Cache Coherent Commutative Operations

by

Webb H. Horn

S.B. Computer Science and Engineering,
Massachusetts Institute of Technology (2014)

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

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2015

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2015

Certified by .....
Daniel Sanchez
Assistant Professor
Thesis Supervisor

Accepted by .....
Albert Meyer
Chairman, Masters of Engineering Thesis Committee

Cache Coherent Commutative Operations

by

Webb H. Horn

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2015, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis presents COUP, a technique that reduces the cost of updates in shared memory systems. In particular, it describes a new cache coherence protocol, MEUSI, and evaluates its performance under simulation in zsim. MEUSI extends the MESI protocol to allow data to be cached in a new *update-only* state, reducing both block-level thrashing and on-chip network traffic under many parallel workflows. COUP permits both single-word and multi-word commutative data operations, which are implemented as x86-64 ISA extensions. To evaluate single-word instructions, this thesis presents a case study of a new reference counting scheme, and for multi-word commutative operations, this thesis describes the design of a commutative memory allocator. COUP and MEUSI confer significant benefits to the reference counting scheme and the memory allocator, both in terms of performance and ease of programming.

Thesis Supervisor: Daniel Sanchez
Title: Assistant Professor

Acknowledgments

This work was conducted in close collaboration with Guowei Zhang, and much of this thesis was adapted from a paper that we co-authored with Professor Sanchez. Guowei painstakingly implemented support for the coherence protocol in zsim and evaluated its performance under several benchmarks. Guowei also performed the ALU sensitivity study in Sec. 6.3.

I'm deeply grateful for the patient and thoughtful guidance provided by Guowei and Professor Sanchez. They never hesitated to give timely, careful, and pragmatic advice at every stage of this project. Harshad Kasture and the rest of our group explained many of the finer points of our group's simulation infrastructure.

Finally, I cherish the unceasing encouragement, generosity, and support of my family. Thank you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cache coherence is pervasive in shared-memory systems. However, current coherence protocols cause significantly more communication and serialization than needed, especially with *frequent updates to shared data*. For example, consider a shared counter that is updated by multiple threads. 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, as shown in Fig. 1-1a. 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.

Prior work has proposed hardware and software techniques to reduce traffic and serialization of updates in parallel systems. In hardware, prior work has mainly focused on *remote memory operations* (RMOs) [58, 59, 86, 96]. RMO schemes send updates to a single memory controller or shared cache bank instead of having the line ping-pong among multiple private caches, as shown in Fig. 1-1b. Though RMOs reduce the cost of updates, they still cause significant global traffic and serialization, and often make reads slower (as remote reads may be needed to preserve consistency [69, 86]).

In this work we leverage two key insights to further reduce the cost of updates. First, many update operations need not read the data they update. Second, update operations are often *commutative*, and can be performed in any order before the

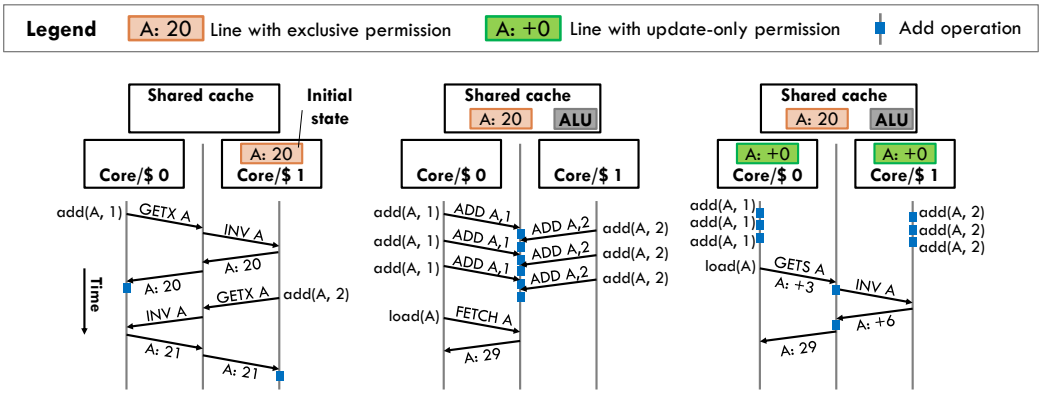
data is read. For instance, in our shared-counter example, multiple additions from different threads can be buffered, coalesced, and delayed until the line is next read. Commutative updates are common in other contexts beyond this simple example.

Two obstacles prevent these optimizations in current protocols. First, conventional coherence protocols only support two primitive operations, reads and writes, so commutative updates must be expressed as a read-modify-write sequence. Second, these protocols do not decouple read and write permissions. Instead, they enforce the *single-writer, multiple-reader invariant*: at a given point in time, a cache line may only have at most one sharer with *read and write* permission, or multiple sharers with read-only permission [33, 87].

We propose COUP (chapter 3), a general technique that extends coherence protocols to allow *local and concurrent commutative updates*. 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, by buffering and coalescing update requests. 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. 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 Fig. 1-1c.

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. 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.

COUP completes a symmetry between hardware and software schemes to reduce the cost of updates. Broadly, software techniques use either *delegation* or *privatization*. Delegation schemes send updates to a single thread [39, 40]. Privatization schemes



(a) Conventional coherence (b) Remote mem ops (c) COUP

Figure 1-1: Example comparing the cost of commutative updates under three 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 remote memory operations, cores send updates to a fixed location, the shared cache in this case. (c) With COUP, caches buffer and coalesce updates locally, and reads trigger a reduction of all local updates to produce the actual value.

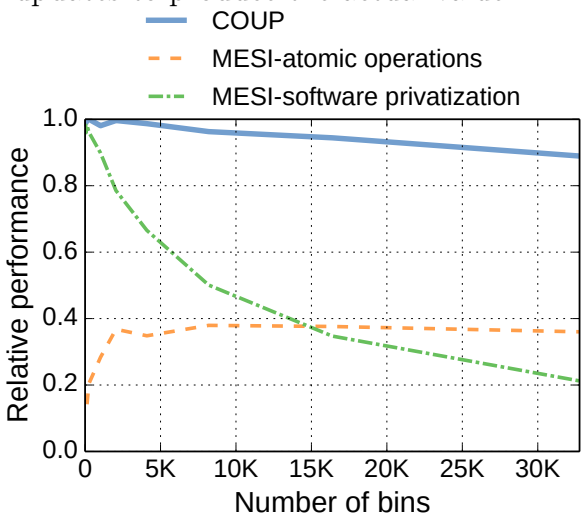


Figure 1-2: Performance of parallel histogramming using atomics, software privatization, and COUP. More bins reduce contention and increase privatization overheads, favoring atomics. COUP does not suffer these overheads, so it outperforms both software implementations.

lower the cost of commutative updates by using thread-local variables [37,45,76]. Each thread updates its local variable, and reads require reducing the per-thread variables. 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.

