

Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols

Dana Vantrease
Univ of Wisconsin - Madison
Madison, WI
Email: danav@cs.wisc.edu

Mikko H. Lipasti
Univ of Wisconsin - Madison
Madison, WI
Email: mikko@engr.wisc.edu

Nathan Binkert
HP Labs
Palo Alto, CA
Email: binkert@hp.com

Abstract—This paper advocates Atomic Coherence, a framework that simplifies cache coherence protocol specification, design, and verification by decoupling races from the protocol’s operation. Atomic Coherence requires conflicting coherence requests to the same addresses be serialized with a mutex before they are issued. Once issued, requests follow a predictable race-free path. Because requests are guaranteed not to race, coherence protocols are simpler and protocol extensions are straightforward.

Our implementation of Atomic Coherence uses optical mutexes because optics provides very low latency. We begin with a state-of-the-art non-atomic MOEFSI protocol and demonstrate that an atomic implementation is much simpler while imposing less than a 2% performance penalty. We then show how, in the absence of races, it is easy to add support for speculative coherence and improve performance by up to 70%. Similar performance gains may be possible in a non-atomic protocol, but not without considerable effort in race management.

I. INTRODUCTION

Cache coherence protocols manage reads and writes to shared memory locations. Given a memory location, a protocol should deliver read values that can be interleaved into a total order of writes to that same location.

The literature usually represents cache coherence protocols as state machines with events that cause atomic transitions between stable states (e.g. M , S , I) [1]. In early bus-based machines, these *stable transitions* could be accomplished atomically, since the bus, once allocated, was held until the transition completed and prevented the initiation of any other concurrent transitions. Such blocking shared buses made it relatively easy to guarantee coherence, since arbitration for the shared resource implicitly specified a total order for all memory references. The buses atomically serialized all racing requests, enabling a trivial construction of a

coherent system-wide interleaving of reads and writes to a common address.

Unfortunately, performance concerns quickly forced designers to abandon blocking shared buses in favor of nonblocking buses and other high performance interconnects [2]. Such interconnects provide much higher utilization by separating a request from its associated responses, allowing the bus to deliver other commands in the interim. The price of this optimization is that transitions between stable states no longer occur atomically. Instead, transitions must be decomposed into a series of *split transitions*. Split transitions correspond to the completion of various steps in the end-to-end state transition. This is an unavoidable result of exposing greater concurrency in the processor-memory interconnect.

Unfortunately, absent arbitration for a single shared resource, split transaction protocols are susceptible to races between conflicting requests to the same addresses. Coherence protocols must provide a means for detecting and resolving these races. If the race is detected after two or more nodes in the system have initiated conflicting coherence state transitions, the protocol must include additional *race transitions* to correctly resolve the situation.

Figure 1 shows the contribution of the above three transitions (stable, split, and race) to a simple MSI directory protocol and a more complex MOESTI bus protocol [3]. Note that race transitions make up the largest category. Next we will discuss the impact these race transitions have on protocol design and verification.

A. Impact of Race Transitions

Although race transitions are make up the plurality of finite state machine transitions, they only make up a small fraction of all observed transitions in our scientific and commercial workloads. Because of their relative infrequency, race transitions have little impact on performance, yet they impose significant design complexity and verification challenges.

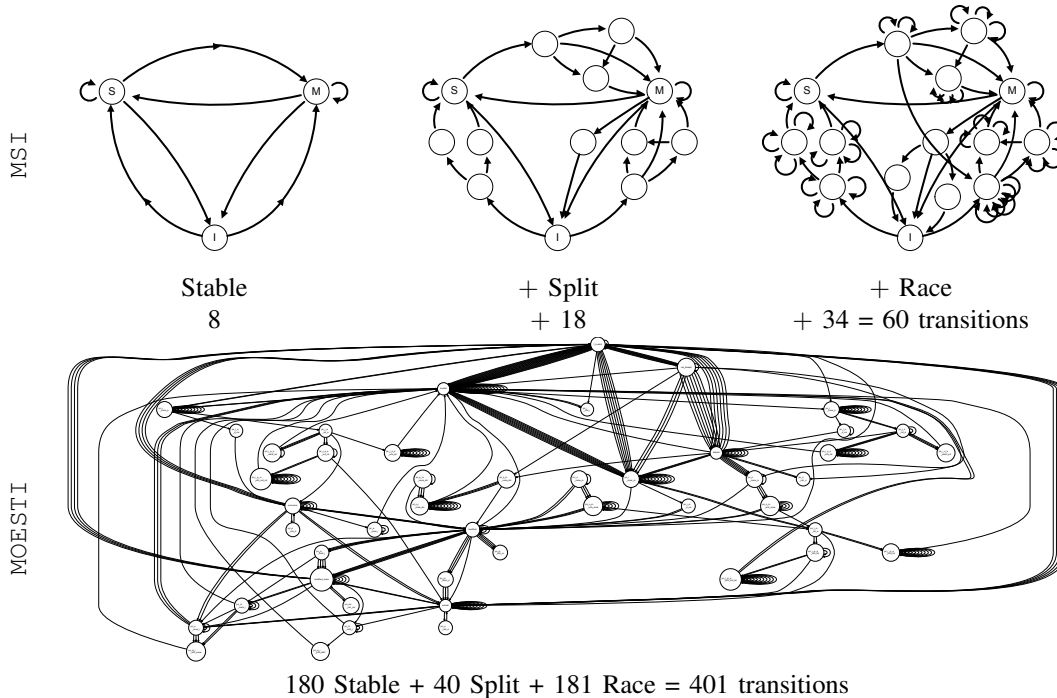


Fig. 1: L2 Transitions for Two Protocols.

