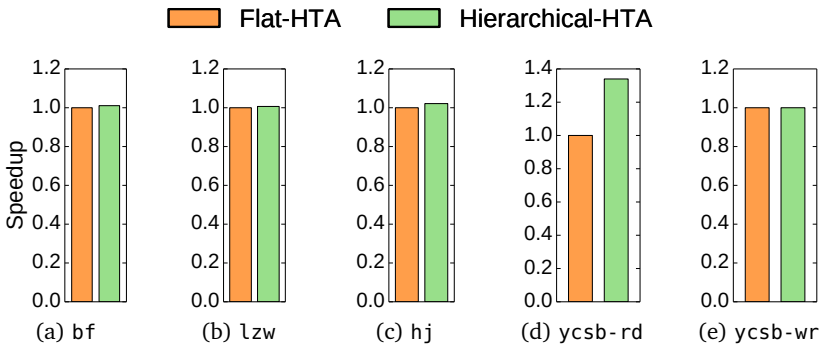


Figure 5.16: Speedups of HIERARCHICAL-HTA over FLAT-HTA.

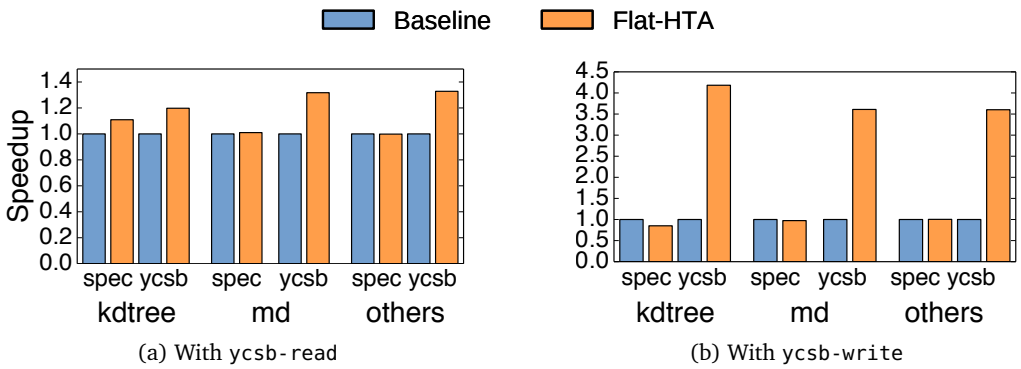


Figure 5.17: Speedups of HTA and baseline when running a SPEC OMP2012 app and `ycsb` simultaneously.

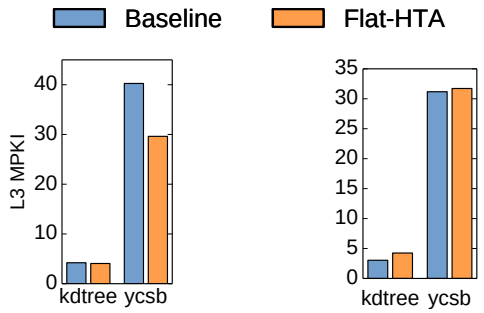
Figure 5.16 compares the performance of both schemes. On `ycsb-read` and `ycsb-write`, where key-value pairs have mixed reuse, HIERARCHICAL-HTA reduces L2 misses by 4.1× and 3.9×, respectively. This happens because HIERARCHICAL-HTA lets the L1 and L2 hold densely-packed pairs. This miss reduction translates to a 35% performance improvement for `ycsb-read`. However, `ycsb-write` attains the same performance because FLAT-HTA completely hides the latency of updates by exploiting memory-level parallelism (as we saw in Section 5.1.2). Finally, the other three applications do not exhibit mixed reuse, so HIERARCHICAL-HTA does not significantly improve performance over FLAT-HTA.

5.7.4 HTA on multiprogrammed workloads

We evaluate FLAT-HTA's impact on co-running applications by running and 8-thread `ycsb` and an 8-thread SPEC OMP2012 application simultaneously on the 16-core system. Threads of both applications are pinned to cores.

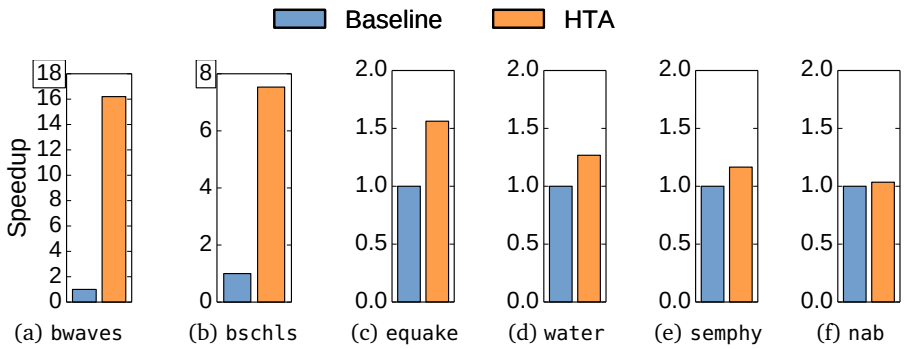
Figure 5.17 summarizes the performance impact of FLAT-HTA. The performance of all SPEC OMP2012 applications except `kdtree` and `md` is not affected by replacing the baseline hash table with FLAT-HTA (as shown in the `others` bar groups). `kdtree`, the most cache-sensitive application, shows that HTA causes *less interference* than the default hash table.

First, when co-running with `ycsb-read`, using FLAT-HTA causes `kdtree`'s performance to improve by 11%, even though FLAT-HTA accelerates `ycsb-read` by 20%, which is therefore performing hash table operations faster. Figure 5.18a gives more insight into this result by reporting the changes in L3 misses per kiloinstruction (MPKI) for both `ycsb` and `kdtree` without and with FLAT-HTA. Despite the higher rate of operations in `ycsb-read`, its MPKI is lower, which leaves more L3 capacity and memory bandwidth for `kdtree`.



(a) w/-read (b) w/-write

Figure 5.18: L3 MPKI when running kdtree and ycsb.



(a) bwaves (b) bschls (c) equake (d) water (e) semphy (f) nab

Figure 5.19: Speedups of HTA-based memoization over the baseline that does not use memoization.

Second, when co-running with ycsb-write, using FLAT-HTA causes kdtree's performance to drop by 15% due to increased cache capacity contention. However, note that ycsb-write is 4.2× faster with FLAT-HTA, so it performs hash table operations much faster than the baseline. Figure 5.18b shows that both ycsb-write versions incur a similar L3 MPKI. Overall, these results show that HTA does not introduce undue L3 and main memory memory pressure.

HTA on memoization 5.7.5

We leverage HTA for memoization, and compare its performance with the baseline implementation and both conventional hardware and software memoization techniques.

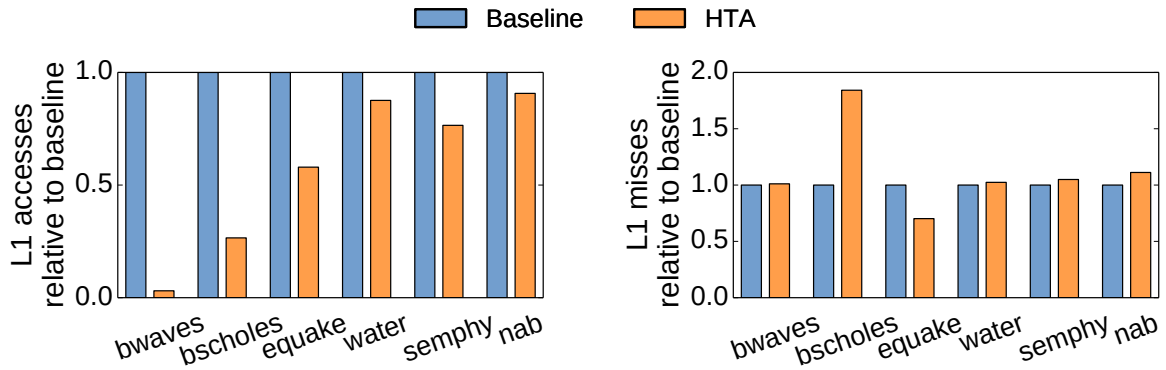
HTA vs. baseline: Figure 5.19 compares the performance of HTA-based memoization over the baseline benchmarks, which do not perform memoization. HTA improves performance substantially, by 16× on bwaves, 7.5× on bscholes, 56% on equake, 27% on water, 17% on semphy, and 4% on nab.

Table 5.6 provides more details into these results by reporting per-function statistics. For example, in bwaves, memoizing the pow function provides most of the benefits. pow takes thousands of instructions to calculate x^y if x is close to 1 and y is around 0.75, which is common in bwaves. Memoizing pow contributes to 99.9% of the instruction reduction in bwaves.

Beyond reducing execution time, HTA reduces the number of L1 cache accesses significantly, as shown in Figure 5.20a: L1 access reductions range from 9% on nab to 97% on bwaves. This happens because the L1 accesses saved through memoization hits exceed the additional L1 accesses incurred by memoization operations. Moreover, HTA does not incur much extra capacity contention in L1 caches. Figure 5.20b shows that HTA increases L1 data cache misses by less than 5% overall. One exception

	Function	Instrs per func call	# of `hta_lookups`	Hit rate
bwaves	slowpow	485160	579	48.0%
	pow	12947	371813	96.3%
	halfulp	77	301	1.3%
	exp1	28	14443	21.7%
bscholes	CDNF	193	15547145	99.6%
	exp	115	7840164	100.0%
	logf	56	7773573	100.0%
equake	phi0	119	7953687	100.0%
	phi1	123	7953687	100.0%
	phi2	118	7953687	100.0%
water	exp	116	7806240	100.0%
semphy	get	19	67123200	94.6%
nab	exp	81	29150496	49.6%
	slowexp_avx	14756	0	N/A

Table 5.6: Per-function breakdown of `hta_lookups`.



(a) L1 data cache accesses

(b) L1 data cache misses

Figure 5.20: L1 data cache accesses and misses of HTA-based memoization relative to those of the baseline (without memoization).

is `bscholes`, which incurs 84% more L1 misses on HTA. However, this is not significant, because the baseline's L1 miss rate is only 0.2%. In fact, such misses bring in valuable memoization data that in the end improve performance by 7.5×. On `equake`, HTA even reduces L1 data misses by 30%, as the functions it memoizes have a larger data footprint than their memoization tables.

HTA vs. conventional hardware memoization: We implement a conventional hardware memoization technique that leverages HTA's ISA and pipeline changes, but uses a dedicated storage buffer like prior work [49, 253] instead of using the memory system to store memoization tables. Beyond its large hardware cost, the key problem of conventional hardware memoization is its lack of flexibility: a too-large memoization buffer wastes area and energy, while a too-small memoization buffer sacrifices memoization potential.

Figure 5.21 quantifies this issue by showing the performance of hardware memoization across a range of memoization buffer sizes: 1, 4, 16, and 64 KB. The buffer is associative, and entries are dynamically shared among all memoized functions. We optimistically model a 1-cycle buffer access latency.

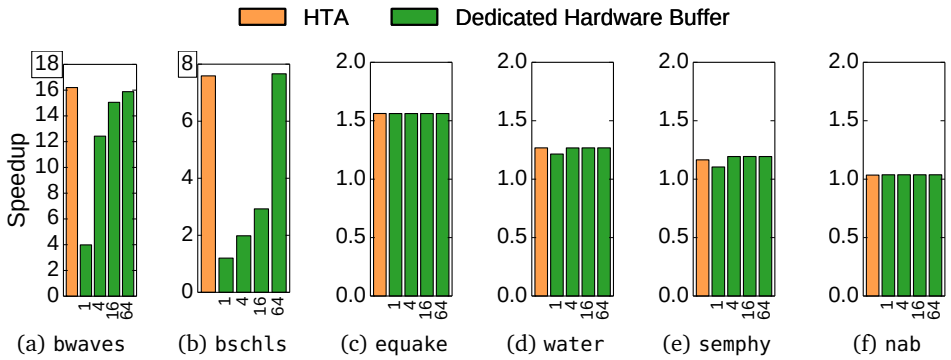


Figure 5.21: Per-application speedups of HTA and conventional hardware memoization with different dedicated buffer sizes (in KB).

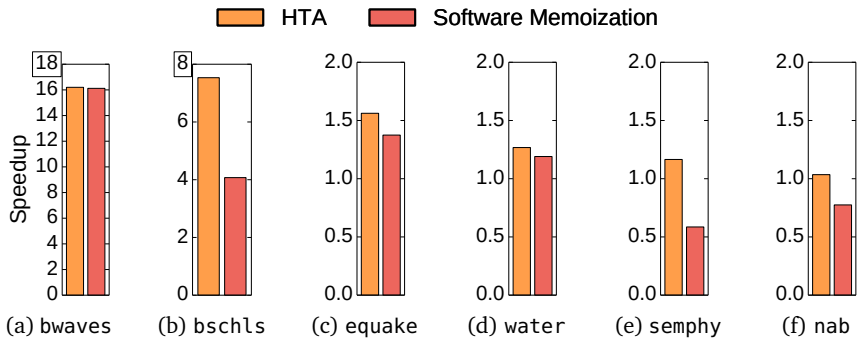


Figure 5.22: Per-application speedups of HTA and software memoization with per-function direct-mapped hash tables.

Figure 5.21 shows that applications are quite sensitive to memoization buffer size: 1 KB is sufficient for `equake` and `nab`, while `water` and `semphy` prefer at least 4 KB, and `bwaves` and `bscholes` prefer at least 64 KB. Smaller buffers than needed by the application result in increased memoization misses and sacrifice much of the speedup of memoization.

Finally, Figure 5.21 shows that HTA matches hardware memoization with a dedicated storage size of 64 KB on all applications. This is achieved even though HTA does not require any dedicated storage, saving significant area and energy. The tradeoff is that storing memoization tables in memory causes longer lookup latencies than using a dedicated buffer. However, these lookup latencies are small, as they mostly hit on the L1 or L2, and branch prediction effectively hides this latency most of the time.

HTA vs. software memoization: We implement software memoization using function wrappers similar to Suresh et al. [246]. Per-function memoization tables are implemented as fixed-size, direct- mapped hash tables, accessed before calling the function and updated after a memoization miss.

Figure 5.22 compares the performance of HTA and software memoization. HTA outperforms software memoization by 85% on `bscholes`, 14% on `equake`, 7% on `water`, 2× on `semphy`, and 34% on `nab`. HTA outperforms software memoization due to its low overheads. For example, `semphy`'s memoizable function runs for 19 instructions on average, too short for software memoization. As a result, software memoization is 41% slower than the baseline. This explains why software memoization needs a careful cost-benefit analysis to avoid performance degradation. By contrast, HTA improves performance by 17% on `semphy`, outperforming software memoization by 2×. Similarly, software memoization makes `nab` 23% slower, while HTA improves performance by 4%.

5.8 Summary

We have introduced HTA, a technique that leverages caches to accelerate hash tables. HTA introduces a table format that exploits the principle: *making the common case fast*: most hash table operations are accelerated with simple hardware, while rare cases are left to a software path.

We have presented two implementations of HTA: FLAT-HTA requires minor changes to cores to reduce runtime overheads, while HIERARCHICAL-HTA requires more modification to caches to further improve spatial locality.