In this work, we make the following contributions:

- We present COUP, a technique that extends coherence protocols to support concurrent commutative updates to shared state (chapter 3). We apply COUP to extend MSI and MESI, but note that COUP could be used beyond hardware cache coherence (e.g., software coherence protocols in distributed systems).
- We identify several update-heavy parallel applications where existing hardware and software techniques have substantial shortcomings (chapter 4), and discuss how COUP addresses them.
- We present an implementation of COUP that accelerates single-word commutative updates (chapter 5). This implementation adds update-only instructions, extends the coherence protocol, and requires simple ALUs in cache controllers to reduce partial updates.
- We evaluate COUP under simulation, using single- and multi-socket systems (chapter 6). At 128 cores, COUP improves the performance of update-heavy benchmarks by 4%–2.4×, and reduces traffic by up to 20×.
- The design and implementation of a commutative memory allocator that uses commutative set insertion/removal operaitons is described.

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.

Chapter 2

Background

We now discuss prior hardware and software techniques that reduce the cost of updates to shared data.

2.1 Hardware Techniques

Remote memory operations (RMOs) are the most closely related scheme to COUP. Rather than caching lines to be updated, update operations are sent to a fixed location. The NYU Ultracomputer [58] proposed implementing atomic fetch-and-add using adders in network switches, which could coalesce requests from multiple processors on their way to memory. The Cray T3D [64], T3E [86], and SGI Origin [72] implemented RMOs at the memory controllers, while TilePro64 [59] and recent GPUs [91] implement RMOs in shared caches. Prior work has also studied adding caches to memory controllers to accelerate RMOs [96], and proposed data-parallel RMOs [35].

COUP has two key advantages over remote memory operations. First, while remote operations avoid ping-ponging cache lines, they still require sending every update to a shared, fixed location, causing global traffic. Remote operations are also limited by the throughput of the single updater. For example, in Fig. 1-1b, frequent remote-add requests require the shared cache to perform additions near saturation. By contrast, COUP buffers and coalesces updates in local caches. Second, remote operations are

challenging to integrate in cache-coherent systems without affecting the consistency model, as remote updates and cacheable reads and writes follow different paths through the memory hierarchy [69].

Note that COUP's advantages come at the cost of a more restricted set of operations: COUP only works with commutative updates, while RMOs support non-commutative operations, such as fetch-and-add and compare-and-swap. Also, COUP significantly outperforms RMOs only if data is reused (i.e., updated or read multiple times before switching between read- and update-only modes). This is often the case in real applications (chapter 4).

2.2 Software Techniques

Conventional shared-memory programs update shared data using atomic operations for single-word updates, or normal reads and writes with synchronization (e.g., locks or transactions) for multi-word updates. Many software optimizations seek to reduce the cost of updates. Though often presented in the context of specific algorithms, we observe they are instances of two general techniques: *delegation* and *privatization*. We discuss these techniques here, and present specific instances in chapter 4.

Delegation schemes divide shared data among threads and send updates to the corresponding thread, using shared-memory queues [39] or active messages [84, 89]. Delegation is common in architectures that combine shared memory and message passing [84, 92] and in NUMA-aware data structures [39, 40]. Delegation is the software counterpart to RMOs, and is subject to the same tradeoffs: it reduces data movement and synchronization, but incurs global traffic and serialization.

Privatization schemes exploit commutative updates. These 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 [50], OpenMP pragmas, and Cilk Plus hyperobjects [56]). Privatization is generally used when updates are frequent and reads are rare.

Privatization is the software counterpart to COUP, and is subject to similar tradeoffs: it is restricted to commutative updates, and works best when data goes through long update-only phases without intervening reads. Unlike COUP, privatization has two major sources of overhead. First, while COUP is about as fast as a conventional coherence protocol if a line is only updated once before being read (Fig. 1-1c), software reductions are much slower, making finely-interleaved reads and updates inefficient. Second, with N threads, privatized variables increase memory footprint by a factor of N. This makes naive privatization impractical in many contexts (e.g., reference counting). Dynamic privatization schemes [45, 76, 93] can lessen space overheads, but cause additional time overheads.

These overheads often make privatization underperform conventional updates. For instance, Jung et al. [62] 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 few output bins, but works poorly with many bins, as the reduction phase dominates and hurts locality. Fig. 1-2 shows this tradeoff. It compares the performance of histogram implementations using atomic fetch-and-add, privatization, and COUP, when running on 64 cores (see chapter 6 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 8 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 software implementations, COUP robustly outperforms both.

Chapter 3

Extending Cache Coherence to Support Commutative Updates

In this chapter, we describe COUP in two steps. First, in Sec. 3.1, we present the main concepts and operation of COUP through a concrete, simplified example. Then, in Sec. 3.2, we generalize COUP to other coherence protocols, operations, and cache hierarchies.

3.1 Coup Example: Extending MSI

To introduce the key concepts behind COUP, we first 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 later generalize COUP to remove these restrictions.

3.1.1 Structural changes

COUP requires a few additions and changes to existing hardware structures, which we describe below. Fig. 3-1 summarizes these changes.

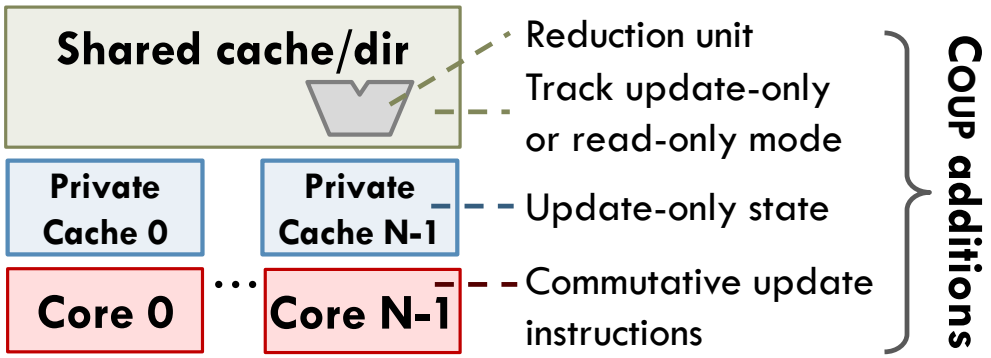


Figure 3-1: Summary of additions and modifications needed to support COUP.