Designing for races is hard, because a race event is an “unexpected interruption” to an in-flight request. Designers must consider all races and determine which races are possible (or impossible). The possible transition and action pairs require specification in the finite state machine. At a minimum, the designer must prune down the space of all event-state combinations. If races are allowed to race, the space can compound indefinitely.

Verifying racy protocols is also hard, because race-induced design bugs may be masked away during most executions, but may reveal themselves at inopportune times. Random traffic generators may stress, but not exhaust, potential race combinations [4]. Formal verification is more robust, but the space explosion caused by the races can consume large amounts of hardware resources and time. Formal verification is also prone to human-error during protocol-to-formal-language translation [5].

Ultimately, despite significant efforts to improve protocol verification [4], [5], [6], coherence bugs still make it into shipping products [7]. Recently, work by Zhang, Lebeck, and Sorin focused on easing the burden on verifiers by incorporating verifiability into the coherence protocol design [8]. Instead of focusing on verification, we turn our attention to the protocols themselves, believing the best way to simplify verification is to simplify the objects being verified.

B. A Return to Atomic Protocols

In this paper, we advocate a return to atomic protocols. Since a return to atomic interconnects is unlikely, we decouple atomicity from the interconnect and require that a fine-grained mutex for a block must be acquired before any coherence request on that block may be performed. By removing the races from the protocol, we substantially simplify the protocol.


Before a coherence request issues, the request must obtain exclusive access to the memory block it is requesting. When the request is complete, and the address is system-wide coherent, the mutex may be released. Conflicting accesses are serialized before they begin, just as in blocking shared-bus systems. However, mutual exclusion is accomplished via fine-grained mutexes, rather than through arbitrating for a single shared bus, hence exposing abundant concurrency across independent requests, while completely eliminating the need for detecting and resolving races within the coherence protocol proper.

Because acquiring a mutex is on the critical path, we investigate a low-latency, optical substrate inspired by the optical arbitration protocol used by Corona [17].

This paper shows that an atomic implementation of a state-of-the-art MOEFSI protocol is much simpler and easier to verify while imposing less than 2% performance penalty. We also add features to this protocol

Eager

Lazy



Before Coherence	Front-End of Coherence	During Coherence	Back-End of Coherence/Time-Out
Atomic Coherence	Wildfire [9] , Gigaplane [10]	SGI-Origin 2000 [11]	Token Coherence [12]
Blocking Bus [13]	Ring-Order [14]	SCI [15]	Token Tenure [16]

TABLE I: Eager/Lazy Race Detection. Where race detection takes place along a racing-request’s path.

to support `F` state migration (`ShiftF`), minimizing off-chip memory references, and an aggressive push protocol (`PushS`) to prefetch on-chip cache-to-cache transfers. These extensions would introduce significant complexity in a non-atomic framework and would likely be deemed unattractive, but in our Atomic Coherence framework they are realized easily, by a single graduate student in a matter of hours, while delivering significant performance improvements (up to 7% for `ShiftF` and 70% for `PushS`).

Section II and Section III define Atomic Coherence and discuss performance implications, respectively. Section IV describes our photonic implementation of Atomic Coherence. Section V and Section VI describe our experimental methodology and results, while Section VII concludes the paper.

II. ATOMIC COHERENCE

A. Definition and Semantics

We define Atomic Coherence in the following way:

We say a system implements *Atomic Coherence* if, for any block, every transition from one stable coherence state to another stable coherence state occurs atomically with respect to all other stable transitions for that block.

Some protocols consider a coherence state to be stable once its control permissions have been established [9], [18]. Others additionally require data-transfer completion [11]. We explore two atomicity policies, reflecting both views:

- `Atomic CResp` Policy: Guarantee atomicity until all control responses received and processed.
- `Atomic DResp` Policy: Guarantee atomicity until all control *and* data responses received and processed.

The only prior coherence protocols that we are aware of that satisfy the `Atomic DResp` flavor of Atomic Coherence are ones that rely on a shared blocking bus [19], [13], where atomicity is guaranteed by successfully arbitrating for and holding the bus for the entire transaction. Two-transaction split busses, where control is completed in the first bus transaction and data in the second, satisfy `Atomic CResp` requirements.

All other protocols are non-atomic, but some are more atomic than others. Blocking directories, like Wildfire and Safe Points [9], [20], are the closest to Atomic Coherence because races are detected early-on at the directory (i.e. eagerly), shortly after the requester has left its stable state. Meanwhile, Token Coherence, which uses tokens to resolve permissions, is the furthest because it detects races with time-outs (i.e. lazily) [12]. In general, the later a race has been detected, the more system state has been affected, and the harder it is to resolve (undo) the race. Table I categorizes race detection for a variety of protocols.