Finally, we have shown that HTA bridges the gap between hardware and software memoization: FLAT-HTA outperforms software memoization by up to 2×, and matches the performance of conventional hardware techniques, but avoids the overheads of large dedicated buffers.

GAMMA 6

In this chapter, we present a solution for dependent reads with irregular data reuse. Instead of proposing a general technique, the solution is tailored to a specific application: sparse-matrix sparse-matrix multiplication (SPMSPM).

SPMSPM is at the heart of a wide range of scientific and machine learning applications. SPMSPM is inefficient on general-purpose architectures, making accelerators attractive. However, prior SPMSPM accelerators use inner- or outer-product dataflows that suffer poor input or output reuse, leading to high traffic and poor performance. These prior accelerators have not explored Gustavson's algorithm, an alternative SPMSPM dataflow that does not suffer from these problems but features irregular memory access patterns that prior accelerators do not support.

We present GAMMA, the Gustavson-Algorithm Matrix-Multiplication Accelerator. GAMMA combines three key features:

- GAMMA uses simple processing elements (PEs) that linearly combine sparse input rows to produce each output row. PEs implement high-radix mergers that combine many input rows (e.g., 64 in our design) in a single pass, reducing work and memory accesses. Instead of expensive high-throughput mergers as in prior work [291], GAMMA uses simple scalar mergers, and relies on Gustavson's row-level parallelism to achieve high throughput efficiently, using tens of PEs to perform many combinations in parallel. Thus, GAMMA concurrently processes *thousands* of compressed sparse *fibers*, variable-sized rows from inputs or partial outputs.
- GAMMA uses a novel storage structure, FIBERCACHE, to efficiently buffer the thousands of fibers required by PEs. FIBERCACHE is organized as a cache to capture Gustavson's irregular reuse patterns. However, FIBERCACHE is managed explicitly, like a large collection of buffers, to fetch missing fibers ahead of time and avoid PE stalls. This saves megabytes of dedicated on-chip buffers.
- GAMMA dynamically schedules work among PEs to ensure high utilization and minimize memory traffic despite the irregular nature of Gustavson's algorithm.

While Gustavson's algorithm is an improvement over other dataflows, it still incurs excessive traffic on some inputs. To address this issue, we propose a preprocessing technique (Section 6.3). that combines row reordering and selective tiling of one matrix input. Preprocessing improves GAMMA's performance and avoids pathologies across the full range of inputs.

In summary, we make the following contributions:

- We show that prior SPMSPM accelerators have missed a key dataflow, Gustavson's, which is often more efficient but has less regular access patterns than previously used dataflows.
- We build GAMMA, a novel SPMSPM accelerator that combines specialized PEs, a novel cache-based structure to capture Gustavson's irregular reuse, and dynamic scheduling to achieve high utilization despite irregularity.
- We propose preprocessing techniques that boost GAMMA's effectiveness and avoid Gustavson's pathologies.

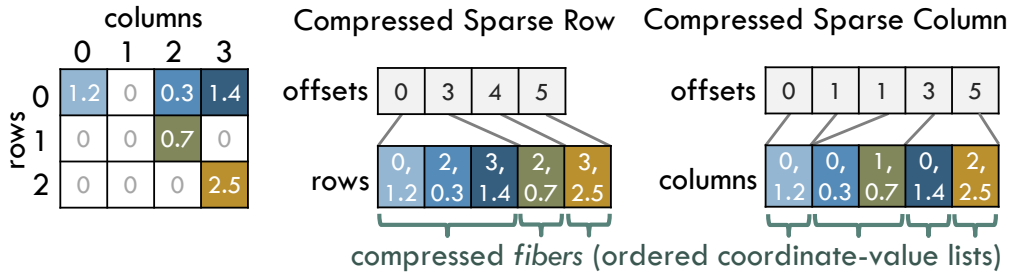


Figure 6.1: Compressed sparse matrix formats.

- We evaluate GAMMA under a broad range of matrices, showing large performance gains and memory traffic reductions over prior systems, as well as higher versatility.

6.1 Motivation

6.1.1 SPMSPM

Sparse matrix-sparse matrix multiplication (SPMSPM) is widely used in deep learning inference [105, 199, 267], linear algebra [29, 140, 274], and graph analytics [93, 136] (including BFS [93], maximum matching [210], cycle detection [283], triangle counting [20], clustering [256], and all-pair shortest paths [47]). It is also a key building block for many other workloads, such as parsing [202], searching [129], and optimization [130].

We first describe the data structures used by SPMSPM and the basic SPMSPM dataflows; then, we review prior accelerators, the optimizations they introduce, and their limitations, motivating the need for a Gustavson-based accelerator.

6.1.2 Compressed sparse data structures

SPMSPM operates on compressed sparse data structures, i.e., structures where only nonzero values are represented. Figure 6.1 shows a sparse matrix encoded in two commonly used formats, compressed sparse row (CSR) and compressed sparse column (CSC). In CSR, rows are stored in a compressed format: each row is an ordered list of *coordinates* (in this case, column indexes) and nonzero *values*, stored contiguously. Indexing into a particular row is achieved through the offsets array, which stores the starting position of each row. CSC is analogous to CSR, but stores the matrix by compressed columns. In general, we call each compressed row or column a *fiber*, represented by a list of coordinates and values, sorted by coordinate.

Compressed sparse data structures introduce two challenges. First, certain kinds of traversals are more efficient than others. These efficient traversals are called *concordant* traversals [248]. For example, a CSR matrix can be traversed row by row, but traversing it by columns or accessing elements at random coordinates is inefficient. Thus, to be efficient, different SPMSPM dataflows impose different constraints on the preferred representation of input and output matrices. Second, SPMSPM relies on indirect accesses (through the offsets array) to variable-sized fibers, and requires combining or intersecting those fibers. These operations are inefficient on CPUs and GPUs.

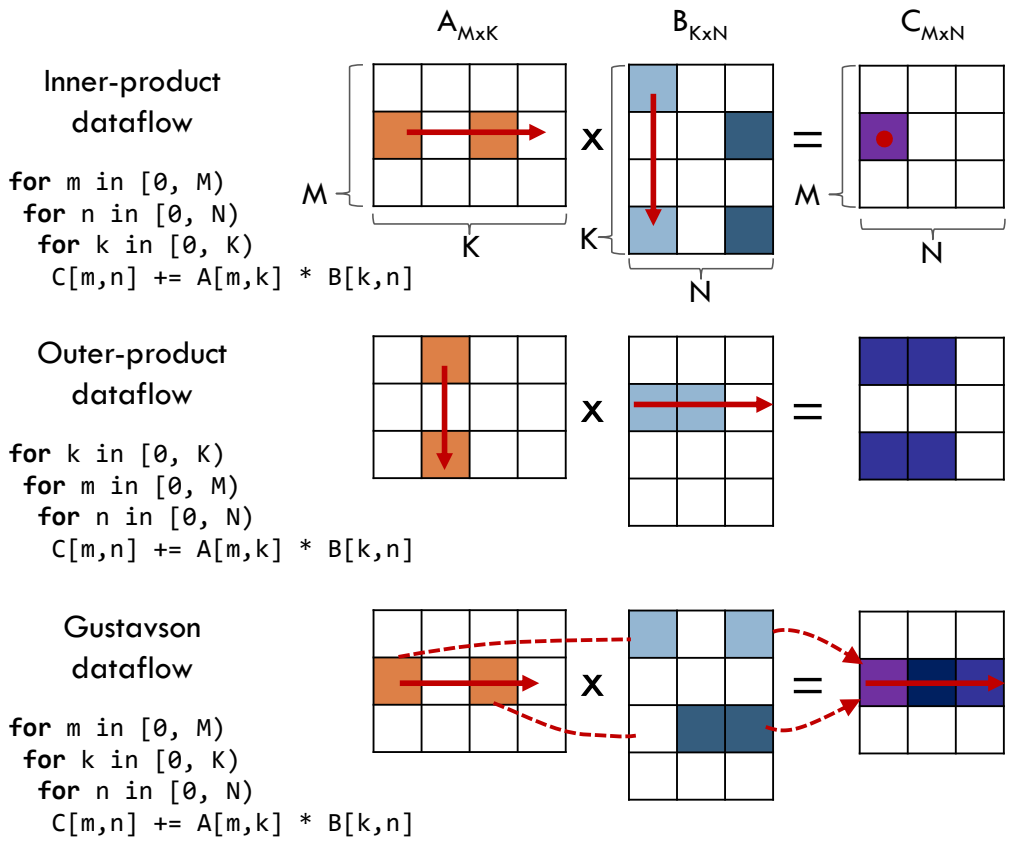


Figure 6.2: Comparison of basic SPMSPM dataflows.

SPMSPM dataflows 6.1.3

Figure 6.2 shows the three basic dataflows for SPMSPM: *inner-product*, *outer-product*, and *Gustavson*. Figure 6.2 also shows the abstract loop nest corresponding to each dataflow (for simplicity, these loop nests assume dense matrices; with compressed sparse matrices, operations are more complex). SPMSPM computes $C_{MxN} = A_{MxK} \times B_{KxN}$ using a triply-nested loop that iterates over A's and B's independent dimensions, M and N, and *co-iterates* over their shared dimension, K. The dataflow is determined by the level of this co-iteration: in inner-product, co-iteration happens at the innermost loop; in outer-product, at the outermost loop; and in Gustavson's at the middle loop.[1]

Inner-product is an *output-stationary*[2] dataflow: it computes the output matrix one element at a time, simultaneously traversing (i.e., co-iterating) rows (m) of A and columns (n) of B. This achieves good output reuse, but poor reuse of the inputs. Since A and B are sparse, this traversal requires an *intersection*, as only nonzeros with matching k coordinates contribute towards the output. Inner-product is relatively efficient when input matrices are nearly dense. But with highly sparse matrices, inner-product is dominated by the cost of intersections, which are inefficient because all elements of the rows and

[1]While Figure 6.2 shows 3 loop nest orders, there are 6 possible orders. The remaining 3 stem from swapping the M and N loops; this merely switches the dimensions in which inputs are traversed, but results in an otherwise identical dataflow. For example, Figure 6.2 shows an inner-product dataflow where A is traversed by rows and B by columns; swapping the outer two loops results in an inner-product dataflow where A is traversed by columns and B by rows.

[2]We use the *-stationary terminology from [54].

columns must be traversed, even though there are few *effectual* intersections, i.e., cases where both elements are nonzero. For example, in Figure 6.2, intersecting row A_1 and column B_2 is completely ineffectual, as they have no nonzeros with the same coordinate.

Outer-product, by contrast, is an *input-stationary* dataflow: it computes the output one *partial matrix* at a time, traversing each column of A (k) and row of B (k) once and computing a full $M \times N$ matrix that incorporates all their contributions in the output. Then, all K partial output matrices are combined to produce the final output matrix. Outer-product achieves good reuse of input matrices. Additionally, outer-product avoids inner-product's inefficiencies of ineffectual intersections: each co-iteration of column m of A and row n of B is ineffectual only when either is all-zeros, which is unlikely. However, outer-product is limited by poor output reuse: the combined size of the partial output matrices is often much larger than the final output, so they cause significant traffic. Moreover, combining these partial output matrices is a complex operation.

Gustavson, finally, is a *row-stationary* dataflow: it computes the output matrix one row at a time, by traversing a row of A (m) and scaling and reducing, i.e., linearly combining, the rows of B (k) for which the row of A has nonzero coordinates. Specifically, given a row A_i with nonzeros a_{ij}, output row C_i is produced by linearly combining B's rows B_j, i.e., $C_i = \sum_j a_{ij} B_j$. Gustavson is more efficient because it avoids the extremes of inner- and outer-product dataflows. While Gustavson does not get as much reuse of a single value as either inner- or outer-product dataflows, it gets reuse of modestly sized rows. Unlike outer-product, Gustavson requires combining partial output *rows* rather than partial output matrices, a simpler operation on much smaller intermediates that more easily fit on-chip; and unlike inner-product, Gustavson avoids ineffectual intersections and poor input reuse.

Finally, Gustavson has an additional advantage over the other dataflows: its inputs and outputs are all in a consistent format, CSR.[3] By contrast, inner- or outer-product require one input to be in CSR and the other in CSC, to support efficient concordant traversals by rows and columns. We do not evaluate this issue further, but for compound operations (e.g., matrix exponentiation), having different formats requires expensive operand transformations, e.g., converting CSC to CSR, that rival the cost of accelerated SpMSpM [59].

6.1.4 SpMSpM accelerators

Despite the advantages of Gustavson's algorithm, prior SpMSpM accelerators have focused on inner- and outer-product dataflows, seeking to maximize reuse of one operand. These designs incorporate different optimizations over the basic dataflow they adopt to mitigate its inefficiencies.

Accelerators like UCNN [109] and SIGMA [207] implement *inner-product* SpMSpM, These designs are built around hardware support to accelerate intersections: UCNN traverses compressed sparse data structures, while SIGMA uses a hardware-friendly bitmap-based fiber representation to further accelerate intersections. To counter poor input reuse, some designs also tile input matrices [107] to fit on-chip. While these designs achieve much higher throughput than CPUs and GPUs when matrices are relatively dense (as is typical in e.g. deep learning inference), they suffer from the algorithmic inefficiencies of ineffectual intersections on sparse matrices.

By contrast, accelerators including OuterSPACE [196], SpArch [291], and SCNN [199] implement an *outer-product* dataflow, and take different approaches to mitigate its inefficiencies. To reduce merge complexity, OuterSPACE divides partial output matrices in rows, then merges rows individually. However,

[3] Or CSC in the alternative Gustavson dataflow; see footnote 1.

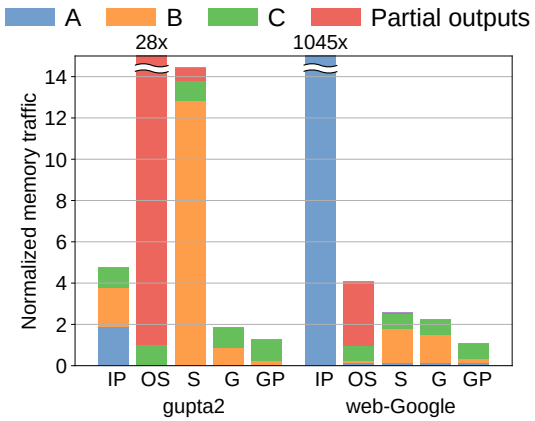


Figure 6.3: Off-chip traffic of tiled inner-product (IP), OuterSPACE (OS), SpArch (S), and GAMMA without/with preprocessing (G/GP).

OuterSPACE produces a large amount of off-chip traffic due to partial outputs, which do not fit on-chip. SpArch, by contrast, is built around a very complex high-throughput, high-radix merger that can merge up to 64 partial matrices per pass, and two main techniques to use this merger well: pipelining the production of the partial output matrices and merging to avoid spilling them off-chip, and employing a matrix condensing technique that reduces the number and size of partial output matrices. Scaling up SpArch is inefficient because its throughput is bottlenecked by the merger, and scaling up the merger's throughput incurs *quadratic* area and energy costs. Instead, GAMMA achieves high throughput with linear cost by performing many independent merges in parallel. On highly sparse matrices, SpArch often achieves nearly perfect off-chip traffic because it can produce fewer than 64 partial output matrices; however, on large or less-sparse matrices, SpArch incurs high output traffic as it needs to spill many partial outputs off-chip. SpArch's matrix condensing technique also sacrifices reuse of the B matrix, which can add significant traffic.