Commutative-update instructions: COUP needs a way for software to convey commutative updates, mainly because conventional atomic instructions (e.g., fetch-and-add) often return the latest value of the data they update. For performance reasons, we use additional instructions. 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. Lower-performance implementations could use cheaper ways to convey updates, such as memory-mapped registers.

Update-only permissions and requests: 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. Fig. 3-2 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 in Fig. 3-2 shows a clear symmetry between the S and U states: 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 describe MUSI's operation in detail later.

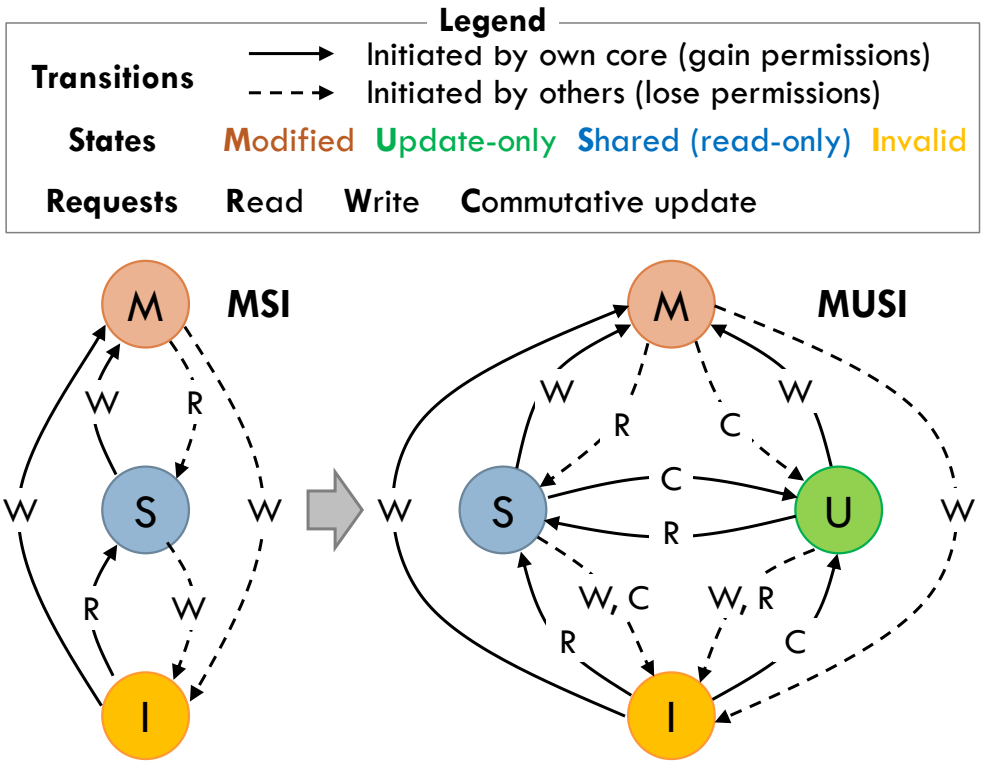


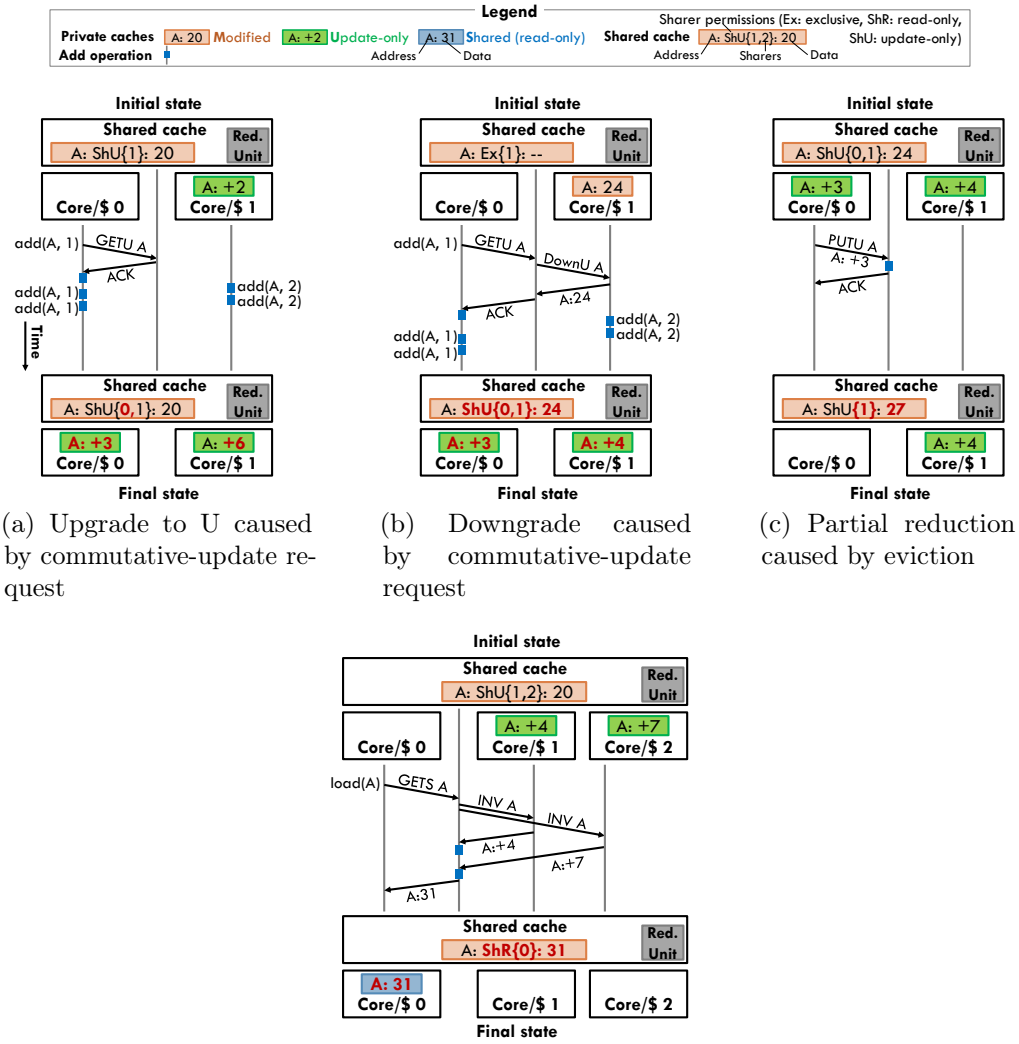
Figure 3-2: State-transition diagrams of MSI and MUSI protocols. For clarity, diagrams omit actions that do not cause a state transition (e.g., R requests in S).