As alluded to above, blocking directories are similar to and share many benefits with atomic protocols. However, blocking directories are not strictly atomic, because conflicting requests are allowed to leave requesting nodes and race to the directory. Blocking directories will also have much higher resource requirements, because each directory must provide worst-case buffering to absorb conflicting requests from each node in the system (or fall back on NACKs, which can lead to congestion, thrashing, and poor performance). Finally, blocking directories restrict implementations to protocols that always visit the directory first before performing any other coherence operations. This precludes not only snoopy coherence implementations, but also speculative optimizations that directly probe some subset of caches in parallel with or prior to reaching the directory (e.g. multicast snooping [21]). Atomic Coherence is a general framework, applying to snoopy and directory systems alike.

B. Atomic Coherence Substrates

From the definition of Atomic Coherence, we can define atomic protocols as being made up of two independent substrates:

- The *Atomic substrate* enforces the invariant that only one stable-to-stable transaction to a memory block may be in progress at a time (mutual exclusion).
- The *Coherence substrate* performs the coherence, enforcing contracts for block permissions and sometimes data.

The Atomic substrate is dedicated to resolving races, so the Coherence substrate does not have to. The Co-

herence substrate is left with only stable transitions and split transitions. Once a request has been granted issuing-rights in the Atomic substrate, it may enter the Coherence substrate where it will only receive events that will further it towards its destination stable state (e.g. responses, acknowledgements, data). Guaranteeing forward progress eliminates the possibility of pathological livelock and starvation scenarios in the protocol.

Without races, the Coherence substrate is easier to design and verify. The Atomic substrate also needs to be designed and verified, but its job, to detect and resolve races, is simple and so it should also be relatively easy to verify. Finally, because the Atomic substrate is entirely decoupled from the Coherence substrate, verification of both substrates may proceed in parallel, shortening end-to-end verification time.

C. Atomic Coherence with Mutexes

Mutexes are a natural way to support mutual exclusion in the Atomic substrate. According to the semantics outlined above, the block’s coherence state may not be altered until the mutex has been obtained. If, while pending in the Atomic substrate, a mutex acquisition is interrupted by a conflicting coherence request, the mutex must have been already claimed by another node and the acquisition must be retried. In this way, mutexes provide strong guarantees on message arrival order, even on top of a general unordered interconnect.

A miss that has seized its mutex may begin its transaction to the next stable state, leaving its node to perform coherence activity in the Coherence substrate. The mutex is released only after it has reached its next stable state, implying that all relevant coherence activity has quiesced and guaranteeing that there will be no coherence interference with subsequent conflicting requests.

For safety, we only allow a cache to make one coherence request per mutex acquisition before passing the mutex on. Still, all nodes should have fair access to the mutex. We leave mutex fairness techniques to future work, but believe the Atomic substrate can provide fair service by adapting techniques from prior work [22].

In Section VI we evaluate two policies for releasing the mutex. The `Atomic CResp` policy holds the mutex through permission acquisition, and the `Atomic DResp` further delays release of the mutex until the data itself has arrived at the requester. With `Atomic CResp`, a node that is otherwise coherent but is still waiting for data (typically from a long-latency DRAM fetch) may release the mutex but must be prepared to queue a subsequent request to the same block. The `Atomic DResp` policy removes even this requirement, holding the mutex until data arrival at the expense of a

considerable increase in mutex hold time and additional pressure on finite mutex resources.

III. PERFORMANCE IMPLICATIONS

Atomic Coherence has performance pitfalls and opportunities, which we discuss below.

A. Performance Pitfalls

Atomic Coherence can negatively affect performance in two ways. First, the act of acquiring the mutex before issuing a request is latency critical because it adds to the request’s critical path. Second, Atomic Coherence may serialize coherence requests when it would have been perfectly coherent to service them in parallel (e.g. parallel reads).

1) *Mutex Constraints*: Because atomicity is on the critical path, a free mutex should be seized as quickly as possible. Ideally, we would like to have as many mutexes as we have blocks and be able to seize these mutexes instantaneously, but physical constraints prohibit this. Hashing or associating a mutex with a super-block region of memory [23] can help with a limited mutex pool. For latency, techniques include: spatially locating a mutex near its likely requester, prefetching a mutex, or quickly circulating a mutex by requesters. In this work, we employ hashing and circulating the mutexes optically. Optics allows an available mutex to make a full pass by every potential requester in about 800 picoseconds (4 cycles at 5 GHz).

2) *Serialization*: Many protocols allow parallel reads to proceed simultaneously barring any intervening write. Conceptually, this seems safe, as long as the reads are free of side effects. However, coherence read requests often do have side effects, such as updates to sharing lists or LRU state. As long as these metadata updates can be correctly handled, it would be safe to allow concurrent read requests in the coherence subsystem. As described, Atomic Coherence serializes these reads, which may negatively affect performance by inhibiting concurrency.

Our results show that read-serialization effects are usually negligible. However, when read-serialization is non-negligible, it can be mitigated with a straightforward push protocol optimization we call `PushS`, which we discuss next.