Finally, some prior work adopts a hybrid of inner- and outer-product: ExTensor [108] is a flexible accelerator for tensor algebra that combines outer-product at the chip level, and inner-product with individual PEs. This approach requires tiling to be used well, and though this hierarchical design eliminates more ineffectual work than a pure inner-product design (by skipping entire ineffectual tiles when possible), it still suffers from the drawbacks of the dataflows it adopts.

Despite these optimizations, prior SPMSPM accelerators are saddled by the fundamental inefficiencies of the dataflows they adopt. Figure 6.3 shows this by comparing the memory traffic of different accelerators when squaring (multiplying by itself) two representative sparse matrices: `gupta2` (49 MB, density 1×10^{-3}), which is relatively dense, and `web-Google` (58 MB, density 6×10^{-6}), which is highly sparse. We compare five accelerators with similar hardware budgets (see Section 6.4 for methodology details): (1) IP uses an inner-product dataflow with optimally tiled input matrices; (2) OS is OuterSPACE; (3) S is SpArch; (4) G is GAMMA without preprocessing; and (5) G+P is GAMMA with preprocessing. Each bar shows traffic normalized to compulsory traffic (i.e., the traffic all designs would incur with unbounded on-chip memory, equivalent to reading the inputs and writing the output matrix). Traffic is broken down by data structure: reads of A and B, writes of the final output C, and writes and reads of partial outputs.

Figure 6.3 shows that, despite their optimizations, prior accelerators have significant drawbacks: IP works reasonably well on the denser matrix, but is inefficient on the sparser one because of many sparse tiles resulting from the hard-to-predict distribution of nonzeros. OuterSPACE suffers from partial outputs, while SpArch incurs less traffic on partial outputs, but more on matrix B. They both perform

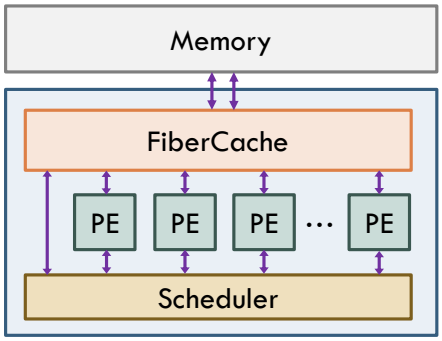


Figure 6.4: GAMMA overview.

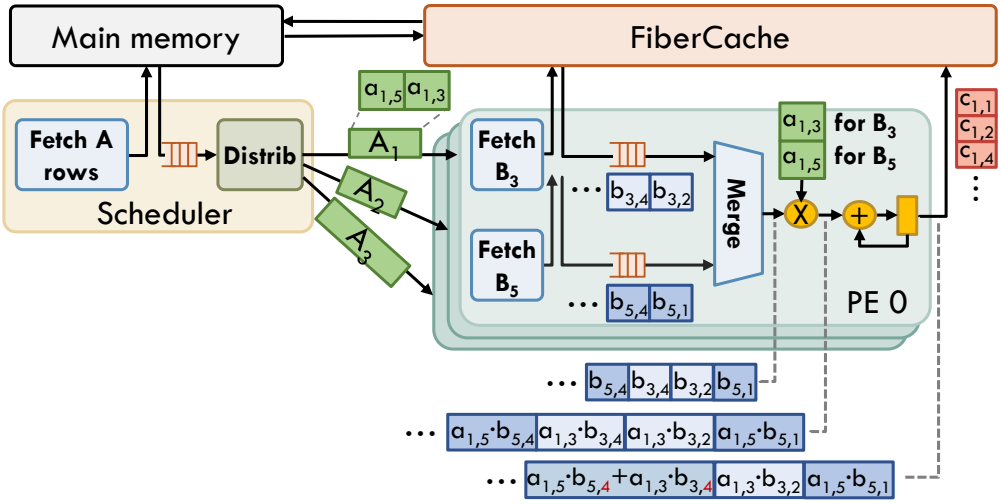


Figure 6.5: Example showing GAMMA's operation.

well on the sparser matrix, but not on the denser one. Even without preprocessing, GAMMA outperforms them all *solely by virtue of using Gustavson's dataflow*. But GAMMA supports matrix tiling and reordering techniques like prior work, as we will see in Section 6.3. With these preprocessing techniques, GAMMA achieves even larger traffic reductions. Finally, since SPMSPM is memory-bound, this lower bandwidth translates to higher performance (Section 6.5).

6.2 GAMMA

Figure 6.4 shows an overview of GAMMA. GAMMA consists of multiple *processing elements (PEs)* that linearly combine sparse fibers; a *scheduler* that adaptively distributes work across PEs; and a FIBERCACHE that captures irregular reuse of fibers.

Figure 6.5 illustrates GAMMA's operation through a simple example that shows how the first few elements of an output row are produced. GAMMA always operates on *fibers*, i.e., streams of nonzero values and their coordinates sorted by coordinate. First, the scheduler fetches matrix A's rows and dispatches them to PEs. Each PE then computes a *linear combination* of rows of B to produce a row of output C. For example, in Figure 6.5, the scheduler dispatches row A_1 to PE 0. Row A_1 has only two nonzeros, at coordinates 3 and 5. Therefore, PE 0 linearly combines rows B_3 and B_5. Figure 6.5 shows how the first few elements of each row are combined. First, the B_3 and B_5 fibers are streamed

from the FIBERCACHE. (The FIBERCACHE retains these fibers, so subsequent uses do not incur off-chip traffic.) Then, these fibers are *merged* into a single fiber, with elements ordered by their shared (column, i.e., N-dimension) coordinate. Each element in the merged fiber is then scaled by the coefficient of A's row corresponding to the fiber element's row (K) coordinate. Finally, consecutive values with the same column (N) coordinate are summed up, producing the output fiber. Figure 6.5 shows the values of these intermediate fibers needed to produce the first three elements of output row C_1.

GAMMA PEs have a bounded *radix*, R: PEs can linearly combine up to R input fibers in a single pass (though Figure 6.5 illustrates the combination of only two fibers, GAMMA PEs have a higher radix, 64 in our implementation). When a row of A has more than R nonzeros, the scheduler breaks the linear combination into multiple rounds. For example, with $R = 64$, processing a row of A with 256 nonzeros would be done using four 64-way linear combinations followed by a 4-way linear combination. Each of the initial linear combinations produces a *partial output fiber*, which is then consumed by the final linear combination. The FIBERCACHE buffers these partial output fibers, avoiding off-chip traffic when possible.

GAMMA PEs use high-radix, modest-throughput mergers: PEs have two key design parameters: *radix*, i.e., how many input fibers they can take; and *throughput*, i.e., how many input and output elements they can consume and produce per cycle. These parameters are given by the radix and throughput of the PE's hardware *merger*, which takes R input fibers and produces a single output fiber with all the elements of all the input fibers, sorted by their coordinates. Radix and throughput choices have a substantial impact on PE and system efficiency, and on memory system design, so we discuss them first.

Implementing high-radix merges is cheap: merger area grows linearly with radix. A high radix in turn makes computation more efficient: it allows many linear combinations to be done in a single pass, and increasing the radix reduces the number of merge rounds and partial output fibers needed. For example, linearly combining 4096 fibers with radix-64 PEs would require 65 PE invocations in a depth-2 tree; using radix-2 PEs would require 4095 PE invocations in a depth-12 tree. The radix-64 PEs would produce one partial output fiber, whereas the radix-2 PEs would produce 11, increasing FIBERCACHE traffic by about an order of magnitude.[4]

Since higher radix mergers are larger, there is a tradeoff between the size and power cost of the merger and both PE performance (measured in number of passes required) and FIBERCACHE traffic (due to partial output fibers). With current technology, the sweet spot balancing overall PE cost and performance occurs around $R = 64$.

Another consideration is the *throughput* of the merger. High-throughput mergers is costly: merger area and energy grow *quadratically* with throughput, as producing N output elements per cycle requires the merger to consume up to N elements from a single input, and up to N^2 comparisons. Thus, GAMMA uses simple pipelined merge units that produce one output and consume one input per cycle, and achieves high throughput by doing many independent linear combinations in parallel, e.g., by using multiple PEs to process distinct rows of A.

This design tradeoff stands in contrast to SpArch [291], the SPMSPM accelerator that comes closest to GAMMA's efficiency. Because SpArch merges partial output *matrices* rather than fibers, it cannot exploit row-level parallelism, and implements a single high-throughput merger that dominates area and limits throughput. GAMMA and SpArch both implement radix-64 mergers, but while in GAMMA each PE's merger is about the same area as its floating-point multiplier, SpArch spends 38× more area on the merger than on multipliers.

[4] In highly sparse matrices, fibers rarely have matching coordinates, so the size of the linear combination of R fibers is close to the sum of the size of the partial output fibers (whereas for dense fibers, the final output would be a factor of R smaller).

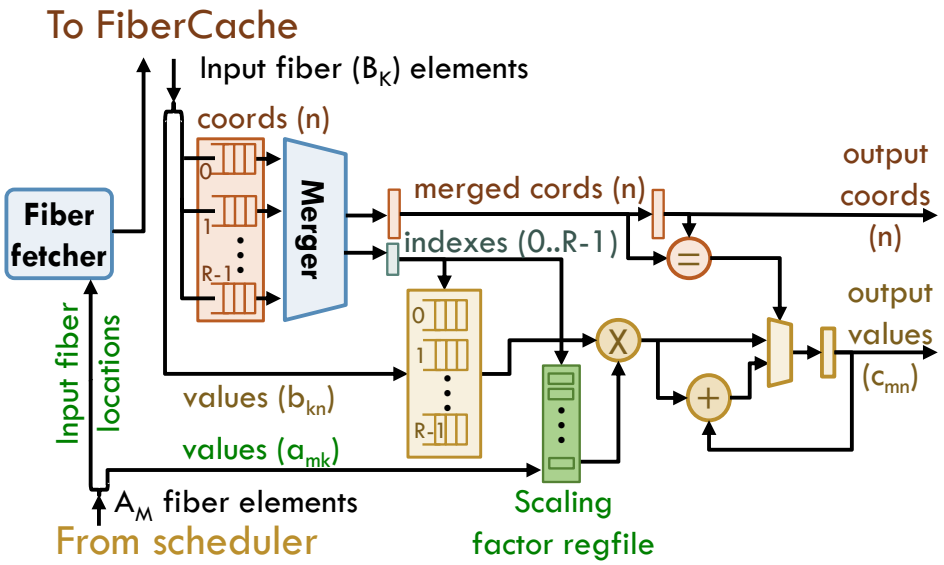


Figure 6.6: GAMMA's PE architecture.

GAMMA's on-chip storage captures irregular reuse across many fibers: Although GAMMA's PEs are efficient, the combination of high-radix and many PEs to achieve high throughput means that GAMMA's memory system must support efficient accesses to a large number of concurrent fibers. For example, a system using 32 radix-64 PEs can fetch 2048 input fibers concurrently. GAMMA relies on a novel on-chip storage idiom, FIBERCACHE, to support the irregular reuse patterns of Gustavson's algorithm efficiently. FIBERCACHE takes two key design decisions: sharing a single structure for all fibers that may have reuse, and combining caching and explicit decoupled data orchestration [201] to avoid large fetch buffers.

GAMMA processes four types of fibers: rows of A and B, and partial and final output rows of C. Rows of A and final output rows of C have no reuse, so they are streamed from/to main memory. Rows of B and partial output rows of C have reuse, but different access patterns: rows of B are read-only and are accessed potentially multiple times (depending on A's nonzeros), whereas partial output fibers, which need to be further merged to produce a final output row, are produced and consumed by PEs, typically within a short period of time. The FIBERCACHE buffers both types of fibers within a single structure, instead of having separate buffers for inputs and outputs. Sharing capacity across fiber types helps because different matrices demand a widely varying share of footprint for partial outputs, but requires careful management to maximize reuse.

FIBERCACHE is organized as a highly banked cache, which allows it to flexibly share its capacity among many fibers or fiber fragments. However, FIBERCACHE is managed using the explicit data orchestration idioms common in accelerators [201]: the fibers needed by each PE are fetched ahead of time, so that when the PE reads each input fiber element, it is served from the FIBERCACHE. This avoids PE stalls and lets the FIBERCACHE pull double duty as a latency-decoupling buffer. This feature is important because, due to the large number of concurrent fibers processed, implementing such buffering separately would be inefficient: with 32 radix-64 PEs and an 80 ns main memory, implementing these buffers would require about 2 MB of storage, a large fraction of the 3 MB FIBERCACHE we implement.

6.2.1 Processing element

Figure 6.6 details the design of GAMMA's PE. The PE linearly combines up to R fibers incrementally. Operation begins with a request from the scheduler, which streams up to R input fiber descriptors: for

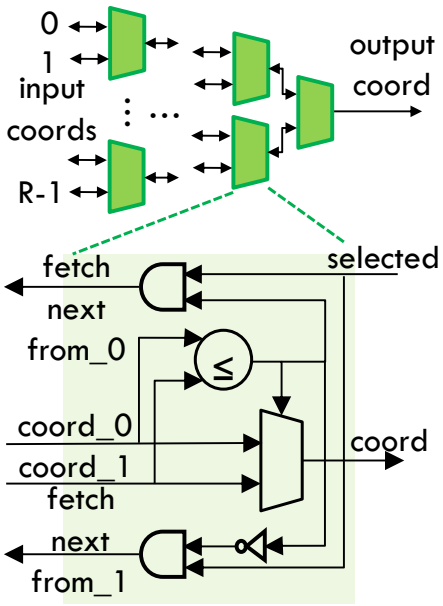


Figure 6.7: High-radix merger implementation.

each input, the scheduler specifies its starting location, size, and a scaling factor. If the input fiber is a row of B, B_k, the scaling factor is value a_{mk}; otherwise, the input fiber is a previously generated partial output, and its scaling factor is 1.0. The PE stores scaling factors in a register file, and input fiber locations in the fiber fetcher.

The fiber fetcher then begins streaming input fibers from the FIBERCACHE. The read elements are streamed into two sets of circular buffers: coordinates (N) are staged as inputs to the high-radix merger, while values are buffered separately. Each set has R buffers, one of each *way* of the merger. Since the FIBERCACHE ensures low access latency, these buffers are small and incur low overheads.

The **merger** consumes the minimum coordinate (N) among the heads of its R input buffers, and outputs the coordinate together with its way index, i.e., a value between 0 and $R-1$ that identifies which input fiber this coordinate came from.

The way index is used to read both the corresponding value from the value buffer and the scaling factor. The PE then multiplies these values. Finally, the coordinate and value are processed by an accumulator that buffers and sums up the values of same-coordinate inputs. If the accumulator receives an input with a different coordinate, it emits the currently buffered element, which is part of the output fiber.

Figure 6.7 shows the implementation of the merger. The merger is organized as a balanced binary tree of simple compute units. Each unit has an integer comparator for coordinates, and merges coordinate streams incrementally. This design achieves a small area cost, e.g., 67% of a 64-bit floating point multiplier for a radix of 64, and achieves an adequately high frequency.