Directory state: Conventional directories must track both the sharers of each line (using a bit-vector or other techniques [41, 82, 94]), 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 only requires 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.

3.1.2 Protocol operation

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



(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 3-3: 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.

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 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 permissions to 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 Fig. 3-3a; or by being downgraded from M, as shown in Fig. 3-3b.

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 Fig. 3-3c: 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 the U state. This triggers a *full reduction*: all caches with update-only permissions 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.

Finally, read requests from any core also trigger a full reduction, as shown in Fig. 3-3d. 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 (Sec. 3.2). In our evaluation, we observe that reduction overheads are small compared to communication latencies.

3.2 Generalizing Coup

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* $(G, \circ)$. 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 different type than the current one.

Larger cache blocks: Supporting multi-word cache 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 possible 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.

Other protocols: COUP can extend protocols beyond MSI by adding the U state. Fig. 3-4 shows how MESI [78] 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 only needs reduction units 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: From now on, we focus on single- and multi-word atomic operations and hardware cache coherence, but note that COUP could apply to a variety of other contexts. For example, COUP could be used in software coherence protocols (e.g., in distributed shared memory).

3.3 Coherence and Consistency

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 [48]:

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 reorderings of memory operations as strictly for commutative updates as it does for stores, COUP does not affect the consistency model. In other words, from the perspective of memory consistency, 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.

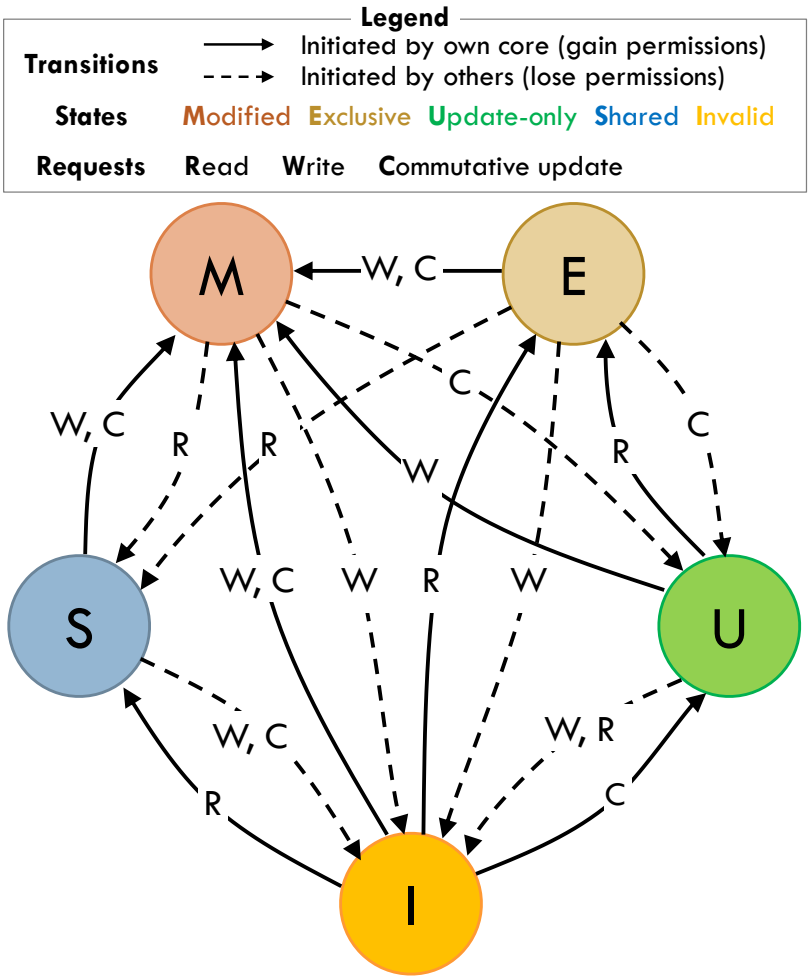


Figure 3-4: State-transition diagram of the MEUSI protocol, used in our evaluation. Just as MESI grants E to a read request if a line is unshared, MEUSI grants M to the first update request if a line is unshared. For clarity, the diagram omits actions that do not cause a state transition (e.g., C requests in U).

Chapter 4

Motivating Applications

In this work, we apply COUP to accelerate single-word updates to shared data, though we also providing a design and implementation for a memory allocator that uses multi-word commutative set operations in chapter 7. 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 chapter 2, COUP is the hardware counterpart to privatization. Privatization schemes create several replicas of variables to be updated. Each thread performs its updates on 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 the memory footprint and pressure on shared caches.

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

4.1 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) [79,80], and their intermediate state is not read. Reduction variables are natively supported in parallel programming languages and libraries such as HPF [67], MapReduce [50], OpenMP [52], TBB [60], and Cilk Plus [56]. Prior work in parallelizing compilers and runtimes has developed a wide array of techniques to detect and exploit reduction variables [61, 79, 80]. 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 in loops that compute the mean or maximum element from a set of values. 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 tree 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 [62], as shown in Fig. 1-2. Yu and Rauchwerger [93] 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 [93]. 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 [35]. However, floating-point operations are not associative or commutative, and the order of operations can affect the final result in some cases [88]. 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 [51].

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 these boundary cells using ghost or halo cells [65], private copies of boundary cells that are 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 [77, 85]. In these cases, partitioning work among threads to minimize communication can be expensive, and is rarely done on shared-memory machines [85]. By reducing the cost of concurrent updates to shared data, COUP helps irregular iterative algorithms as well.