B. Performance Opportunities

We can counteract the negative attributes of Atomic Coherence by adding low-cost speculation features to the protocol. Our speculation features are based on the observation that while a processor holds a mutex, it may perform coherence in any way it sees fit, as long as it leaves the system in a coherent state when it releases the mutex. During a mutex holding, we allow potential future sharers to be “pushed” a block (`PushS`)

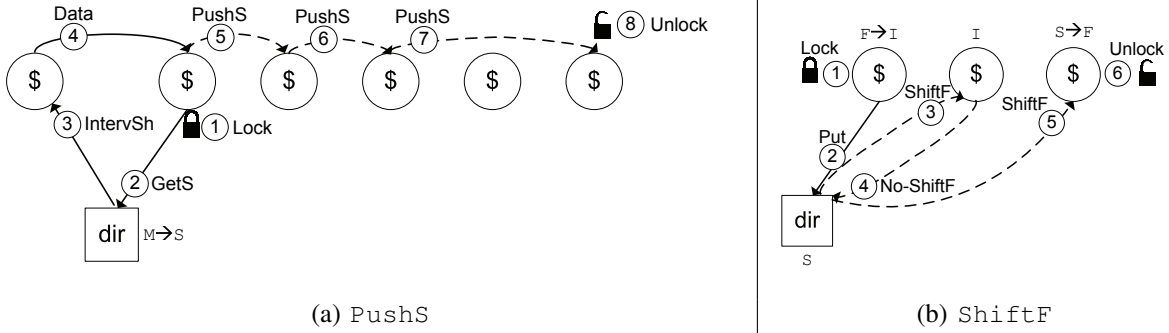


Fig. 2: PushS/ShiftF Overview. (a) **PushS**: When a block goes to Shared at the directory, the old sharing list is used to push the block into caches (pushes shown with dotted lines). (b) **ShiftF**: When the F block is evicted, the directory serially sends ShiftF messages until it finds an S block that can be upgraded to F.

Note for both figures: The mutex is seized by the requester but released by a non-requester.

and owner permissions to be manipulated (ShiftF). We believe both of these protocol features are novel contributions.

PushS and ShiftF are coherence optimizations, much like any other coherence optimization [24], [25], [26]. In theory, they could be implemented in any atomic or non-atomic system, but non-atomic implementations would be non-trivial, because the optimizations require multiple steps and potentially many serial visits to multiple nodes. This would make non-atomic implementations very vulnerable to races and latent protocol bugs which beget additional split and race transitions in the protocol, contributing to state-space explosion and creating verification headaches. In Atomic Coherence, the absence of races makes them trivial to implement, and as we show in Section VI, provide compelling performance benefits.

Now, the optimizations:

1) *PushS*: Our PushS protocol pushes data into caches so to avoid future misses.

It works as follows (shown in Figure 2(a)): when a block transitions to exclusive at the directory, we do not erase the sharer list. Instead, we hold the sharing list in place for future reference. When the block is later read and leaves the Exclusive state, the old sharing list becomes our prediction of new sharers as well as the new sharing list. The data is sent in a message, visiting each predicted cache in series, similar in spirit to Piranha’s Cruise Missile Invalidate [27]. At each visit, the data is prefetched (or “pushed”) into each processors’ cache for later use. A processor that has a read outstanding to the block (and is waiting for the mutex) will have its read immediately satisfied and the mutex request will be dropped. The last cache visited by the missile releases the mutex. For performance reasons, our implementation eagerly passes the missile. A separate missile trails the initial missile, waiting for the L2 to process the push

before continuing. Only when the last node in the list has received *both* the missile and the trailing missile, may the mutex be released. Note that different nodes seize and release the mutex: the initiator of the miss seizes the mutex while the last missile site releases the mutex.

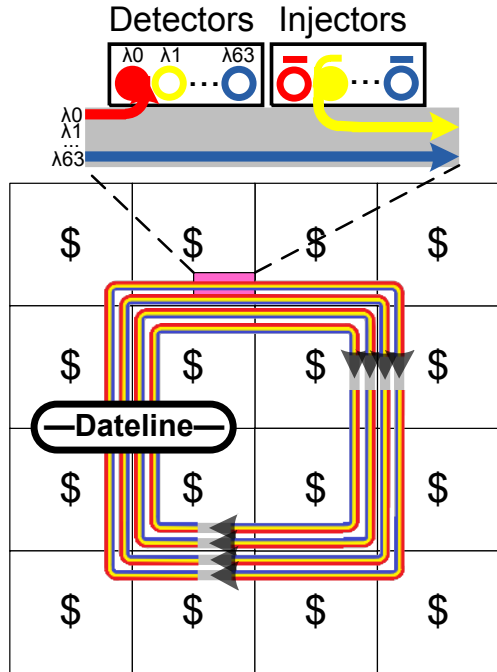
2) *ShiftF*: Our ShiftF protocol tries to keep a shared block in a state that sources to other on-chip caches (e.g. O and F¹). A block that is in one of these sourcing states saves other sharers from supplying redundant responses and elides a memory reference, saving both time and energy. Of course, all sharers may respond to a new sharing miss, but this can potentially lead to a flood of extraneous responses.

Our ShiftF protocol works as follows (shown in Figure 2(b)): when an F or O block is evicted, the directory tries to find another sharer to take on the F state (if O, the data is cleaned to F with a memory writeback). While the mutex is being held, the directory goes through the sharing list, serially making attempts to assign F and sending ShiftF messages to sharers. The sharer may be in the S or I state:

- If the sharer is in S, the block transitions to F, its LRU state updated, and the mutex released.
- If the sharer is in I (silently evicted), the sharer informs the directory it is no longer a sharer; the directory removes the sharer from the sharing list, corrects its F assignment, and tries sending another ShiftF to the next sharer on the list.