Unlike prior mergers [222, 291] with throughputs that are high on average but are very sensitive to coordinate distribution, GAMMA's merger maintains a constant 1-element-per-cycle throughput. Thus, in steady state, the PE consumes one input fiber element per cycle and performs one scaling operation. This achieves high utilization of its most expensive components, the multiplier and the merger.

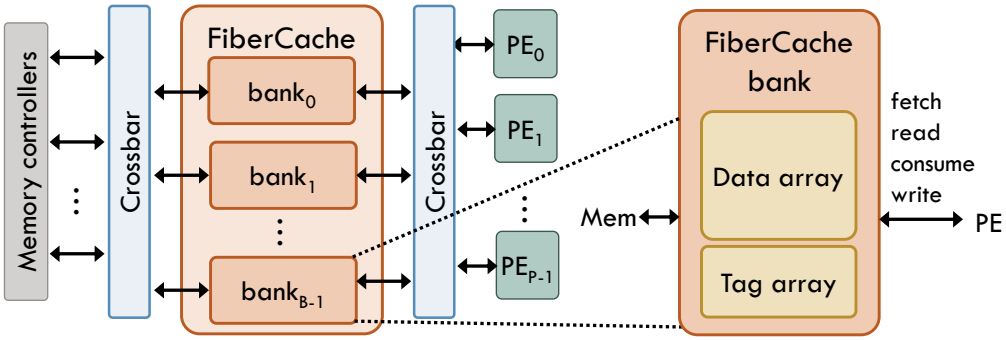


Figure 6.8: FIBERCACHE architecture overview.

6.2.2 FIBERCACHE

Figure 6.8 shows the FIBERCACHE design and interface. FIBERCACHE builds upon a cache: it has data and tag arrays, organizes data in lines, and uses a replacement policy tailored to fiber access patterns. But FIBERCACHE has two key distinct features. First, FIBERCACHE extends the usual `read-write` access interface with primitives that manage data movement more explicitly: `fetch` and `consume`. `fetch` enables decoupled data orchestration by fetching data from memory ahead of execution. Second, to ensure that `read`'s hit in most cases, FIBERCACHE ensures that fetched data is unlikely to be evicted. This is achieved through the replacement policy. This effectively turns a dynamic portion of FIBERCACHE into buffer-like storage, but without the high overheads of separate, statically sized buffers.

Reading rows of B that are not cached incurs a long latency, stalling the PE and hurting performance. FIBERCACHE addresses this issue by decoupling PE data accesses into two steps: `fetch` and `read`. A `fetch` request is sent ahead of execution and places the data into the FIBERCACHE, accessing main memory if needed, and a `read` request directs the actual data movement from FIBERCACHE to the PE. This decouples the accesses to memory and the computation on PEs.

Unlike speculative prefetching, a `fetch` is *non-speculative*: the data accessed by a `fetch` is guaranteed to have a short reuse distance. FIBERCACHE exploits this property through the replacement policy. FIBERCACHE assigns each line a priority in replacement. The priority is managed as a counter: e.g., a 5-bit counter for 32 PEs. A `fetch` request increments the priority, while a `read` request decrements it. Lower-priority lines are selected for eviction. This guarantees that most `read`'s hit in the cache; effectively, the priority is a soft lock on lines that are about to be used. FIBERCACHE uses simple 2-bit SRRIP [121] to break ties among same-priority lines.

Reading and writing partial outputs use the other two primitive requests: `write` and `consume`. Both `write` and `consume` exploit the fact that partial output fibers need not be backed up by memory. Upon a `write`, FIBERCACHE allocates a line without fetching it from memory, updates the data, and sets a dirty bit. A `consume` is similar to a `read`, but instead of retaining the line after the access, FIBERCACHE invalidates the line, without writing it back even though it is dirty.

Banks and interconnect: Since FIBERCACHE must accommodate concurrent accesses from multiple PEs, we use a highly banked design (e.g., 48 banks for 32 PEs). Banks are connected with PEs and memory controllers using crossbars.

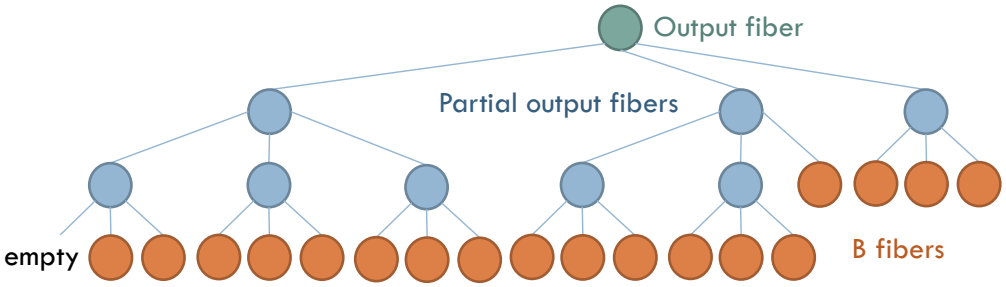


Figure 6.9: Example schedule tree (balanced and top-full) to combine 18 input fibers on PEs with radix 3.

Scheduler 6.2.3

The scheduler assigns compute tasks to PEs to ensure high utilization and minimize memory traffic.

From A to tasks: The scheduler assigns work by traversing the rows of A. Each row of A with fewer nonzeros than the PE radix results in a single task that produces the corresponding output row and writes it directly to main memory.

When a row of A has more nonzeros N than the PE radix R, the scheduler produces a *task tree* that performs an radix-N linear combination in multiple radix-R steps. Figure 6.9 shows an example of a task tree that combines 18 fibers using radix-3 mergers. Each node represents a fiber: the root is the output; leaves are rows of B; and intermediate nodes are the partial output fibers. Edges denote which input fibers (children) contribute to a partial or final output fiber (parent).

The scheduler produces a *balanced, top-full* tree. *Balance* improves merge efficiency: in the common case, the rows of B have similar nonzeros, so a balanced tree results in similarly sized input fibers at each tree level. This is more efficient than a linear tree, which would build an overlong fiber. Moreover, a balanced tree enables more PEs to work on the same row in parallel. (SpArch [291] uses more sophisticated dynamic selection of merge inputs based on their lengths; this is helpful in SpArch because it purposefully constructs uneven partial output matrices, but does not help in GAMMA). *Top-fullness* keeps footprints of partial output fibers low: by keeping the radix of the top levels full, and allowing only the lowest level to have empty input fibers, partial fibers are kept small, reducing the pressure on FIBERCACHE storage.

Mapping tasks to PEs: The scheduler dynamically maps tasks to PEs: when a PE becomes ready to receive a new task, the scheduler assigns is the next available one. Tasks are prioritized for execution in row order, to produce the output in an ordered fashion. For multi-task rows, the scheduler follows a dataflow (i.e., data-driven) schedule: it schedules as many leaf tasks from a single row as needed to fill PEs, and schedules each higher-level task as soon as its input fibers become available. The scheduler prioritizes higher-level tasks over lower-level ones to reduce the footprint of partial outputs.

Staging tasks and data: To avoid stalls when starting up a linear combination, PEs can accept a new task while processing the existing one. When a PE receives a new task, it starts staging its data into its merge buffers, so that it can switch from processing the old task to the new task in a single cycle.

6.2.4 Memory management

Prior to the execution, matrices A and B are loaded into memory, and a sufficiently wide range of address space is allocated for C and partial output fibers.

Since the lengths of partial output fibers are unknown ahead of time, GAMMA allocates and deallocates them dynamically. Upon scheduling a merge that produces a partial output fiber, the scheduler estimates the number of nonzeros of the fiber conservatively, by using the sum of the numbers of nonzeros in all its input fibers. The scheduler then assigns and records the address range of the partial output fiber. This space is only used if the FIBERCACHE needs to evict a partial output, a rare occurrence. The scheduler deallocates the memory when the partial output fiber is consumed. The number of partial outputs is limited to twice the number of PEs, so this dynamic memory management requires negligible on-chip memory.

6.3 Preprocessing for GAMMA

Though Gustavson is a more efficient dataflow than inner- and outer-product, it can incur high traffic. Consider Gustavson on dense operands: processing each row of A requires a complete traversal of every row of B, and results in high memory traffic. This phenomenon is mitigated for sparse operands, because processing a sparse row of A only touches a subset of rows of B, and reuse across those subsets makes the FIBERCACHE effective. Specifically, rows of B enjoy reuse in the FIBERCACHE when multiple nonzeros in A with the same column coordinate appear in nearby rows of A. However, there are two reasons this may not happen: either nearby rows of A contain largely disjoint sets of column coordinates (the matrix lacks structure), so there is minimal reuse of rows of B; or a single row of A has many nonzeros, which requires many rows of B, thrashing the FIBERCACHE.

Prior work has addressed improving such problematic memory access patterns in sparse matrices and graphs using preprocessing techniques like tiling and reordering [116, 124, 203]. Similarly, GAMMA, like prior accelerators, can exploit preprocessing techniques tailored to its memory system and dataflow to further reduce data movement.

To improve data reference behavior, we design two preprocessing techniques for rows of A. *Affinity-based row-reordering* targets disparate adjacent rows of A by reordering rows so that similar rows are processed consecutively. *Selective coordinate-space tiling* breaks (only) dense rows of A into subrows to avoid thrashing, and is applied before row-reordering to extract affinity among the subrows. Both techniques can be implemented by either relying on auxiliary data for indirections or by modifying the memory layout of A. These techniques improve the reuse of sets of rows of B, achieving better versatility and efficiency.

6.3.1 Affinity-based row reordering

Problem definition: We use a score function $S(i,j)$ to represent the affinity of two rows A_i and A_j. $S(i,j)$ is the number of coordinates for which both A_i and A_j have a nonzero value.

Because on-chip storage can hold rows of B corresponding to several rows of A, we are interested in maximizing the affinity of a row with the previous W adjacent rows:

$$\alpha(i) = \sum_{j=max(0,i-W)}^{i-1} S(i,j) \tag{6.1}$$

Algorithm 1: Affinity-based row reordering.

```
Result: Permutation P of row indices
for r ∈ rows do Q.insert(r, 0);
select some r to start, P[0] ← r, Q.remove(r);
for i ∈ [1, M) do
    for u ∈ column coords of row P[i − 1] do
        for r ∈ row coords of column u do
            if r ∈ Q then Q.incKey(r);
    if i > W then
        for u ∈ column coords of row P[i − W − 1] do
            for r ∈ row coords of column u do
                if r ∈ Q then Q.decKey(r);
    P[i] ← Q.pop();
```

We set the window size W to capture the number of rows of B that fit in the FiberCache on average:

$$W = \frac{max\ nnz\ in\ \text{FiberCache}}{nnz\ per\ row_A \cdot nnz\ per\ row_B} \tag{6.2}$$

The goal of the algorithm is to find a proper permutation of rows to maximize the affinity of the whole matrix, which we call α:

$$\alpha = \sum_{i=1}^{M-1} \alpha(i) = \sum_{i=1}^{M-1} \sum_{j=max(0,i-W)}^{i-1} S(i,j) \tag{6.3}$$

Algorithm: Algorithm 1 shows the pseudocode for the affinity-based reordering algorithm. This algorithm is greedy and uses a priority queue (Q) to efficiently find the row with highest affinity. The algorithm produces a permutation P of A's rows.

Selective coordinate-space tiling 6.3.2

Tiling improves input reuse (as each input tile is sized to fit on-chip) at the expense of additional intermediate outputs that must be merged. Tiling *dense* matrices is nearly always a good tradeoff [52, 198] because each input contributes to many outputs, and tiling introduces a large gain in input locality for a few extra fetches of intermediate outputs. However, this no longer holds with sparse matrices, because output traffic often dominates. In other words, tiling sparse rows may reduce traffic to B but produce many partial output fibers that must be spilled off-chip and then brought back to be merged.

Therefore, we apply tiling *selectively*, only to extremely dense rows of A. Specifically, we split rows of A whose footprint to hold rows of B is estimated to be above 25% of the FiberCache capacity (the estimated footprint is the length of A's row times the average number of nonzeros per row of B). Each subrow resulting from this split contributes to a partial output fiber that must be combined eventually. Because these partial output fibers are *not* accessed close in time, they are likely to be spilled. To ensure that the partial output fibers generated by subrows can be combined in just one round, we use the merger's radix R as the tiling factor, i.e., the number of subrows. Rather than splitting rows into evenly-sized subrows, we perform *coordinate-space* tiling [248]: we split evenly in coordinate space, so if column coordinates are in the range $[0, K)$, we create up to R subrows with the ith subrow having the nonzeros within an even subrange $[iK/R, (i+1)K/R)$. Experimentally, we find this creates subrows with higher affinity, improving performance. In large matrices, the resulting subrows may still be large, so this process is repeated recursively.

PEs	32 radix-64 PEs; 1 GHz
FiberCache	3 MB, 48 banks, 16-way set-associative
Crossbars	48×48 and 48×16, swizzle-switch based
Main memory	128 GB/s, 16 64-bit HBM channels, 8 GB/s/channel

Table 6.1: Configuration of the evaluated Gamma system.

	Area (mm^2)	PE component	Area (mm^2)	% PE
32 PEs	2.8	Merger	0.025	29%
Scheduler	0.11	FP Mul	0.037	43%
FiberCache	22.6	FP Add	0.007	8%
Crossbars	3.1	Others	0.018	20%
Total	28.6	PE total	0.087	100%

Table 6.2: Area breakdown of Gamma (left) and one PE (right).

6.4 Methodology

System: We evaluate a Gamma system sized to make good use of high-bandwidth memory and consume similar levels of resources compared to prior accelerators [196, 291], in order to make fair comparisons. Our system has 32 radix-64 PEs, a 3 MB FiberCache, and a 128 GB/s High-Bandwidth Memory (HBM) interface. The system runs at 1 GHz. Table 6.1 details the system's parameters. We built a cycle-accurate simulator to evaluate Gamma's performance and resource utilization.

We measure Gamma's area by writing RTL for the PEs and scheduler. We then synthesize this logic using yosys [269] and the 45 nm FreePDK45 standard cell library [143]. We use CACTI 7.0 [23] to model the FiberCache at 45 nm. We model the same swizzle-switch networks [229] as in prior work [196]. Table 6.2 shows Gamma's area breakdown, which we contrast with prior work in Section 6.5.

Baselines: We compare Gamma with two state-of-the-art accelerators, OuterSPACE and SpArch. We built detailed memory traffic models for OuterSPACE and SpArch to understand their key operational differences. We use the same approach as prior work [291] to compare end-to-end performance, by using the same set of matrices used in their evaluations.

We also compare Gamma against the multicore spMspM implementation from Intel MKL [262] (`mkl_sparse_spmm` function), running on a 4-core, 8-thread Skylake Xeon E3-1240 v5, with two DDR4-2400 channels. We do not include GPU results because existing GPU spMspM implementations perform similarly to MKL on CPUs [291].