4.2 Interleaved Updates and Reads

Several parallel algorithms read and update shared data within the same phase. Unlike the applications in Sec. 4.1, 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 [34, 42]. 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 [34], or use non-atomic load-*or*-store sequences, which reduce overheads but miss updates, causing some nodes to be visited multiple times [42]. 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 [46], buddy memory allocation [66], and other graph algorithms [70].

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 [44, 45, 53, 75]. Scalable Non-Zero Indicators (SNZIs) [53] 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 fast and allow some concurrency in increments and decrements, but add significant space and time overheads, and need to be carefully tuned.

Refcache [44] 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. When non-zero checks are performed every decrement, COUP reduces L3 invalidations by 4.6% and L2 invalidations by 1.9% over an atomic-based implementation. COUP also allows delayed reference count reads as in Refcache without a software cache.

Chapter 5

Coup Implementation

We now describe our implementation of COUP, which seeks to accelerate single-word commutative updates. Multi-word commutative updates are described in chapter 7.

Operations and data types: We support the following operations:

- Addition of 16, 32, and 64-bit integers, and 32 and 64-bit floating-point values.
- AND, OR, and XOR bit-wise 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 Sec. 4.1. Thus, we do not support *min* or *max*. Finally, we only support one word size for bit-wise operations, because this suffices to express updates to bitmaps of any size (smaller or larger).

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. These instructions produce no register output. Sec. 6.1 describes how we model and simulate these instructions in x86-64.

Reduction unit: Each shared cache bank has a reduction unit that can perform each of the supported operations. Since functional units for the required operations

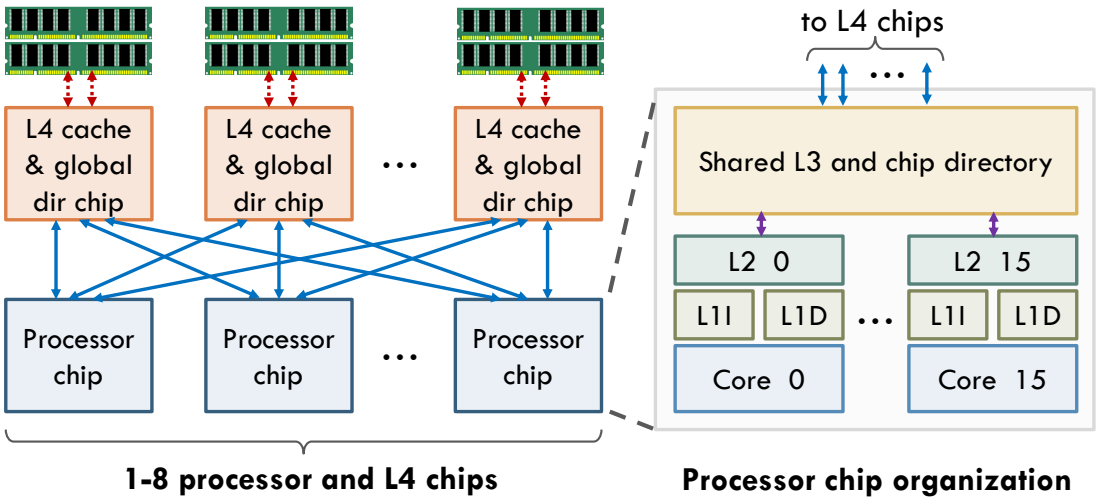


Figure 5-1: Architecture of the simulated SMP system we target.

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.

Hierarchical organization: We evaluate single- and multi-socket systems with up to 128 cores and a four-level cache hierarchy, shown in Fig. 5-1. Each processor chip has 16 cores. Each core has private L1s and a private L2, and all cores in the chip share a banked level-3 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 [90].

To support COUP, L3 and L4 caches use the MEUSI protocol (Fig. 3-4), and each of their bank has a reduction unit. We perform hierarchical reductions as described in Sec. 3.2: 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 could also be implemented with snoopy L3s or an L3 directory without an L4 cache. Not having an L4 cache, however, requires either reading memory for reductions or keeping all the data needed for reductions in the L3 caches. For example,

with snoopy coherence, each line with multiple sharers in U state can have one of the sharers act as its owner, similar to the forwarding state in MESIF [57]. This owner retains the line's data when it transitions into U instead of initializing it with identity elements, and, on an eviction from another cache, needs to capture its partial update and aggregate it to its copy. This is reasonable to do with L3 caches, which already have a reduction unit.

Hardware overheads: Our COUP implementation introduces the following overheads:

1. Eight additional instructions, which reuse the core's existing machinery for atomic operations.
2. Additional tag bits per cache line through all the levels of the hierarchy, to encode additional coherence states and the current update-only operation if the line is in U state. These take four bits per line in our implementation.
3. One reduction unit per L3 and L4 bank, with support for integer and floating-point addition and bit-wise logical operations.

These overheads are modest, and allow significant performance gains for a range of applications.

Chapter 6

Evaluation

6.1 Methodology

Modeled systems: We perform microarchitectural, execution-driven simulation using zsim [83], and model parallel systems with a four-level cache hierarchy as described in chapter 5. We evaluate systems with up to 128 cores (8× 16-core chips). Table 6.1 details the configuration of this system.

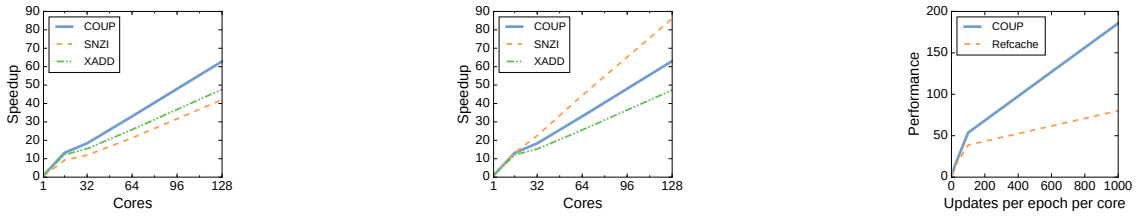
We augment zsim to emulate commutative-update instructions, which we encode with no-op instructions that are never emitted by the compiler. The simulator detects these instructions and simulates their functionality. We implement each commutative update using four μops: load-linked, execute (in one of the appropriate execution ports), store-conditional, and a store-load fence. Conventional atomic operations use exactly the same μop sequence in our implementation.

6.2 Case Study: Reference Counting

We use two microbenchmarks to compare COUP's performance on reference counting against the software techniques described in Sec. 4.2. The first microbenchmark models immediate-deallocation schemes, and we use it to compare against a conventional atomic-based implementation and SNZI [53]. The second microbenchmark models delayed-deallocation schemes, and we use it to compare against Refcache [44].