If all ShiftF’s encounter I blocks, the block is no longer cached; the directory entry transitions to invalid and releases the mutex. Note that shifting has many benefits: it keeps an on-chip sourcer, updates the directory’s sharer information (which may be stale due to silent evictions), and cleans the data.

¹A block in the F, or Forward state, is clean with respect to memory and is responsible for supplying data to on-chip misses [28].



$$4 \text{ wg} \times \frac{64 \lambda}{\text{wg}} \times \frac{4 \text{ mutexes}}{\lambda} = 1024 \text{ mutexes}$$

Fig. 3: Optical Atomic Substrate. ~ 2 mutexes fit in circulation on a wavelength (and the dateline buffers ~ 2 more, bringing the total to 4 mutexes per wavelength per waveguide). A waveguide (wg) may be accessed by any of the 16 L2 caches, and each waveguide multiplexes 64 wavelengths (λ), for a total of 1024 mutexes in the system. At the top of the figure, we show one cache’s detectors and injectors. The cache is detecting one mutex on λ_0 and injecting another on λ_1 . λ_2 is not of interest to the cache and passes unperturbed.

IV. IMPLEMENTATION

On-chip silicon photonics is a promising up-and-coming technology that allows for fast communication with little power, as evidenced by recent production of silicon-based transceivers in the telecom industry [29]. Pulses of light circulate around a ring-shaped interconnect made of silicon waveguides. The presence of a light pulse represents an available mutex while its absence represents an unavailable mutex. The nodes are equipped with ring resonators, which they periodically bring into resonance when the mutex they require is passing by. An on ring resonator diverts a specific wavelength of light from the waveguide, effectively atomically performing a destructive read on the mutex (i.e. seizing the mutex); downstream nodes will not divert the mutex. If the ring resonator detects light, it has won the mutex. An off ring resonator has a negligible effect on the light, and the light passes by at the speed of light.

Requesters must know when to turn on their resonators

to divert a mutex. Many mutexes may be in flight simultaneously, occupying different spatial “slots” on different wavelengths [30]. We assume the assignment of a mutex to a slot is global knowledge. For instance, mutex m might be found by cache c at slot i on wavelength j in waveguide k . Slots are separated by clock edges, and nodes turn on and off their ring resonators to align with the clock’s edges. If the ring resonator does not divert the mutex, it must try again on the next revolution of the mutex. The retry is effectively a local-NACK and thus consumes no interconnect bandwidth.

The nodes obey an optical clock signal that travels on a parallel broadcast-powered wavelength. This wavelength is in phase with the mutex transmission, allowing nodes to divert the entire pulse associated with a mutex. The mutex transmission can come out of phase with the clock if the ring-shaped interconnect is not an integral of clock edges in length. To remedy this, we use a global dateline to re-time the in-flight mutexes through a Optical-to-Electrical-to-Optical (OEO) repeat [31]. The dateline’s main job is retiming, but it also doubles as a means to refresh the mutex’s power.

The resource cost of having an optical mutex for every cached address, much less every memory address, would be prohibitive. We reduce this number by hashing several addresses to a single mutex. Furthermore, we divide the mutex-space such that each directory “owns” an equally sized mutex subset, believing that accesses should be spread equally across all directories. Thus, a request maps its addresses to a directory (a subset of mutexes) and then hashes its address (excluding the bits for the offset and directory) into a mutex mapping. We use a simple hash function that operates on the lower bits of a block address exclusive-or’d with some of the higher bits. We found this hash function exploits the high-entropy of the lower bits and performs competitively with a more sophisticated H3 hash function [32].

When the request completes and it comes time to release the mutex, the node must return the mutex to its assigned slot in the waveguide. It does so by injecting light at the precise moment the slot is passing by. A ring resonator with a power source serves as the injector. After injection, the mutex passes without interruption (except for the dateline) to the next requester of the mutex.

We draw inspiration from Corona’s arbitration system, which also uses optical pulses to maintain mutual exclusion [17]. Figure 3 shows the optical layout and configuration of our Atomic substrate.

Component	Count	Details
Nodes		
Cores	128	single-threaded, in-order, 5 GHz
L1Is/L1Ds	64	32 KB, 4 way, 2 cycle
L2s	16	2 MB, 16 way, 10 cycle, 64 byte block
Directory Caches	16	2048 sets, 24 way, 13 cycle
Memory Controllers	16	
Interconnects		
On-chip Data Waveguides	64	Optical, 2 cycles/64 byte msg, < 3 cycle latency, 4 wg/channel
Off-chip Memory Waveguides	32	Optical, 4 cycles/64 byte msg, 90 ns latency, 2 wg/channel
Atomic Coherency Substrate		
O/E E/O latency	1 cycle	
Mutexes Per Wavelength	4	
Wavelengths Per Waveguide	64	
Mutex Waveguides	4	
Mutexes	1024	
Cycles per Mutex Revolution	4	

TABLE II: System Configuration

V. METHODOLOGY

A. Experimental Setup

We evaluate Atomic Coherence on a chip with 128 single-threaded cores. The chip has 16 L2s connected by on- and off-chip optical interconnects, similar to a scaled down version of the 1024-threaded Corona architecture [17]. The on-chip interconnect allows any-to-any communication with point-to-point ordering between nodes (note that our coherence protocols do not rely on these ordering properties).