Inputs: We use two sets of matrices in the evaluation. First, the ***Common*** set of matrices is the set used in the evaluations of OuterSPACE and SpArch. We use the Common set for direct performance comparisons with these accelerators. However, the Common set covers only a fraction of the space of possible inputs: these matrices are square, and most are very sparse, with a maximum mean of 26 nonzeros per row. This is not representative of other commonly used matrices, and *masks the inefficiencies of outer-product*

Matrix	Nnz/row	Rows	Cols
NotreDame_actors	3.75	392,400	127,823
relat8	3.86	345,688	12,347
Maragal_7	25.63	46,845	26,564
degme	43.81	185,501	659,415
EternityII_Etilde	116.42	10,054	204,304
nemsemm1	267.17	3,945	75,352

Matrix (Square)	Nnz/row	Rows	Matrix (Square)	Nnz/row	Rows
gupta2	68.45	62,064	x104	80.4	108,384
vsp_bcsstk30_500	69.12	58,348	m_t1	99.96	97,578
Ge87H76	69.85	112,985	ship_001	111.58	34,920
raefsky3	70.22	21,200	msc10848	113.36	10,848
sme3Db	71.6	29,067	opt1	124.97	15,449
Ge99H100	74.8	112,985	ramage02	170.31	16,830

Table 6.3: Characteristics of the extended set of matrices.

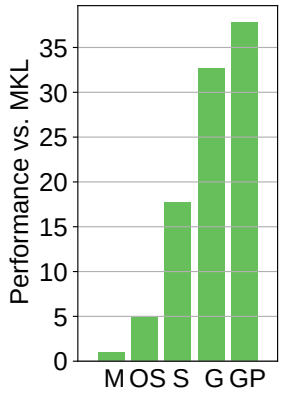


Figure 6.10: Gmean speedup vs. MKL on common set for OuterSPACE (OS), SpArch (S), and GAMMA without and with preprocessing (G/GP).

designs. To evaluate the designs with a broader range of inputs, we construct the ***Extended*** set of matrices, which includes 18 matrices from SuiteSparse Matrix Collection [146]. Table 6.3 lists these matrices, which include non-square and square matrices with a wider range of sparsities and sizes. We evaluate $A \times A$ for square matrices (like prior work), and $A \times A^T$ for non-square matrices.

Evaluation 6.5

Performance on Common-set matrices 6.5.1

Figure 6.10 reports the performance of all accelerators on common-set matrices. Each bar shows the gmean speedup over our software baseline, MKL. Note that common-set matrices are highly sparse and thus well suited for OuterSPACE and SpArch. On these matrices, GAMMA (with preprocessing) is gmean 2.1× faster than SpArch, 7.7× faster than OuterSPACE, and 38× faster than MKL. Even without preprocessing, which makes GAMMA gmean 16% faster, GAMMA outperforms SpArch by 1.84×, OuterSPACE by 6.6×, and MKL by 33×.

Figure 6.11 further shows the per-matrix speedups of GAMMA (with preprocessing) over MKL. GAMMA outperforms MKL by up to 184×.

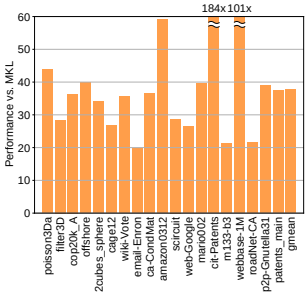


Figure 6.11: Speedups of GAMMA over MKL on the common set.

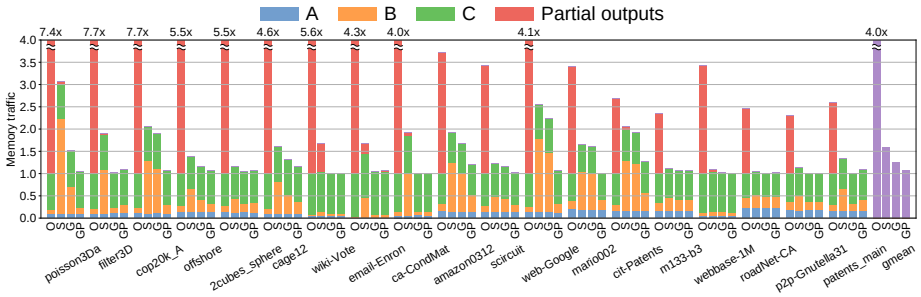


Figure 6.12: Off-chip traffic on common-set matrices of OuterSPACE (O), SpArch(S), and GAMMA without/with preprocessing (G/GP) (lower is better).

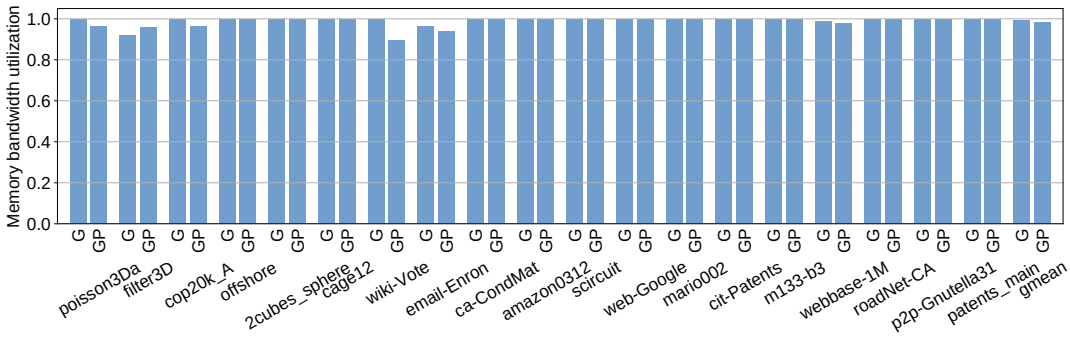


Figure 6.13: Memory bandwidth utilization on common-set matrices of GAMMA without and with preprocessing (G/GP).

Figure 6.12 and Figure 6.13 explain how GAMMA outperforms SpArch, and OuterSPACE: through a combination of reducing memory traffic and improving memory bandwidth utilization.

Figure 6.12 reports the memory traffic of OuterSPACE, SpArch, and GAMMA without and with preprocessing. Each group of bars shows results for one matrix. Traffic is normalized to the compulsory traffic, which would be incurred with unbounded on-chip storage: fetching A, the needed rows of B, and writing C. Each bar is broken down into four categories: reads of A or B, writes of C, and reads and writes of partial outputs.

Figure 6.12 shows that GAMMA incurs close-to-optimal traffic: across all inputs, it is only 7% higher than compulsory (i.e., minimum) traffic with preprocessing, and 26% higher without preprocessing. By contrast, SpArch is 59% higher, and OuterSPACE is 4× higher. OuterSPACE suffers writes and reads to partial matrices. SpArch reduces partial output traffic over OuterSPACE, but incurs high traffic on B for two reasons. First, to reduce partial output traffic, SpArch preprocesses A to produce a schedule that worsens the access pattern to B. Second, SpArch splits its storage resources across data types (e.g., merge and prefetch buffers), leaving only part of its on-chip storage (around half a megabyte) to exploit reuse of B. By contrast, GAMMA's shared FIBERCACHE allows B's rows to use more on-chip storage when beneficial. Because GAMMA's partial outputs are rows, it has negligible partial output traffic, and its main overhead comes from imperfect reuse of B.

Figure 6.13 further illustrates how memory bandwidth translates to performance. Because GAMMA's PEs achieve very high throughput (processing inputs and outputs at a peak rate of 768 GB/s) and Gustavson's algorithm does not have compute-bound execution phases, GAMMA almost always saturates the available 128 GB/s memory bandwidth. By contrast, OuterSPACE and SpArch suffer from the compute

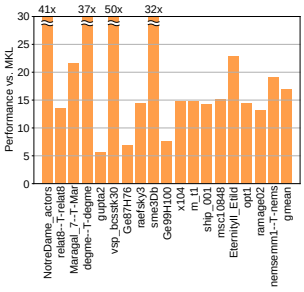


Figure 6.14: Speedups of GAMMA over MKL on the extended set.

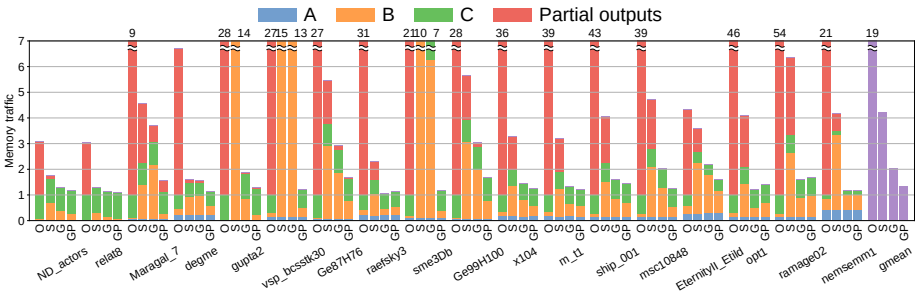


Figure 6.15: Off-chip traffic on extended-set matrices of OuterSPACE (O), SpArch(S), and GAMMA without/with preprocessing (G/GP) (lower is better).

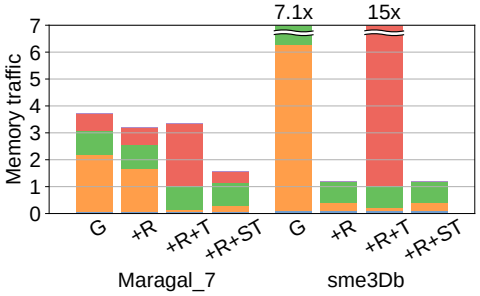


Figure 6.16: Off-chip traffic of GAMMA (G) and GAMMA with different preprocessing on *A*: affinity-based row reordering (+R), selective coordinate-space tiling (+ST), and tiling all rows (+T).

bottleneck of merging all the partial matrices, and hence achieve lower bandwidth utilizations of 48.3% and 68.6%, respectively, on the same matrices. GAMMA's higher performance stems from its lower memory traffic and higher bandwidth utilization.

Performance on Extended-set matrices 6.5.2

To further evaluate the versatility of GAMMA, we use the extended set of matrices, which includes non-square matrices and square matrices more diverse than the common set (Section 6.4)

Figure 6.14 shows the speedups of GAMMA (with preprocessing) over MKL. By exploiting hardware specialization, GAMMA outperforms MKL by gmean 17× and by up to 50×.

Figure 6.15 compares GAMMA with SpArch and OuterSPACE. The off-chip traffic of SpArch and OuterSPACE are are 3× and 14× greater than GAMMA, respectively. This difference is much larger than that in Figure 6.12, because the extended set includes matrices that are denser and have more nonzeros per row. Outer-product struggles on these matrices, as it suffers the excessive memory traffic caused by writing and reading partial output matrices. For instance, on matrices that are relatively dense, such as `msc10848` and `ramage02`, such memory traffic is dominant, reaching 54× over compulsory in OuterSPACE.

Effectiveness of GAMMA preprocessing 6.5.3

Preprocessing improves the performance of GAMMA by 18% on average. Figure 6.16 further illustrates the effects of affinity-based row reordering and selective coordinate-space tiling in two cases. Affinity-based row reordering improves the reuse of *B*. For instance, it contributes to a 6× reduction of traffic on

sme3Db. As Section 6.3.2 explained, tiling *all* rows of A (+T in Figure 6.16) may hurt: it does little harm to Maragal_7 but causes 13× extra traffic on sme3Db due to excessive partial outputs. This is why GAMMA selectively tiles long rows only. Selective coordinate-space tiling reduces traffic of B drastically by tiling dense rows (e.g, on Maragal_7), and also avoids performance pathologies by not tiling sparse rows (e.g., on sme3Db).

6.5.4 GAMMA area analysis

As shown in Table 6.2, the total area of GAMMA is 28.6 mm^2, synthesized with a 45 nm standard cell library. Scaled down to 40 nm, GAMMA's area is 22.6 mm^2, smaller than the 28.5 mm^2 of SpArch at 40 nm and the 87 mm^2 of OuterSPACE at 32 nm. The vast majority of area is used by the FIBERCACHE. This is a good tradeoff for SPMSPM, since the key bottleneck is memory traffic and data movement. The PEs are simple, taking 10% of chip area, and the merger and multiplier are its main components. By contrast, SpArch and OuterSPACE spend far more area on compute resources, e.g., 60% on SpArch's merger.

6.6 Additional Related Work

Much prior work has proposed optimized CPU and GPU implementations for SPMSPM, e.g., using autotuning [260], input characteristics [272], or code generation [140] to pick a well-performing SPMSPM implementation. Intel's MKL [262], which we use in our evaluation, is generally the fastest, or close to the fastest, across input matrices [272]. Although GPUs have higher compute and memory bandwidth than CPUs, SPMSPM is a poor match to the regular data parallelism supported in current GPUs, so GPU frameworks [74, 165, 191] achieve similar SPMSPM performance to CPUs [272, 291].

Most CPU and GPU implementations follow Gustavson's dataflow; variants differ in how they merge rows of B, e.g., using sparse accumulators [92, 139], bitmaps [128], hash tables [188, 191], or heaps [19] to hold outputs. This algorithmic diversity arises because merging fibers is a very expensive operation in general-purpose architectures. At a high level, heaps are space-efficient but very slow, and the other data structures trade lower compute for higher space costs. GAMMA's high-radix merges are both space-efficient and make merges very cheap, avoiding this dichotomy.

As explained in Section 6.1.4, to the best of our knowledge, accelerators earlier than GAMMA did not exploit Gustavson's dataflow. However, MatRaptor [242], which is concurrent with GAMMA, does exploit Gustavson's dataflow. Nonetheless, MatRaptor and GAMMA are very different: MatRaptor does not exploit the reuse of B fibers: it streams such fibers from DRAM and uses them once. By contrast, GAMMA exploits the reuse of B fibers with FIBERCACHE. This adds area costs, but since reusing B fibers is the key way by which Gustavson's dataflow minimizes traffic, GAMMA improves performance significantly. Consequently, on the common-set matrices, MatRaptor outperforms OuterSPACE by only 1.8× [242], worse than SpArch's improvement over OuterSPACE (3.6×), while GAMMA outperforms OuterSPACE by 6.6× even without preprocessing.

Preprocessing of sparse matrices [56, 70, 80, 264] has been studied extensively. Matrix preprocessing on CPUs and GPUs typically targets creating dense tiles [203] to reduce irregularity of partial outputs, disjoint tiles [25] to minimize communication, or balanced tiles [116, 124] to ease load balancing. These techniques differ from GAMMA's: our goal is to improve the locality of B, whereas CPUs and GPUs lack high-radix mergers and have more on-chip storage, making B's locality a less pressing concern.

Summary 6.7

SPMSPM is the basic building block of many emerging sparse applications, so it is crucial to accelerate it. However, prior SPMSPM accelerators use inefficient inner- and outer-product dataflows, and miss Gustavson's more efficient dataflow.

We have presented GAMMA, an SPMSPM accelerator that leverages Gustavson's algorithm. GAMMA uses dynamically scheduled PEs with efficient high-radix mergers and performs many merges in parallel to achieve high throughput, reducing merger area by about 40× over prior work [291]. GAMMA uses a novel on-chip storage structure, FIBERCACHE, which supports Gustavson's irregular reuse patterns and streams thousands of concurrent sparse fibers with explicitly decoupled data movement. We also devise new preprocessing algorithms that boost GAMMA's efficiency and versatility. As a result, GAMMA outperforms prior accelerators by gmean 2.1×, and reduces memory traffic by 2.2× on average and by up to 13×.