Processor chip	**Cores**	1–128 cores, 16 cores/processor chip, x86-64 ISA, 2.4 GHz, Nehalem-like OOO [83]
	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 with 40-cycle point-to-point links between each pair of processor and L4 chips
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 6.1: Configuration of the simulated SMP (Fig. 5-1).



(a) Immediate dealloc, low count

(b) Immediate dealloc, high count

(c) Delayed dealloc

Figure 6-1: Performance of COUP on reference counting microbenchmarks: immediate deallocation (a, b) and delayed deallocation (c).

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 uses commutative add instructions, and XADD uses atomic fetch-and-add instructions.

Fig. 6-1a and Fig. 6-1b show the results of these experiments. In the low-count variant (Fig. 6-1a), SNZI incurs high overhead when the counts drop to zero, so both COUP and XADD outperform SNZI (by 50% and 17%, respectively). COUP outperforms XADD because the MEUSI protocol enables multiple cores to update lines. By contrast, in the high-count variant (Fig. 6-1b), SNZI enjoys lower contention and outperforms COUP (by 35% at 128 cores).

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 [44].

Our COUP implementation works by updating counters with commutative add instructions and maintaining a bitmap with "modified" bits 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.

Fig. 6-1c 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

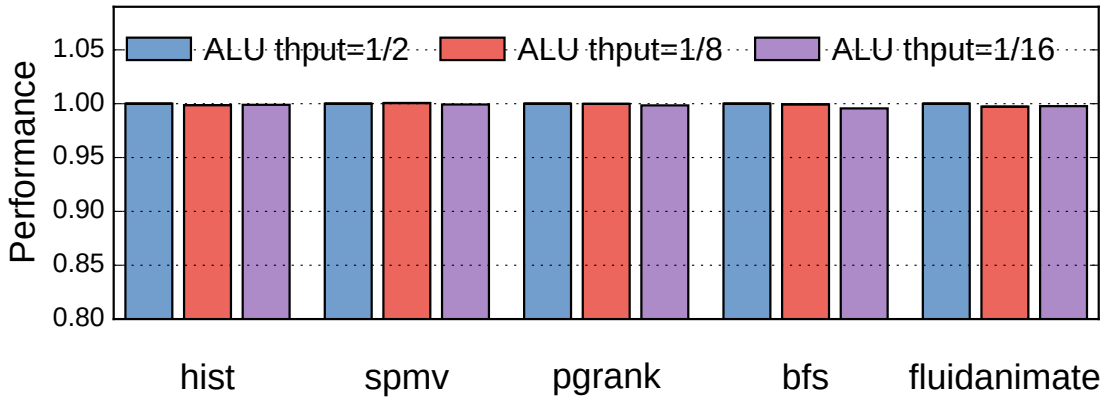


Figure 6-2: Sensitivity to ALU throughput at 128 cores.

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).

6.3 Sensitivity to Reduction Unit Throughput

Fig. 6-2 shows the performance of COUP on 128-core systems with three different ALU implementations: the default 256-bit ALU, which has a throughput of one cache line per 2 cycles; a non-data-parallel, 64-bit ALU, with a throughput of one line per 8 cycles; and an unpipelined 64-bit ALU with a throughput of one line per 16 cycles. Each set of bars show performance for a single application. Each bar shows performance relative to the default ALU; higher numbers are better. Fig. 6-2 shows that ALU thoughput has a minor impact on system performance. The maximum performance degradation incurred with simpler ALUs is 0.88% (on `bfs`).

Chapter 7

Multi-word Operations

In principle, in addition to the single-word operations described in chapter 5, COUP can support complex, multi-word operations . For example, many multicore workflows use work-stealing algorithms and thread-caching data structures to increase their performance. Such operations typically involve privatizing sets on each core, with periodic or lazy operations to maintain coherence. These workflows are a natural fit for COUP.

For this project, we implemented functional simulation of commutative set operations in COUP, though a detailed performance analysis is deferred to future work.

7.1 Set Representation

We support sets whose elements are 64-bit words. A set is represented by an ordinary, contiguous block of memory set up by software. Sets must be cache-aligned and their size must be a multiple of a single cache line so that reduction units at caches can correctly manage set membership.

Software interacts with sets through `push` and `pop` operations. These operations require a pointer which points to the descriptor for a set. Each descriptor contains the following fields:

- base (8 byte): the fixed memory address of the beginning of the set.

- limit (8 bytes): the fixed memory address of the end of the set.
- head (8 bytes): a pointer to the beginning of a region of the set.
- tail (8 bytes): a pointer to the end of a region of the set.
- flags (1 byte): A bit-field with internal state for the set, including whether it has been initialized yet or not.
- reserved (1 bytes): Application specific data reserved for software use.

Once software has initialized a descriptor and a contiguous block of memory for a set, it can update the set with two new instructions, which we describe next.

7.2 ISA Extensions for Sets

Sets provide two update-only operations, insert and delete. Because our sets support 64-bit words, we can implement these operations in one instruction each (`pushl` and `popl`, respectively). These instructions each take a pointer to a set descriptor. `pushl` additionally accepts a literal 64-bit word to add to the set.

An error code indicating the status of the operation is written to a register at the end of execution. For `popl` operations, the word extracted from the set is written to a separate register.

7.3 Proposed Implementation

These instructions could be implemented by placing specialized programmable controllers next to the reducers described in chapter 5. These controllers would support set operations that require communication between cores (for example, if a core's local set is empty and it needs to return an element that is cached elsewhere in reply to a `popl` operation).

In this scenario, cores would keep a local copy of the set descriptor cached in U state, which supports both `pushl` and `popl` instructions.

The details of how controllers would balance set membership between cores, as well as a properly detailed discussion of a micro-architectural implementation, are both deferred to future work.

7.4 Commutative Memory Allocation

Many popular memory allocators, like TCMalloc [98], pin software caches of free blocks in each core. Because memory allocators typically keep blocks in bins of fixed sizes (e.g. powers of two), these caches must be replicated across cores. If a cache becomes empty, cores must "borrow" free blocks of the same size from another core, or demote and split a larger block from its own core, increasing fragmentation.