Our data interconnect differs from Corona in a few key ways. Corona uses a snake-shaped interconnect to connect 64 nodes (256 quad-threaded cores); our interconnect uses a ring-shaped interconnect to connect 16 nodes (128 single-threaded cores). The difference in interconnect shape translates to a shorter light-path, allowing light to complete a revolution in less than 3 cycles at 5 GHz, compared to Corona’s 8 cycles. Like Corona, the data channels are 4 waveguides wide, however there are fewer channels (16 vs 64) and they are clocked at half the rate (on the leading clock edge instead of both clock edges). In total, our data interconnect consumes 64 waveguides. Our Atomic substrate interconnect consumes 4 waveguides.

Our atomic baseline is a directory-invalidate MOEFISI protocol. We built this protocol by extending a multicore appropriate version of SGI-Origin 2000 MESI protocol to support the O and F states, which allow for on-chip sourcing of dirty-shared and clean-shared data, respectively. Like SGI-Origin, we allow silent evictions of S blocks at the L2. Our directory protocol uses an on-chip directory cache with a backing DRAM directory to track the 32 MB of L2 cache. We support sequential consistency.

Ring resonators can convert a mutex from Optical-

to-Electrical (OE) or Electrical-to-Optical (EO) in one cycle (200 ps). This estimate is in line with other 2017 targets: it is less aggressive than Phastlane (20 ps) but more aggressive than FlexiShare (400 ps) ([33], [34]).

We achieve 1024 mutexes by spatial packing, buffering at the dateline, and wavelength multiplexing. Each mutex has a binary value (available or unavailable), so the 1024 mutex system has 128 bytes of state.

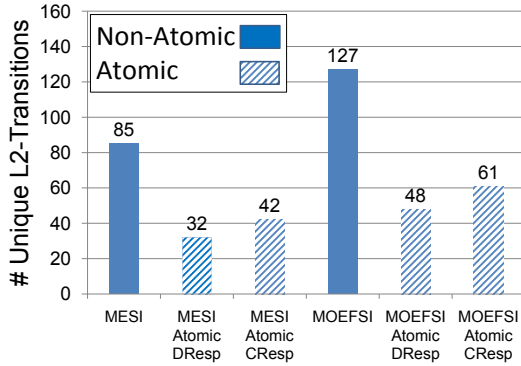
Details of our configuration can be found in Table II.

B. Experiments and Measurements

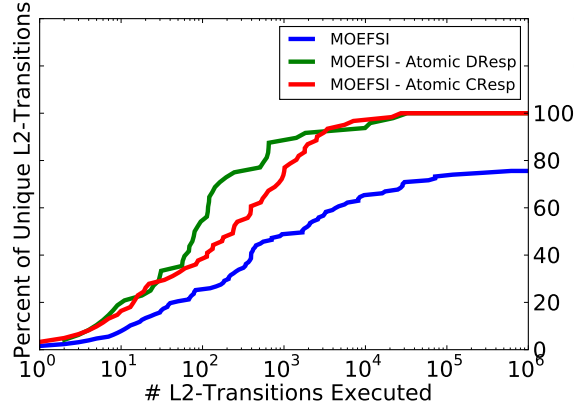
We apply our Atomic-to-Data-Response (Atomic DResp) and Atomic-to-Coherence-Response (Atomic CResp) to our baseline MOEFISI protocol and demonstrate the impact of mutex acquisition and request serialization on bottom line performance. Next, we optimize the performance of our protocols by adding support for shifting the F state (ShiftF) and pushing the S state (PushS).

Our results measure performance and complexity. For performance, we tested our protocols with a memory-intensive subset of workloads from the SPLASH-2 suite (Barnes and Ocean) [35] and three commercial workloads from the Wisconsin Commercial Suite (OLTP, Apache, and SPECjbb) [36]. We also added a micro-benchmark that models a 3-level feed-forward neural network, as a “best-case” scenario for PushS. A neural network should benefit from PushS, because its sharing behavior alternates between single-writer and many-reader.

Each experiment was run in server-consolidation mode with eight 16-threaded instances, except for the neural network, which was run as a single 128 thread instance. Each instance was offset from the next, and pages were dynamically re-mapped on demand. Threads were assigned to cores in a semi-random fashion, to both



(a) Static



(b) Dynamic

Fig. 4: (a) Static. L2 Transition Counts **(b) Dynamic.** Cumulative coverage of L2 Transitions using a directed random tester (Note: log-scale on x-axis)

simulate real-world scheduling and to stress coherence. Inter-core sharing occurs at the L1 (2-cores) and the L2 (8-cores).

All simulations were trace-driven, except the neural network, which was driven by Pin [37]. Traces contained L1 misses retrieved from runs in the GEMS simulator [38]. The simulator faithfully modeled the miss at the L2 and all levels below the L2, dynamically generating L2 misses and the appropriate coherence messages on-demand. Caches were warmed with a fixed number of trace entries and then run for a fixed number of trace entries, the equivalent of about 50 million and 500 million instructions respectively.