Conclusion 7

This thesis has presented novel techniques to extend memory system semantics to accelerate challenging irregular applications. These techniques are tailored to different degrees: COUP and COMMTM are general support for exploiting commutativity to reduce traffic and serialization, while HTA and GAMMA address issues for a specific data structure, hash table, and a specific application, SPMSPM, respectively. In particular, we have presented the following contributions:

- COUP is a general technique that extends coherence protocols to allow local and concurrent single-instruction commutative updates. Specifically, COUP decouples read and write permissions, and introduces commutative-update primitive operations, in addition to reads and writes. With COUP, multiple caches can acquire a line with update-only permission, and satisfy commutative-update requests locally, buffering and coalescing updates. On a read request, the coherence protocol gathers all the local updates and reduces them to produce the correct value before granting read permission.

  COUP integrates seamlessly into existing coherence protocols, requires inexpensive hardware, preserves coherence and does not affect the memory consistency model. Simulation results on a 128-core system show that COUP accelerates update-heavy applications by up to 2.4×. Meanwhile, COUP lowers traffic by up to 20× and reduces memory access latency by up to 12×.
- COMMTM is a hardware transactional memory that exploits semantic commutativity to avoid conflicts that limit scalability in prior hardware speculation techniques. COMMTM extends the coherence protocol and conflict detection scheme to allow multiple cores to perform an unlimited number of user-defined multi-instruction commutative operations concurrently and without conflicts. COMMTM preserves transactional guarantees: COMMTM triggers reductions when non-commutative operations access the same data as commutative ones, so they never observe any partial state or out-of-order updates. We have shown that COMMTM's basic scheme allows as much concurrency as semantic locking, and gather requests allow COMMTM to reduce even more conflicts.

  COMMTM bridges the precision-overhead dichotomy of hardware vs software conflict detection. In return, COMMTM scales many operations that serialize in conventional HTMs, while retaining the low overhead of HTMs. As a result, at 128 cores, COMMTM outperforms an eager-lazy HTM by up to 3.4× and reduces or even eliminates aborts.
- HTA is a technique that leverages caches to accelerate hash tables. HTA introduces simple ISA extensions and hardware changes to address the high runtime overheads and the poor spatial locality of conventional hash table implementations. HTA adopts a hash table format that exploits the characteristics of caches. HTA uses new instructions that leverage existing core structures to accelerate hash table lookups and updates.

  We have presented two implementations of HTA: FLAT-HTA and HIERARCHICAL-HTA. FLAT-HTA adopts a simple, hierarchy-oblivious memory layout and reduces runtime overheads through simple

changes to cores. HIERARCHICAL-HTA uses a multi-level hierarchy-aware layout and requires modifications in caches to further improve spatial locality.
As a result, FLAT-HTA outperforms state-of-the-art implementations of hash-table-intensive applications by up to 2×, while HIERARCHICAL-HTA outperforms FLAT-HTA by up to 35%. Finally, we have shown that HTA bridges the gap between hardware and software memoization: FLAT-HTA outperforms software memoization by up to 2×, and matches the performance of conventional hardware techniques, but avoids the overheads of large dedicated buffers.

- GAMMA is an SPMSPM accelerator that leverages Gustavson's algorithm. GAMMA uses dynamically scheduled PEs with efficient high-radix mergers and performs many merges in parallel to achieve high throughput, reducing merger area by about 40× over prior work [291]. GAMMA uses a novel on-chip storage structure, FIBERCACHE, which supports Gustavson's irregular reuse patterns and streams thousands of concurrent sparse fibers with explicitly decoupled data movement. GAMMA features a dynamic scheduling algorithm to achieve high utilization despite irregularity. We also devise new preprocessing algorithms that boost GAMMA's efficiency and versatility.
  GAMMA is an example of efficient algorithmic implementation enabled by novel architectural support. FIBERCACHE combines the features of buffers and caches, and decouples memory accesses and reads. This allows dependent reads to be processed in a decoupled fashion, and thus enables a specialized hardware implementation of Gustavson's dataflow.
  As a result, GAMMA outperforms prior accelerators by gmean 2.1×, and reduces memory traffic by 2.1× on average and by up to 13×.

These contributions demonstrate that it is possible to extend the memory system semantics with reasonable extra complexity.

7.1 Future Work

This thesis opens exciting avenues for future research. It is promising to combine and generalize the introduced techniques to further improve the performance and the generalizability of the system.

Combining these techniques is promising, as many challenging access patterns often coexist in the same irregular application. One such example is to combine GAMMA and COUP in an accelerator. While GAMMA focuses on the data-dependent *gather reads*, *scatter commutative updates* are the other side of SPMSPM [107], which can leverage COUP to reduce their costs. Such a tradeoff between gather reads and scatter updates has been studied extensively in graph analytics [30, 98, 290]. Architectural support may change the tradeoff significantly and may enable new data representations and compute schedules to improve performance. For example, recent work like PHI [186] and CCache [22], combines COUP's insight of exploiting commutativity with an improved data representation to achieve better performance.

Applying these techniques to other architectures: Though COUP, COMMTM, and HTA are proposed for general-purpose multicores, systems such as GPUs, accelerators, and heterogeneous systems, may benefit from similar architectural support. For instance, though COMMTM is built on hardware transactional memory, the idea of exploiting commutativity to reduce conflicts can be applied to other systems that exploit speculative parallelism [123, 238]. Similarly, GAMMA is a specialized accelerator for SPMSPM, but its key insight, extending caches to benefit from decoupled execution, may be applicable to general-purpose computing.

Generalizing these techniques: There are many other useful properties of operations, data structures, and applications that can be exploited by the architecture. For example, HTA may be extended to accelerate trees, which suffer from the issues caused by small fragments similar to hash tables, and hence may benefit from a flexible memory layout and simple hardware acceleration. GAMMA may be extended to support applications that are beyond the typical SPMSPM. For instance, deep learning workloads consist of mostly tensor/matrix multiplications with characteristics that can be further exploited: they tend to introduce windows and bidirectional passes, and typically process sparse tensors/matrices with structure and higher density. Intersections and unions on data structures that are compressed to formats other than ordered coordinate-value lists, e.g, run-length encoding, may also incur dependent reads with irregular reuse, and hence may benefit from similar architectural support to GAMMA.

These techniques enable other promising research directions as well. For instance, since GAMMA demonstrates in practice the superiority of Gustavson's dataflow on a wide range of matrices, it would be insightful to develop an analytical model for selecting the optimal SPMSPM dataflows for any input matrices. Even with HTA, there are still more opportunities for memoization, e.g., at sub-function granularity, in domain-specific accelerators, or with compiler-based automation. We leave all these explorations to future work.

Bibliography

[1] GReat Images in NASA (GRiN), http://grin.hq.nasa.gov.

[2] HSA Platform System Architecture Specification. Technical report, HSA Foundation, 2015.

[3] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *Proceedings of the 12th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2000.

[4] A. Adya. *Weak consistency: a generalized theory and optimistic implementations for distributed transactions*. PhD thesis, Massachusetts Institute of Technology, 1999.

[5] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: architecture and performance. In *Proceedings of the 22nd annual International Symposium on Computer Architecture (ISCA-22)*, 1995.

[6] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5), 1995.

[7] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2007.

[8] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader. Scalable graph exploration on multicore processors. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC10)*, 2010.

[9] J. H. Ahn, M. Erez, and W. Dally. Scatter-add in data parallel architectures. In *Proceedings of the 11th IEEE international symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[10] S. Ainsworth and T. M. Jones. Graph prefetching using data structure knowledge. In *Proceedings of the International Conference on Supercomputing (ICS'16)*, 2016.

[11] A. R. Alameldeen and D. A. Wood. Adaptive cache compression for high-performance processors. In *Proceedings of the 31st annual International Symposium on Computer Architecture (ISCA-31)*, 2004.

[12] A. R. Alameldeen and D. A. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006.

[13] G. Almási, P. Heidelberger, C. Archer, X. Martorell, C. C. Erway, J. E. Moreira, B. Steinmacher-Burow, and Y. Zheng. Optimization of MPI collective communication on BlueGene/L systems. In *Proceedings of the International Conference on Supercomputing (ICS'05)*, 2005.

[14] L. Alvarez, L. Vilanova, M. Moreto, M. Casas, M. Gonzàlez, X. Martorell, N. Navarro, E. Ayguadé, and M. Valero. Coherence protocol for transparent management of scratchpad memories in shared memory manycore architectures. In *Proceedings of the 42nd annual International Symposium on Computer Architecture (ISCA-42)*, 2015.

[15] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *Proceedings of the 11th IEEE international symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[16] J. Arai, H. Shiokawa, T. Yamamuro, M. Onizuka, and S. Iwamura. Rabbit order: Just-in-time parallel reordering for fast graph analysis. In *Proceedings of the 30th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016.

[17] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-32)*, 1999.

[18] U. Aydonat and T. S. Abdelrahman. Hardware support for relaxed concurrency control in transactional memory. In *Proceedings of the 43rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-43)*, 2010.

[19] A. Azad, G. Ballard, A. Buluc, J. Demmel, L. Grigori, O. Schwartz, S. Toledo, and S. Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6), 2016.

[20] A. Azad, A. Buluç, and J. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015.

[21] P. Bailis, A. Fekete, M. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *VLDB*, 8(3), 2014.

[22] V. Balaji, D. Tirumala, and B. Lucia. POSTER: An Architecture and Programming Model for Accelerating Parallel Commutative Computations via Privatization. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2017.

[23] R. Balasubramonian, A. B. Kahng, N. Muralimanohar, A. Shafiee, and V. Srinivas. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(2), 2017.

[24] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Proceedings of the 29th IEEE International Conference on Data Engineering (ICDE-29)*, 2013.

[25] G. Ballard, A. Druinsky, N. Knight, and O. Schwartz. Hypergraph partitioning for sparse matrix-matrix multiplication. *ACM Transactions on Parallel Computing (TOPC)*, 3(3), 2016.

[26] R. Banakar, S. Steinke, B.-S. Lee, M. Balakrishnan, and P. Marwedel. Scratchpad memory: A design alternative for cache on-chip memory in embedded systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)*, 2002.

[27] A. Basak, S. Li, X. Hu, S. M. Oh, X. Xie, L. Zhao, X. Jiang, and Y. Xie. Analysis and optimization of the memory hierarchy for graph processing workloads. In *Proceedings of the 25th IEEE international symposium on High Performance Computer Architecture (HPCA-25)*, 2019.

[28] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2), 1966.

[29] N. Bell, S. Dalton, and L. N. Olson. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing*, 34(4), 2012.

[30] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler. To push or to pull: On reducing communication and synchronization in graph computations. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*.

[31] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT-17)*, 2008.

[32] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2), 2011.

[33] C. Blundell, A. Raghavan, and M. M. Martin. RETCON: transactional repair without replay. In *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA-37)*, 2010.

[34] J. Bobba, K. E. Moore, H. Volos, L. Yen, M. D. Hill, M. M. Swift, and D. A. Wood. Performance pathologies in hardware transactional memory. In *Proceedings of the 34th annual International Symposium on Computer Architecture (ISCA-34)*, 2007.

[35] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly, 2008.

[36] A. D. Breslow, D. P. Zhang, J. L. Greathouse, N. Jayasena, and D. M. Tullsen. Horton Tables: Fast Hash Tables for In-Memory Data-Intensive Computing. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2016.

[37] I. Calciu, D. Dice, T. Harris, M. Herlihy, A. Kogan, V. Marathe, and M. Moir. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *PODC*. 2013.

[38] I. Calciu, J. Gottschlich, and M. Herlihy. Using elimination and delegation to implement a scalable NUMA-friendly stack. In *HotPar*, 2013.

[39] A. Canning, G. Galli, F. Mauri, A. De Vita, and R. Car. O (n) tight-binding molecular dynamics on massively parallel computers: an orbital decomposition approach. *Computer Physics Communications*, 94(2-3), 1996.

[40] J. L. Carter and M. N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2), 1979.

[41] C. Cascaval, C. Blundell, M. Michael, H. W. Cain, P. Wu, S. Chiras, and S. Chatterjee. Software transactional memory: Why is it only a research toy? *Queue*, 6(5), 2008.

[42] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE transactions on computers*, (12), 1978.

[43] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the 33rd annual International Symposium on Computer Architecture (ISCA-33)*, 2006.

[44] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, C. Kozyrakis, and K. Olukotun. A scalable, non-blocking approach to transactional memory. In *Proceedings of the 13th IEEE international symposium on High Performance Computer Architecture (HPCA-13)*, 2007.

[45] D. Chaiken, J. Kubiatowicz, and A. Agarwal. LimitLESS directories: A scalable cache coherence scheme. In *Proceedings of the 4th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, 1991.

[46] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala. An FPGA memcached appliance. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA-21)*, 2013.

[47] T. M. Chan. More algorithms for all-pairs shortest paths in weighted graphs. *SIAM Journal on Computing*, 39(5), 2010.

[48] D. Chen, N. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, J. J. Parker, et al. The IBM Blue Gene/Q interconnection network and message unit. In *SC*, 2011.

[49] P. Chen, K. Kavi, and R. Akl. Performance enhancement by eliminating redundant function execution. In *ANSS-39*, 2006.

[50] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, et al. Scheduling threads for constructive cache sharing on CMPs. In *Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2007.

[51] S. Chen, P. B. Gibbons, M. Kozuch, and T. C. Mowry. Log-based architectures: using multicore to help software behave correctly. *ACM SIGOPS Operating Systems Review*, 45(1), 2011.

[52] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014.

[53] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE transactions on computers*, 44(5), 1995.

[54] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd annual International Symposium on Computer Architecture (ISCA-43)*, 2016.

[55] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey. Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2012.

[56] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*.

[57] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. Denovo: Rethinking the memory hierarchy for disciplined parallelism. In

Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques (PACT-20), 2011.

[58] B.-S. Choi and J.-D. Cho. Partial resolution for redundant operation table. *Microprocessors and Microsystems*, 32(2), 2008.

[59] S. Chou, F. Kjolstad, and S. Amarasinghe. Automatic generation of efficient sparse tensor format conversion routines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2020.

[60] D. Citron and D. G. Feitelson. Hardware memoization of mathematical and trigonometric functions. Technical report, The Hebrew University of Jerusalem, 2000.

[61] A. Clements, M. F. Kaashoek, and N. Zeldovich. RadixVM: Scalable address spaces for multithreaded applications. In *EuroSys*, 2013.

[62] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler. The scalable commutativity rule: Designing scalable software for multicore processors. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP-24)*, 2013.

[63] C. Click. AzulâĂŹs experiences with hardware transactional memory. In *Transactional Memory Workshop*, 2009.

[64] D. Connors and W.-M. Hwu. Compiler-directed dynamic computation reuse: rationale and initial results. In *Proceedings of the 32nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-32)*, 1999.

[65] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC-1)*, 2010.

[66] F. J. Corbato. A Paging Experiment with the Multics System. In *MIT Project MAC Report MAC-M-384*, 1968.

[67] N. C. Crago and S. J. Patel. OUTRIDER: Efficient memory latency tolerance with decoupled strands. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA-38)*, 2011.

[68] D. Culler, J. Singh, and A. Gupta. *Parallel computer architecture: a hardware/software approach*. Morgan Kaufmann, 1999.

[69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*.

[70] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference*, 1969.

[71] A. T. Da Costa, F. M. G. França, and E. M. C. Filho. The dynamic trace memoization reuse technique. In *Proceedings of the 9th International Conference on Parallel Architectures and Compilation Techniques (PACT-9)*, 2000.