COUP with sets provides automatic coherence for these per-core caches. In our memory allocator, we use commutative sets of fixed sizes (powers of two) to keep pointers to free blocks. Allocating and freeing memory correspond to popping and pushing pointers into the correspondingly-sized commutative set. This approach avoids the software overhead required to maintain per-core caches, benefiting from the same reduction in complexity as the applications described in chapter 4.

7.5 Future Work for Multi-word Operations

Using the set extensions described above, we plan to implement commutative work-stealing queues and a more optimized commutative memory allocator. With a more detailed study of the micro-architectural requirements to efficiently support this interface, we can evaluate the performance benefits under simulation.

Chapter 8

Conclusion

COUP is a technique that exploits commutativity to reduce the cost of updates in cache-coherent systems. It extends conventional coherence protocols to allow multiple caches to simultaneously hold update-only permissions to data. We have introduced an implementation of COUP that uses this support to accelerate single-word commutative updates. This implementation requires minor hardware additions and, in return, substantially improves the performance of update-heavy applications. Beyond this specific implementation, a key contribution of our work 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 parallel programming. Finally, COUP can apply to other contexts, including the multi-word set operations described in chapter 7, with limited programmability in the cache controller. We leave this and other applications of COUP to future work.

Bibliography

[1] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni, "Read-copy update," in *AUUG Conference Proceedings*, 2001.

[2] 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 *Proc. SOSP-24*, 2013.

[3] N. Narula, C. Cutler, E. Kohler, and R. Morris, "Phase reconciliation for contended in-memory transactions," in *Proc. OSDI-11*, 2014.

[4] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer: Designing an MIMD Shared Memory Parallel Computer," *IEEE Transactions on Computers*, vol. 100, no. 2, 1983.

[5] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proc. ASPLOS-VII*, 1996.

[6] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *Proc. HiPEAC*, 2010.

[7] L. Zhang, Z. Fang, and J. B. Carter, "Highly efficient synchronization based on active memory operations," in *Proc. IPDPS*, 2004.

[8] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. 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 *Proc. ISCA-36*, 2009.

[9] 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 *Proc. PACT-20*, 2011.

[10] 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 *Proc. MICRO-45*, 2012.

[11] G. Kurian, *Locality-aware Cache Hierarchy Management for Multicore Processors*. PhD thesis, Massachusetts Institute of Technology, 2014.

[12] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, 1996.

[13] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, 2011.

[14] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proceedings of the VLDB Endowment*, vol. 8, no. 3, 2014.

[15] R. E. Kessler and J. L. Schwarzmeier, "CRAY T3D: A new dimension for Cray Research," in *Proc. COMPCON*, 1993.

[16] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," in *Proc. ISCA-24*, 1997.

[17] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, 2011.

[18] J. H. Ahn, M. Erez, and W. J. Dally, "Scatter-add in data parallel architectures," in *Proc. HPCA-11*, 2005.

[19] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *Proc. ISCA-22*, 1995.

[20] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-level shared memory," in *Proc. ISCA-21*, 1994.

[21] D. Brooks and M. Martonosi, "Implementing application-specific cache-coherence protocols in configurable hardware," in *Proc. CANPC*, 1999.

[22] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas, "Protozoa: Adaptive granularity cache coherence," in *Proc. ISCA-40*, 2013.

[23] J. Zebchuk, E. Safi, and A. Moshovos, "A framework for coarse-grain optimizations in the on-chip memory hierarchy," in *Proc. MICRO-40*, 2007.

[24] S. Franey and M. Lipasti, "Accelerating atomic operations on GPGPUs," in *Proc. of the 7th IEEE/ACM International Symposium on Networks on Chip (NoCS)*, 2013.

[25] K. Russell and D. Detlefs, "Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing," in *Proc. OOPSLA*, 2006.

[26] I. J. Egielski, J. Huang, and E. Z. Zhang, "Massive atomics for massive parallelism on GPUs," in *Proceedings of the 2014 international symposium on Memory management*, 2014.

[27] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, 2010.

[28] W. Jung, J. Park, and J. Lee, "Versatile and scalable parallel histogram construction," in *Proc. PACT-23*, 2014.

[29] F. Ellen, Y. Lev, V. Luchangco, and M. Moir, "Snzi: Scalable nonzero indicators," in *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pp. 13–22, ACM, 2007.

[30] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, N. Zeldovich, *et al.*, "An analysis of linux scalability to many cores.," in *OSDI*, vol. 10, pp. 86–93, 2010.

[31] J. Corbet, "The search for fast, scalable counters," May 2010. http://lwn.net/Articles/170003/.

[32] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "Radixvm: Scalable address spaces for multithreaded applications," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 211–224, ACM, 2013.

[33] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *IEEE Computer*, vol. 29, no. 12, 1996.

[34] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *Proc. SC10*, 2010.

[35] J. H. Ahn, M. Erez, and W. J. Dally, "Scatter-add in data parallel architectures," in *Proc. HPCA-11*, 2005.

[36] A. R. Alameldeen and D. A. Wood, "IPC considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, 2006.

[37] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "Coordination avoidance in database systems," *Proc. VLDB*, vol. 8, no. 3, 2014.

[38] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. PACT-17*, 2008.

[39] 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 *Principles of Distributed Systems*, 2013.

[40] I. Calciu, J. E. Gottschlich, and M. Herlihy, "Using elimination and delegation to implement a scalable NUMA-friendly stack," in *Proc. HotPar*, 2013.

[41] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS directories: A scalable cache coherence scheme," in *Proc. ASPLOS-IV*, 1991.

[42] J. Chhugani, N. Satish, C. Kim, J. Sewall, and P. Dubey, "Fast and efficient graph traversal algorithm for cpus: Maximizing single-node efficiency," in *Proc. IPDPS*, 2012.

[43] 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 *Proc. PACT-20*, 2011.

[44] A. T. Clements, M. F. Kaashoek, and N. Zeldovich, "RadixVM: Scalable address spaces for multithreaded applications," in *Proc. EuroSys*, 2013.

[45] 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 *Proc. SOSP-24*, 2013.

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

[47] I. Culjak, D. Abram, T. Pribanic, H. Dzapo, and M. Cifrek, "A brief introduction to OpenCV," in *MIPRO, 2012 Proceedings of the 35th International Convention*, 2012.

[48] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel computer architecture: a hardware/software approach*. Gulf Professional Publishing, 1999.

[49] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software (TOMS)*, vol. 38, no. 1, 2011.