VI. RESULTS

A. Complexity

The primary advantage of Atomic Coherence is a reduction in protocol complexity. Complexity is difficult to quantify using a single metric, so this section characterizes our protocols both statically, in terms of the overall number of unique transitions in each protocol, as well as dynamically, by reporting coverage achieved by a random protocol tester.

1) *Static Complexity Analysis:* Figure 4(a) shows the percent of unique transitions for two non-atomic protocols (MESI, MOEFISI) and four atomic protocols ([MESI, MOEFISI] + [DResp, CResp]). We only count the transitions triggered at the L2, because this is where the protocols’ complexity lies.

The two non-atomic protocols contain 85 and 127 unique transitions, respectively. The increase in transitions conveys the additional complexity required for the new \circ and F states. The figure also shows that, if we enforce coherence atomicity through to the completion of the data (Atomic DResp) or the coherence (Atomic

CResp), our transition counts drop by at least 50%. The dropped transitions existed solely to handle protocol races.

Atomic CResp has slightly more transitions than Atomic DResp, because Atomic CResp must deal with a small vulnerability window. The vulnerability exists when coherence request A releases the mutex before it has acquired its data, and the next coherence request (B) acquires the mutex. A will recognize B ’s request as a later ordered event and will delay the request until its data has arrived. When the data arrives, A satisfies its own request and then forwards the data to B . This case can only occur while waiting for data, involves at most a single subsequent request, and adds only a slight amount of complexity to the protocol. It also significantly reduces Atomic CResp’s mutex hold time for misses to main memory.

2) *Dynamic Complexity Analysis:* To further support our simplicity assertion, we collected coverage results using a directed random tester similar to the one provided with GEMS. The tester generates pseudo-random memory references at each node in a targeted mix of reads and writes that are designed to trigger both independent and conflicting memory references in the coherence subsystem. The tester also randomly delays responses on the interconnect in order to synthetically expose transient states to racing requests.

The purpose of the tester is to excite as many unique transitions as quickly as possible in order to quickly uncover protocol bugs and to allow rapid testing and turnaround for protocol fixes. We are not claiming that random testing alone is sufficient for validation of a real design, but it is a very useful and widely-deployed tool for flushing out design errors, particularly early in the design process. An efficient random tester that provides

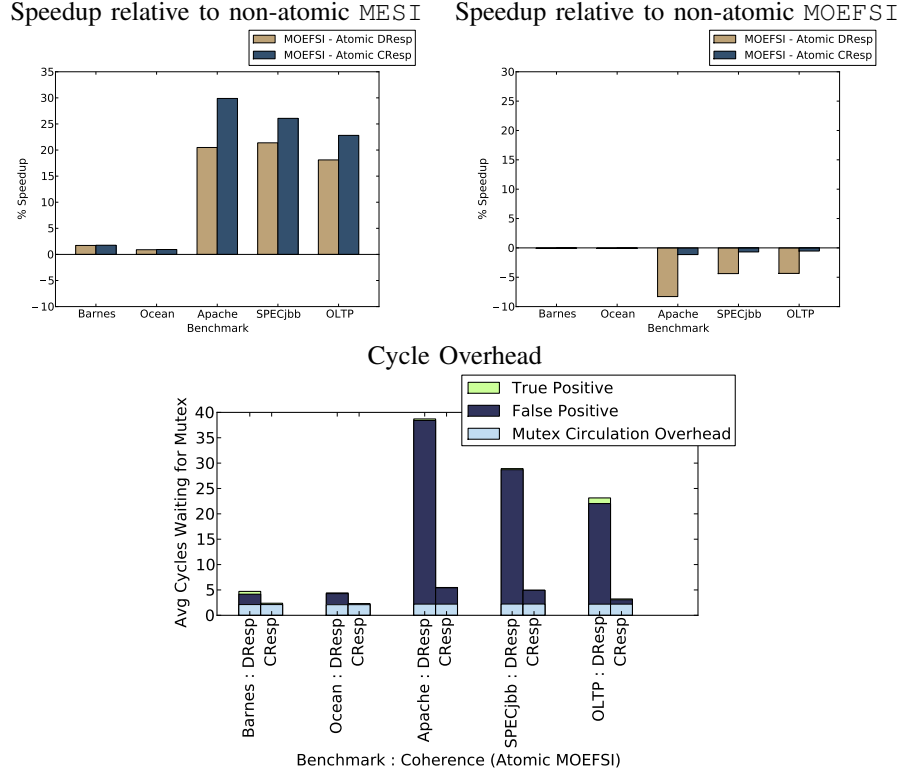


Fig. 5: Performance and Overhead for Atomic Coherence. Configurations use 1024 mutexes across 4 wg.

fast and early coverage of as many unique transitions as possible will substantially improve the productivity of the protocol designer.

The results in Figure 4(b) clearly illustrate the complexity advantages of our atomic protocols. Well within the first million transitions, Atomic DResp and CResp have reached 100% coverage of all unique transitions, while the non-atomic MOEFISI protocol begins leveling off asymptotically at 75% coverage. This result suggests two conclusions about the atomic protocols: (1) turnaround time to uncover, fix, and test design bugs is reduced substantially, since all protocol transitions are exercised early and often; and (2) the rare and likely problematic corner cases lurking in the remaining unexercised 25% of transitions in the non-atomic protocol are completely eliminated, reducing the unpleasant consequences of encountering them late in the design cycle.