[72] W. Dally. GPU computing: To exascale and beyond. Invited talk. *Supercomputing, New Orleans*, 2010.

[73] W. J. Dally and B. P. Towles. *Principles and practices of interconnection networks*. Elsevier, 2004.

[74] S. Dalton, L. Olson, and N. Bell. Optimizing sparse matrixâĂŤmatrix multiplication for the gpu. *ACM Transactions on Mathematical Software (TOMS)*, 41(4), 2015.

[75] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1), 2011.

[76] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, et al. A clustered manycore processor architecture for embedded and accelerated applications. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[77] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proceedings of the 6th USENIX symposium on Operating Systems Design and Implementation (OSDI-6)*, 2004.

[78] L. Della Toffola, M. Pradel, and T. R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[79] J. Demmel and H. D. Nguyen. Fast reproducible floating-point summation. In *ARITH*, 2013.

[80] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita. Compressing graphs and indexes with recursive graph bisection. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.

[81] D. Dill, A. Drexler, A. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *Proceedings of the 10th International Conference on Computer Design (ICCD)*, 1992.

[82] Y. Ding and Z. Li. A compiler scheme for reusing intermediate computation results. In *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2004.

[83] N. Duong, D. Zhao, T. Kim, R. Cammarota, M. Valero, and A. V. Veidenbaum. Improving cache management policies using dynamic reuse distances. In *Proceedings of the 45th annual IEEE/ACM international symposium on Microarchitecture (MICRO-45)*, 2012.

[84] A. Duran, J. Corbalán, and E. Ayguadé. Evaluation of OpenMP task scheduling strategies. In *IWOMP-4*, 2008.

[85] F. Ellen, Y. Lev, V. Luchangco, and M. Moir. SNZI: Scalable nonzero indicators. In *PODC*, 2007.

[86] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proceedings of the 38th annual International Symposium on Computer Architecture (ISCA-38)*, 2011.

[87] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[88] B. Fan, D. G. Andersen, and M. Kaminsky. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[89] S. Franey and M. Lipasti. Accelerating atomic operations on GPGPUs. In *NOCS-7*, 2013.

[90] M. Frigo, P. Halpern, C. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *Proceedings of the 21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2009.

[91] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. *ACM SIGARCH Computer Architecture News*, 18(2SI), 1990.

[92] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1), 1992.

[93] J. R. Gilbert, S. Reinhardt, and V. B. Shah. High-performance graph algorithms from parallel sparse matrices. In *International Workshop on Applied Parallel Computing*, 2006.

[94] J. R. Goodman, J.-t. Hsieh, K. Liou, A. R. Pleszkun, P. Schechter, and H. C. Young. PIPE: a VLSI decoupled architecture. In *Proceedings of the 12th annual International Symposium on Computer Architecture (ISCA-12)*, 1985.

[95] J. R. Goodman and P. J. Woest. The Wisconsin multicube: a new large-scale cache-coherent multiprocessor. *ACM SIGARCH Computer Architecture News*, 16(2), 1988.

[96] D. Gope, D. J. Schlais, and M. H. Lipasti. Architectural Support for Server-Side PHP Processing. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*, 2017.

[97] A. Gottlieb, R. Grishman, C. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU Ultracomputer: Designing a MIMD Shared Memory Parallel Computer. *IEEE Trans. Comput.*, 100(2), 1983.

[98] S. Grossman, H. Litz, and C. Kozyrakis. Making pull-based graph processing performant. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2018.

[99] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3), 1978.

[100] Z. S. Hakura and A. Gupta. The design and analysis of a cache architecture for texture mapping. In *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA-24)*, 1997.

[101] E. G. Hallnor and S. K. Reinhardt. A unified compressed memory hierarchy. In *Proceedings of the 11th IEEE international symposium on High Performance Computer Architecture (HPCA-11)*, 2005.

[102] T. J. Ham, J. L. Aragón, and M. Martonosi. DeSC: Decoupled supply-compute communication management for heterogeneous architectures. In *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*, 2015.

[103] T. J. Ham, L. Wu, N. Sundaram, N. Satish, and M. Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49)*, 2016.

[104] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual International Symposium on Computer Architecture (ISCA-31)*, 2004.

[105] S. Han, H. Mao, and W. J. Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[106] T. Hayes, O. Palomar, O. Unsal, A. Cristal, and M. Valero. Vector extensions for decision support dbms acceleration. In *Proceedings of the 45th annual IEEE/ACM international symposium on Microarchitecture (MICRO-45)*, 2012.

[107] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.

[108] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher. Extensor: An accelerator for sparse tensor algebra. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.

[109] K. Hegde, J. Yu, R. Agrawal, M. Yan, M. Pellauer, and C. Fletcher. Ucnn: Exploiting computational reuse in deep neural networks via weight repetition. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018.

[110] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2008.

[111] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th annual International Symposium on Computer Architecture (ISCA-20)*, 1993.

[112] M. D. Hill. Is transactional memory an oxymoron? *Proceedings of the VLDB Endowment*, 1(1), 2008.

[113] G. Hinton, D. Sager, M. Upton, D. Boggs, et al. The microarchitecture of the Pentium® 4 processor. In *Intel Technology Journal*, 2001.

[114] H. Hoffmann, D. Wentzlaff, and A. Agarwal. Remote store programming. In *Proceedings of the 5th international conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[115] B. Holt, P. Briggs, L. Ceze, and M. Oskin. Alembic: automatic locality extraction via migration. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014.

[116] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan. Adaptive sparse tiling for sparse matrix multiplication. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2019.

[117] J. Huang and D. J. Lilja. Exploiting basic block value locality with block reuse. In *Proceedings of the 5th IEEE international symposium on High Performance Computer Architecture (HPCA-5)*, 1999.

[118] J. Huang, A. Raman, T. B. Jablin, Y. Zhang, T.-H. Hung, and D. I. August. Decoupled software pipelining creates parallelization opportunities. In *Proceedings of the 8th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010.

[119] S. A. R. Jafri, G. Voskuilen, and T. Vijaykumar. Wait-n-GoTM: improving HTM performance by serializing cyclic dependencies. In *Proceedings of the 18th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XVIII)*, 2013.

[120] R. Jain. A comparison of hashing schemes for address lookup in computer networks. *IEEE Transactions on Communications*, 40(10), 1992.

[121] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the 37th annual International Symposium on Computer Architecture (ISCA-37)*, 2010.

[122] M. C. Jeffrey, S. Subramanian, M. Abeydeera, J. Emer, and D. Sanchez. Data-centric execution of speculative parallel programs. In *Proceedings of the 49th annual IEEE/ACM international symposium on Microarchitecture (MICRO-49)*, 2016.

[123] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. A scalable architecture for ordered parallelism. In *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*, 2015.

[124] P. Jiang, C. Hong, and G. Agrawal. A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2020.

[125] N. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August. Speculative separation for privatization and reductions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2012.

[126] W. Jung, J. Park, and J. Lee. Versatile and scalable parallel histogram construction. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation Techniques (PACT-23)*, 2014.

[127] M. Kamruzzaman, S. Swanson, and D. M. Tullsen. Inter-core prefetching for multicore processors using migrating helper threads. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*.

[128] K. Kanellopoulos, N. Vijaykumar, C. Giannoula, R. Azizi, S. Koppula, N. M. Ghiasi, T. Shahroodi, J. G. Luna, and O. Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.

[129] H. Kaplan, M. Sharir, and E. Verbin. Colored intersection searching via sparse rectangular matrix multiplication. In *Proceedings of the twenty-second annual symposium on Computational geometry*.

[130] G. Karypis, A. Gupta, and V. Kumar. A parallel formulation of interior point algorithms. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC94)*, 1994.

[131] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the future of parallel computing. In *Proceedings of the 44th annual IEEE/ACM international symposium on Microarchitecture (MICRO-44)*, 2011.

[132] J. Kelm, D. Johnson, M. Johnson, N. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: an architecture and scalable programming interface for a 1000-core accelerator.

In *Proceedings of the 36th annual International Symposium on Computer Architecture (ISCA-36)*, 2009.

[133] J. H. Kelm, D. R. Johnson, S. S. Lumetta, M. I. Frank, and S. J. Patel. A task-centric memory model for scalable accelerator architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT-18)*, 2009.

[134] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: a hybrid memory model for accelerators. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT-19)*, 2010.

[135] J. H. Kelm, M. R. Johnson, S. S. Lumetta, and S. J. Patel. WayPoint: scaling coherence to 1000-core architectures. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT-19)*, 2010.

[136] J. Kepner, D. Bader, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. *Procedia Computer Science*, 51, 2015.

[137] R. Kessler and J. Schwarzmeier. CRAY T3D: A new dimension for Cray Research. In *COMPCON*, 1993.

[138] V. Kiriansky, H. Xu, M. Rinard, and S. Amarasinghe. Cimple: instruction and memory level parallelism. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT-27)*, 2018.

[139] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe. Tensor algebra compilation with workspaces. In *Proceedings of the 17th IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2019.

[140] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. The tensor algebra compiler. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, volume 1, 2017.

[141] F. Kjolstad and M. Snir. Ghost cell pattern. In *Workshop on Parallel Programming Patterns*, 2010.

[142] K. Knowlton. A fast storage allocator. *CACM*, (8), 1965.

[143] J. Knudsen. Nangate 45nm open cell library. In *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, 2008.

[144] O. Kocberber, B. Grot, J. Picorel, B. Falsafi, K. Lim, and P. Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *Proceedings of the 46th annual IEEE/ACM international symposium on Microarchitecture (MICRO-46)*, 2013.

[145] C. Koelbel. *HPF handbook*. MIT Press, 1994.

[146] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom. The Suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35).

[147] R. Komuravelli, M. D. Sinclair, J. Alsop, M. Huzaifa, M. Kotsifakou, P. Srivastava, S. V. Adve, and V. S. Adve. Stash: Have your scratchpad and cache it too. In *Proceedings of the 42nd annual International Symposium on Computer Architecture (ISCA-42)*, 2015.

[148] L. I. Kontothanassis and M. L. Scott. Software cache coherence for large scale multiprocessors. In *Proceedings of the 1st IEEE international symposium on High Performance Computer Architecture (HPCA-1)*, 1995.

[149] M. Kulkarni, D. Nguyen, D. Prountzos, X. Sui, and K. Pingali. Exploiting the commutativity lattice. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[150] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[151] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips. SQRL: hardware accelerator for collecting software data structures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*.

[152] S. Kumar, H. Zhao, A. Shriraman, E. Matthews, S. Dwarkadas, and L. Shannon. Amoeba-cache: Adaptive blocks for eliminating waste in the memory hierarchy. In *Proceedings of the 45th annual IEEE/ACM international symposium on Microarchitecture (MICRO-45)*, 2012.

[153] G. Kurian. *Locality-aware Cache Hierarchy Management for Multicore Processors*. PhD thesis, MIT, 2014.

[154] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX symposium on Operating Systems Design and Implementation (OSDI-10)*, 2012.

[155] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE transactions on computers*, (9), 1979.

[156] J. Larus, B. Richards, and G. Viswanathan. LCM: Memory system support for parallel language implementation. In *Proceedings of the 6th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[157] J. Laudon and D. Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA-24)*, 1997.

[158] M. Lavasani, H. Angepat, and D. Chiou. An fpga-based in-line accelerator for memcached. *IEEE Computer Architecture Letters*, 13(2), 2014.

[159] A. Lebeck and D. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd annual International Symposium on Computer Architecture (ISCA-22)*, 1995.

[160] C. Leiserson and T. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proceedings of the 22nd ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2010.

[161] S. Li, H. Lim, V. W. Lee, J. H. Ahn, A. Kalia, M. Kaminsky, D. G. Andersen, O. Seongil, S. Lee, and P. 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-42)*, 2015.

[162] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch. Thin servers with smart pipes: designing SoC accelerators for memcached. In *Proceedings of the 40th annual International Symposium on Computer Architecture (ISCA-40)*, 2013.

[163] J. D. Little. A proof for the queuing formula. *Operations Research*, 9(3).

[164] H. Litz, D. Cheriton, A. Firoozshahian, O. Azizi, and J. P. Stevenson. SI-TM: reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014.

[165] W. Liu and B. Vinter. An efficient GPU general sparse matrix-matrix multiplication for irregular data. In *Proceedings of the 28th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[166] S. Lloyd and M. Gokhale. Near Memory Key/Value Lookup Acceleration. Technical report, Lawrence Livermore National Lab (LLNL), 2017.

[167] W. Lu, G. Yan, J. Li, S. Gong, Y. Han, and X. Li. Flexflow: A flexible dataflow accelerator architecture for convolutional neural networks. In *Proceedings of the 23rd IEEE international symposium on High Performance Computer Architecture (HPCA-23)*, 2017.

[168] C.-K. Luk. Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors. In *Proceedings of the 28th annual International Symposium on Computer Architecture (ISCA-28)*, 2001.

[169] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[170] K. Mackenzie, J. Kubiatowicz, M. Frank, W.-J. Lee, W. Lee, A. Agarwal, and M. F. Kaashoek. Exploiting two-case delivery for fast protected messaging. In *Proceedings of the 4th IEEE international symposium on High Performance Computer Architecture (HPCA-4)*, 1998.

[171] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2), 1970.

[172] W. D. Maurer and T. G. Lewis. Hash table methods. *ACM Computing Surveys (CSUR)*, 7(1), 1975.

[173] J. Mayfield, T. Finin, and M. Hall. Using automatic memoization as a software engineering tool in real-world AI systems. In *Proceedings the 11th Conference on Artificial Intelligence for Applications (CAIA'95)*, 1995.

[174] P. McNamee and M. Hall. Developing a tool for memoizing functions in C++. *ACM SIGPLAN Notices*, 33(8), 1998.

[175] P. Melsted and J. K. Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC bioinformatics*, 12(1), 2011.

[176] M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE TPDS*, 15(6), 2004.

[177] D. Michie. Memo functions and machine learning. *Nature*, 218(5136), 1968.

[178] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[179] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, D. A. Wood, et al. LogTM: log-based transactional memory. In *Proceedings of the 12th IEEE international symposium on High Performance Computer Architecture (HPCA-12)*, 2006.

[180] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *Proceedings of the 12th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XII)*, 2006.

[181] J. E. B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, 2006.

[182] T. C. Mowry. Tolerating latency in multiprocessors through compiler-inserted prefetching. *ACM Transactions on Computer Systems (TOCS)*, 16(1), 1998.

[183] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. *ACM Sigplan Notices*, 27(9), 1992.

[184] S. S. Mukherjee, P. Bannon, S. Lang, A. Spink, and D. Webb. The Alpha 21364 network architecture. In *Hot Interconnects 9, 2001.*, 2001.

[185] A. Mukkara, N. Beckmann, M. Abeydeera, X. Ma, and D. Sanchez. Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*, 2018.

[186] A. Mukkara, N. Beckmann, and D. Sanchez. Phi: Architectural support for synchronization-and bandwidth-efficient commutative scatter updates. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.

[187] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An alternative to very large instruction windows for out-of-order processors. In *Proceedings of the 9th IEEE international symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[188] Y. Nagasaka, S. Matsuoka, A. Azad, and A. Buluç. High-performance sparse matrix-matrix products on Intel KNL and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*.