[50] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," in *Proc. OSDI-6*, 2004.

[51] J. Demmel and H. D. Nguyen, "Fast reproducible floating-point summation," in *Computer Arithmetic (ARITH), 2013 21st IEEE Symposium on*, 2013.

[52] A. Duran, J. Corbalán, and E. Ayguadé, "Evaluation of OpenMP task scheduling strategies," in *4th Intl. Workshop in OpenMP*, 2008.

[53] F. Ellen, Y. Lev, V. Luchangco, and M. Moir, "SNZI: Scalable nonzero indicators," in *Proc. PODC*, 2007.

[54] A. Fraknoi, "Images on the web for astronomy teaching: Image repositories," *Astronomy Education Review*, vol. 7, no. 1, 2008.

[55] S. Franey and M. Lipasti, "Accelerating atomic operations on GPGPUs," in *Proc. of the 7th IEEE/ACM International Symposium on Networks on Chip (NoCS)*, 2013.

[56] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, "Reducers and other Cilk++ hyperobjects," in *Proc. SPAA*, 2009.

[57] J. Goodman and H. Hum, "MESIF: A Two-Hop Cache Coherency Protocol for Point-to-Point Interconnects," 2009.

[58] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer: Designing an MIMD Shared Memory Parallel Computer," *IEEE Trans. Comput.*, vol. 100, no. 2, 1983.

[59] H. Hoffmann, D. Wentzlaff, and A. Agarwal, "Remote store programming," in *Proc. HiPEAC*, 2010.

[60] Intel, "TBB http://www.threadingbuildingblocks.org."

[61] N. P. Johnson, H. Kim, P. Prabhu, A. Zaks, and D. I. August, "Speculative separation for privatization and reductions," in *Proc. PLDI*, 2012.

[62] W. Jung, J. Park, and J. Lee, "Versatile and scalable parallel histogram construction," in *Proc. PACT-23*, 2014.

[63] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. 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 *Proc. ISCA-36*, 2009.

[64] R. E. Kessler and J. L. Schwarzmeier, "CRAY T3D: A new dimension for Cray Research," in *Proc. COMPCON*, 1993.

[65] F. B. Kjolstad and M. Snir, "Ghost cell pattern," in *Proc. of the 2010 Workshop on Parallel Programming Patterns*, 2010.

[66] K. C. Knowlton, "A fast storage allocator," *Comm. ACM*, no. 8, 1965.

[67] C. H. Koelbel, *The high performance Fortran handbook*. MIT Press, 1994.

[68] 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 *Proc. MICRO-45*, 2012.

[69] G. Kurian, "Locality-aware Cache Hierarchy Management for Multicore Processors," Ph.D. dissertation, Massachusetts Institute of Technology, 2014.

[70] A. Kyrola, G. E. Blelloch, and C. Guestrin, "GraphChi: Large-Scale Graph Computation on Just a PC." in *Proc. OSDI-10*, 2012.

[71] J. R. Larus, B. Richards, and G. Viswanathan, "LCM: Memory system support for parallel language implementation," in *Proc. ASPLOS-VI*, 1994.

[72] J. Laudon and D. Lenoski, "The SGI Origin: a ccNUMA highly scalable server," in *Proc. ISCA-24*, 1997.

[73] A. R. Lebeck and D. A. Wood, "Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors," in *Proc. ISCA-22*, 1995.

[74] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *Proc. SPAA*, 2010.

[75] M. M. Michael, "Hazard pointers: Safe memory reclamation for lock-free objects," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 15, no. 6, 2004.

[76] N. Narula, C. Cutler, E. Kohler, and R. Morris, "Phase reconciliation for contended in-memory transactions," in *Proc. OSDI-11*, 2014.

[77] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web." 1999.

[78] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," in *Proc. ISCA-11*, 1984.

[79] L. Rauchwerger and D. Padua, "The privatizing doall test: A run-time technique for doall loop identification and array privatization," in *Proc. ICS'94*, 1994.

[80] L. Rauchwerger and D. A. Padua, "The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization," *IEEE Trans. on Parallel and Distributed Systems*, vol. 10, no. 2, 1999.

[81] S. K. Reinhardt, J. R. Larus, and D. A. Wood, "Tempest and Typhoon: User-level shared memory," in *Proc. ISCA-21*, 1994.

[82] D. Sanchez and C. Kozyrakis, "SCD: A scalable coherence directory with flexible sharer set encoding," in *Proc. HPCA-18*, 2012.

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

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

[85] N. Satish, N. Sundaram, M. M. A. 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 *Proc. SIGMOD*, 2014.

[86] S. L. Scott, "Synchronization and communication in the T3E multiprocessor," in *Proc. ASPLOS-VII*, 1996.

[87] D. J. Sorin, M. D. Hill, and D. A. Wood, "A primer on memory consistency and cache coherence," *Synthesis Lectures on Computer Architecture*, vol. 6, no. 3, 2011.

[88] 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.

[89] T. Von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser, "Active messages: a mechanism for integrated communication and computation," in *Proc. ISCA-19*, 1992.

[90] 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 *Proc. ISSCC*, 2015.

[91] C. M. Wittenbrink, E. Kilgariff, and A. Prabhu, "Fermi GF100 GPU architecture," *IEEE Micro*, vol. 31, no. 2, 2011.

[92] 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 *Proc. PACT-17*, 2008.

[93] H. Yu and L. Rauchwerger, "Adaptive reduction parallelization techniques," in *Proc. ICS'00*, 2000.

[94] J. Zebchuk, M. K. Qureshi, V. Srinivasan, and A. Moshovos, "A tagless coherence directory," in *Proc. MICRO-42*, 2009.

[95] J. Zebchuk, E. Safi, and A. Moshovos, "A framework for coarse-grain optimizations in the on-chip memory hierarchy," in *Proc. MICRO-40*, 2007.

[96] L. Zhang, Z. Fang, and J. B. Carter, "Highly efficient synchronization based on active memory operations," in *Proc. IPDPS*, 2004.

[97] H. Zhao, A. Shriraman, S. Kumar, and S. Dwarkadas, "Protozoa: Adaptive granularity cache coherence," in *Proc. ISCA-40*, 2013.

[98] Ghemawat, Sanjay and Menage, Paul, "Tcmalloc: Thread-caching malloc," at goog-perftools.sourceforge.net/doc/tcmalloc.html, 2009.