[189] T. Nakaike, R. Odaira, M. Gaudet, M. M. Michael, and H. Tomari. Quantitative comparison of hardware transactional memory for blue gene/q, zenterprise ec12, intel core, and power8. In *Proceedings of the 42nd annual International Symposium on Computer Architecture (ISCA-42)*, 2015.

[190] N. Narula, C. Cutler, E. Kohler, and R. Morris. Phase reconciliation for contended in-memory transactions. In *Proceedings of the 11th USENIX symposium on Operating Systems Design and Implementation (OSDI-11)*, 2014.

[191] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. Cusparse library. In *GPU Technology Conference*.

[192] Q. M. Nguyen and D. Sanchez. Pipette: Improving Core Utilization on Irregular Applications through Intra-Core Pipeline Parallelism. In *Proceedings of the 53rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-53)*, 2020.

[193] J. Nickolls and W. J. Dally. The GPU computing era. In *Proceedings of the 43rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-43)*, 2010.

[194] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[195] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2), 2004.

[196] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. Outerspace: An outer product based sparse matrix multiplication accelerator. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[197] M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the 11th annual International Symposium on Computer Architecture (ISCA-11)*, 1984.

[198] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer. Timeloop: A systematic approach to dnn accelerator evaluation. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[199] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. Scnn: An accelerator for compressed-sparse convolutional neural networks. In *Proceedings of the 44th annual International Symposium on Computer Architecture (ISCA-44)*, volume 45, 2017.

[200] L.-S. Peh and W. J. Dally. A delay model and speculative architecture for pipelined routers. In *Proceedings of the 7th IEEE international symposium on High Performance Computer Architecture (HPCA-7)*, 2001.

[201] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer. Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[202] G. Penn. Efficient transitive closure of sparse matrices over closed semirings. *Theoretical Computer Science*, 354(1), 2006.

[203] A. Pinar and M. T. Heath. Improving performance of sparse matrix-vector multiplication. In *SC'99: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[204] P. Prabhu, S. Ghosh, Y. Zhang, N. P. Johnson, and D. I. August. Commutative set: A language extension for implicit parallel programming. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011.

[205] X. Qian, W. Ahn, and J. Torrellas. Scalablebulk: Scalable cache coherence for atomic blocks in a lazy environment. In *Proceedings of the 43rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-43)*, 2010.

[206] X. Qian, B. Sahelices, and J. Torrellas. OmniOrder: Directory-based conflict serialization of transactions. In *Proceedings of the 41st annual International Symposium on Computer Architecture (ISCA-41)*, 2014.

[207] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[208] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the 34th annual International Symposium on Computer Architecture (ISCA-34)*, 2007.

[209] M. K. Qureshi, D. Thompson, and Y. N. Patt. The V-Way cache: demand-based associativity via global replacement. In *Proceedings of the 32nd annual International Symposium on Computer Architecture (ISCA-32)*, 2005.

[210] M. O. Rabin and V. V. Vazirani. Maximum matchings in general graphs through randomization. *Journal of Algorithms*, 10(4), 1989.

[211] R. Rajwar and J. R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, 2002.

[212] R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd annual International Symposium on Computer Architecture (ISCA-32)*, 2005.

[213] H. E. Ramadan, C. J. Rossbach, and E. Witchel. Dependence-aware transactional memory for increased concurrency. In *Proceedings of the 41st annual IEEE/ACM international symposium on Microarchitecture (MICRO-41)*, 2008.

[214] L. Rauchwerger and D. Padua. The privatizing doall test: A run-time technique for doall loop identification and array privatization. In *Proceedings of the International Conference on Supercomputing (ICS'94)*, 1994.

[215] L. Rauchwerger and D. Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE TPDS*, 10(2), 1999.

[216] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for multi-core processor parallelism*. O'Reilly, 2007.

[217] S. Reinhardt, J. Larus, and D. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st annual International Symposium on Computer Architecture (ISCA-21)*, 1994.

[218] R. F. Resende, D. Agrawal, and A. El Abbadi. Semantic locking in object-oriented database systems. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 1994.

[219] M. C. Rinard and P. C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1996.

[220] H. Rito and J. Cachopo. Memoization of methods using software transactional memory to track internal state dependencies. In *Proceedings of the 8th international conference on the Principles and Practice of Programming in Java (PPPJ-8)*, 2010.

[221] W. Ruan, T. Vyas, Y. Liu, and M. Spear. Transactionalizing legacy code: An experience report using GCC and memcached. In *Proceedings of the 19th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIX)*, 2014.

[222] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. Pileggi, and F. Franchetti. Efficient SPMV operation for large and highly sparse matrices using scalable multi-way merge parallelization. In *Proceedings of the 52nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-52)*, 2019.

[223] D. Sanchez and C. Kozyrakis. SCD: A scalable coherence directory with flexible sharer set encoding. In *Proceedings of the 18th IEEE international symposium on High Performance Computer Architecture (HPCA-18)*, 2012.

[224] D. Sanchez and C. Kozyrakis. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th annual International Symposium on Computer Architecture (ISCA-40)*, 2013.

[225] D. Sanchez, R. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the 15th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XV)*, 2010.

[226] S. R. Sarangi, W. L. Torrellas, Y. Zhou, et al. Reslice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proceedings of the 38th annual IEEE/ACM international symposium on Microarchitecture (MICRO-38)*, 2005.

[227] N. Satish, N. Sundaram, M. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey. Navigating the maze of graph analytics frameworks using massive graph datasets. In *SIGMOD*, 2014.

[228] S. Scott. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the 7th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[229] K. Sewell, R. G. Dreslinski, T. Manville, S. Satpathy, N. Pinckney, G. Blake, M. Cieslak, R. Das, T. F. Wenisch, D. Sylvester, et al. Swizzle-switch networks for many-core systems. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2(2), 2012.

[230] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7), 2010.

[231] J. Shalf, S. Dosanjh, and J. Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2010.

[232] K. S. Shim, M. Lis, M. H. Cho, I. Lebedev, and S. Devadas. Design tradeoffs for simplicity and efficient verification in the Execution Migration Machine. In *Proceedings of the 31st International Conference on Computer Design (ICCD)*, 2013.

[233] J. Shun, G. E. Blelloch, J. T. Fineman, and P. B. Gibbons. Reducing contention through priority updates. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[234] H. V. Simhadri, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and A. Kyrola. Experimental analysis of space-bounded schedulers. In *Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2014.

[235] T. Skare and C. Kozyrakis. Early release: Friend or foe. In *Workshop on Transactional Memory Workloads*, 2006.

[236] J. E. Smith. Decoupled access/execute computer architectures. In *Proceedings of the 9th annual International Symposium on Computer Architecture (ISCA-9)*, 1982.

[237] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th annual International Symposium on Computer Architecture (ISCA-24)*, 1997.

[238] G. S. Sohi, S. E. Breach, and T. Vijaykumar. Multiscalar processors. In *Proceedings of the 22nd annual International Symposium on Computer Architecture (ISCA-22)*, 1995.

[239] E. Solomonik and J. Demmel. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*, 2011.

[240] H. Song, S. Dharmapurikar, J. Turner, and J. Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *Proceedings of the ACM SIGCOMM Conference (SIGCOMM)*, 2005.

[241] D. Sorin, M. Hill, and D. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6(3), 2011.

[242] N. Srivastava, H. Jin, J. Liu, D. Albonesi, and Z. Zhang. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *Proceedings of the 53rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-53)*, 2020.

[243] J. Sugerman, K. Fatahalian, S. Boulos, K. Akeley, and P. Hanrahan. GRAMPS: A programming model for graphics pipelines. *ACM Transactions on Graphics (TOG)*, 28(1), 2009.

[244] M. Sung, R. Krashinsky, and K. Asanović. Multithreading decoupled architectures for complexity-effective general purpose computing. *ACM SIGARCH Computer Architecture News*, 29(5), 2001.

[245] A. Suresh, E. Rohou, and A. Seznec. Compile-Time Function Memoization. In *Proceedings of the 26th international conference on Compiler Construction (CC-26)*, 2017.

[246] A. Suresh, B. N. Swamy, E. Rohou, and A. Seznec. Intercepting functions for memoization: a case study using transcendental functions. *ACM TACO*, 12(2), 2015.

[247] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the IEEE futurebus. *ACM SIGARCH Computer Architecture News*, 14(2), 1986.

[248] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer. Efficient Processing of Deep Neural Networks. *Synthesis Lectures on Computer Architecture*, 15(2), 2020.

[249] S. Tanaka and C. Kozyrakis. High performance hardware-accelerated flash key-value store. In *The 2014 Non-volatile Memories Workshop (NVMW)*, 2014.

[250] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, et al. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. In *Proceedings of the 35th annual IEEE/ACM international symposium on Microarchitecture (MICRO-35)*, 2002.

[251] The ENCODE Project Consortium. An integrated encyclopedia of DNA elements in the human genome. *Nature*, 489(7414), 2012.

[252] P.-A. Tsai, Y. L. Gan, and D. Sanchez. Rethinking the memory hierarchy for modern languages. In *Proceedings of the 51st annual IEEE/ACM international symposium on Microarchitecture (MICRO-51)*, 2018.

[253] T. Tsumura, I. Suzuki, Y. Ikeuchi, H. Matsuo, H. Nakashima, and Y. Nakashima. Design and evaluation of an auto-memoization processor. In *Proceedings of the 25th conference Parallel and Distributed Computing and Networks (PDCN)*, 2007.

[254] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas. SoftSig: software-exposed hardware signatures for code analysis and optimization. In *Proceedings of the 13th international conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII)*, 2008.

[255] R. A. Van De Geijn and J. Watts. SUMMA: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4), 1997.

[256] S. Van Dongen. Performance criteria for graph clustering and Markov cluster experiments. In *NATIONAL RESEARCH INSTITUTE FOR MATHEMATICS AND COMPUTER SCIENCE IN THE*, 2000.

[257] O. Villa, D. Chavarría-Miranda, V. Gurumoorthi, A. Márquez, and S. Krishnamoorthy. Effects of floating-point non-associativity on numerical computations on massively multithreaded systems. *Cray User Group*, 2009.

[258] T. Von Eicken, D. Culler, S. Goldstein, and K. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual International Symposium on Computer Architecture (ISCA-19)*, 1992.

[259] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th annual International Symposium on Computer Architecture (ISCA-19)*, 1992.

[260] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, 2005.

[261] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT-21)*, 2012.

[262] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon PhiâĎć*, pages 167–188. Springer, 2014.

[263] J. Warnock, B. Curran, J. Badar, G. Fredeman, D. Plass, Y. Chan, S. Carey, G. Salem, F. Schroeder, F. Malgioglio, et al. 22nm next-generation IBM System z microprocessor. In *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC)*, 2015.

[264] H. Wei, J. X. Yu, C. Lu, and X. Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*.

[265] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *Computers, IEEE Transactions on*, 37(12), 1988.

[266] T. A. Welch. A technique for high-performance data compression. *Computer*, 6(17), 1984.

[267] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li. Learning structured sparsity in deep neural networks. In *Advances in neural information processing systems*.

[268] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu. Fermi GF100 GPU architecture. *IEEE Micro*, 31(2), 2011.

[269] C. Wolf, J. Glaser, and J. Kepler. Yosys-a free Verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[270] H. Wong, A. Bracy, E. Schuchman, T. M. Aamodt, J. D. Collins, P. H. Wang, G. Chinya, A. K. Groen, H. Jiang, and H. Wang. Pangaea: a tightly-coupled IA32 heterogeneous chip multiprocessor. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT-17)*, 2008.

[271] C.-J. Wu, A. Jaleel, W. Hasenplaugh, M. Martonosi, S. C. Steely Jr, and J. Emer. SHiP: Signature-based hit predictor for high performance caching. In *Proceedings of the 44th annual IEEE/ACM international symposium on Microarchitecture (MICRO-44)*, 2011.

[272] Z. Xie, G. Tan, W. Liu, and N. Sun. IA-SpGEMM: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication. In *Proceedings of the International Conference on Supercomputing (ICS'19)*, 2019.

[273] S. Xu, S. Lee, S.-W. Jun, M. Liu, J. Hicks, and Arvind. BlueCache: A scalable distributed flash-based key-value store. *Proceedings of the VLDB Endowment*, 10(4), 2016.

[274] I. Yamazaki and X. S. Li. On techniques to improve robustness and scalability of a parallel hybrid linear solver. In *International Conference on High Performance Computing for Computational Science*, 2010.

[275] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, et al. Productivity and performance using partitioned global address space languages. In *Proceedings of the 2007 international workshop on Parallel symbolic computation*.

[276] R. M. Yoo, C. J. Hughes, C. Kim, Y.-K. Chen, and C. Kozyrakis. Locality-aware task management for unstructured parallelism: A quantitative limit study. In *Proceedings of the 25th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2013.

[277] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *Proceedings of the ACM/IEEE conference on Supercomputing (SC13)*, 2013.

[278] H. Yu and L. Rauchwerger. Adaptive reduction parallelization techniques. In *Proceedings of the International Conference on Supercomputing (ICS'00)*, 2000.

[279] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proceedings of the VLDB Endowment*, 8(3), 2014.

[280] X. Yu, C. J. Hughes, N. Satish, and S. Devadas. IMP: Indirect memory prefetcher. In *Proceedings of the 48th annual IEEE/ACM international symposium on Microarchitecture (MICRO-48)*, 2015.

[281] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*.

[282] P. Yuan, C. Xie, L. Liu, and H. Jin. PathGraph: A path centric graph processing system. *IEEE Transactions on Parallel and Distributed Systems*, 27(10), 2016.

[283] R. Yuster and U. Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *SODA*.

[284] J. Zebchuk, M. Qureshi, V. Srinivasan, and A. Moshovos. A tagless coherence directory. In *Proceedings of the 42nd annual IEEE/ACM international symposium on Microarchitecture (MICRO-42)*, 2009.

[285] J. Zebchuk, E. Safi, and A. Moshovos. A framework for coarse-grain optimizations in the on-chip memory hierarchy. In *Proceedings of the 40th annual IEEE/ACM international symposium on Microarchitecture (MICRO-40)*, 2007.

[286] G. Zhang and D. Sanchez. Leveraging Hardware Caches for Memoization. *Computer Architecture Letters (CAL)*, 17(1), 2018.

[287] L. Zhang, Z. Fang, and J. Carter. Highly efficient synchronization based on active memory operations. In *Proceedings of the 18th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2004.

[288] M. Zhang, J. Bingham, J. Erickson, and D. Sorin. PVCoherence: Designing flat coherence protocols for scalable verification. In *Proceedings of the 20th IEEE international symposium on High Performance Computer Architecture (HPCA-20)*, 2014.

[289] M. Zhang, A. Lebeck, and D. Sorin. Fractal coherence: Scalably verifiable cache coherence. In *Proceedings of the 43rd annual IEEE/ACM international symposium on Microarchitecture (MICRO-43)*, 2010.

[290] 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)*, 2017.

[291] Z. Zhang, H. Wang, S. Han, and W. J. Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[292] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas. Protozoa: Adaptive granularity cache coherence. In *Proceedings of the 40th annual International Symposium on Computer Architecture (ISCA-40)*, 2013.

[293] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *Proceedings of the 2002 ACM SIGMOD international conference on management of data (SIGMOD)*, 2002.

[294] X. Zhu, W. Han, and W. Chen. GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2015.