Of course, tweaking the random tester to synthetically force decreasingly likely protocol races to occur sooner could probably improve the cumulative distribution for non-atomic protocols, but this activity requires designer effort as well, since such tweaks will likely have to be specific to each protocol and interconnect topology. In contrast, our very simple, generic, first attempt at a random tester (~200 lines of C code) produces excellent

coverage in a very timely manner, as long as it is exercising one of our atomic protocols.

Our experiments with the random tester also produced three interesting anecdotal observations: first, the tester uncovered multiple latent, obscure bugs in our non-atomic MOEFISI protocol; second, the tester failed to uncover any bugs in our atomic protocols, which were correct in their first iteration; third, the tester uncovered opportunities for simplifying our atomic protocols, since it failed to exercise transitions that, upon closer examination, we found to be impossible and subsequently removed from the protocol. These observations lend credence to our claims that atomic protocols are substantially easier to design and verify.

B. Performance Evaluation

In this section we summarize our performance results, provide sensitivity analysis, and evaluate our PushS and ShiftF optimizations.

1) *Performance Summary*: Figure 5 shows Atomic Coherence’s performance relative to the traditional non-atomic MESI (top-left) and non-atomic MOEFISI (top-right). To summarize: relative to non-atomic MESI, atomic MOEFISI achieves on average 14% better performance with 36% fewer transitions. Relative to non-atomic MOEFISI, atomic MOEFISI results in a sub 2%

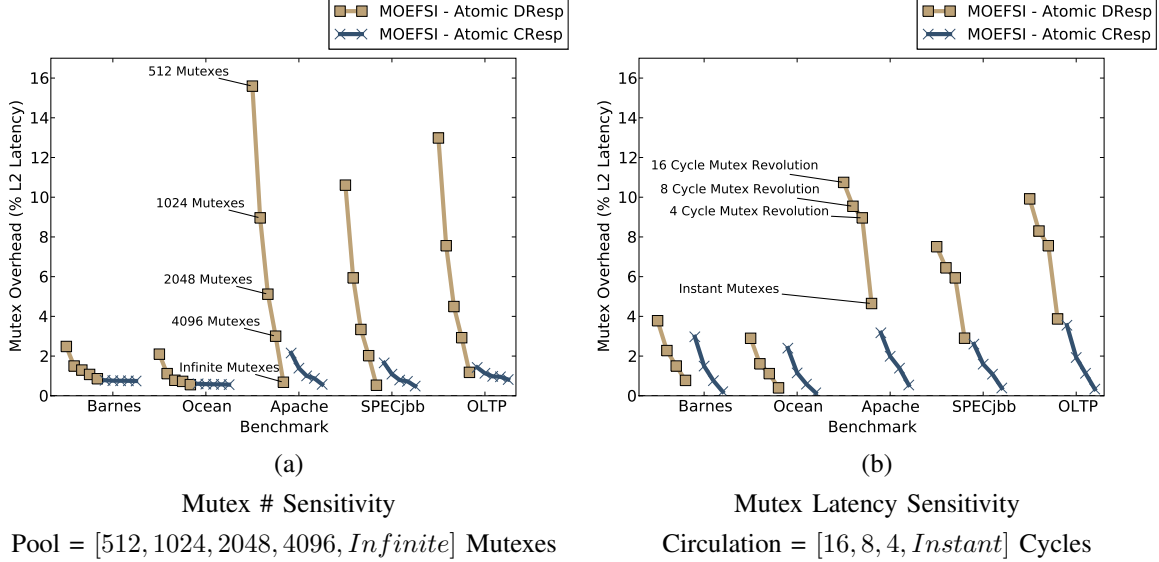


Fig. 6: Sensitivity Study. How the number of mutexes and circulation latency effects the portion of L2 miss time spent waiting to seize a mutex.

slowdown with 57% fewer transitions on average. We believe this is a small price to pay for dramatic protocol simplification.

Looking more closely, we found that Barnes and Ocean, when run with our in-order cores, are bound by computation and are mostly insensitive to the coherence protocol (and also to Atomic Coherence). For the commercial workloads, Atomic CResp performs nearer to the baseline than Atomic DResp, because it benefits from early release of the mutex. The requester eases pressure on the mutex pool by releasing the mutex when data is the only remaining protocol step.

Figure 5(bottom) shows the sources of mutex latency. The category Mutex Circulation Overhead (bottom bar) accounts for the 2 cycles it takes an available mutex to reach the next requester (recall a mutex completes a revolution every 4 cycles). The other categories deal with contention for the mutex. When a mutex is unavailable, it is because either (1) an independent request to the *same* block is active (true positive) or (2) an independent request to a *different* block is active (false positive). The figure shows true positives (top bar) are small contributors to delay, indicating that true positives (races) are rare, an observation echoed by Martin, Hill, and Wood [12]. False positives (middle bar) are the major contributor to latency in Atomic DResp and result from the pressure caused by memory misses holding mutexes for long periods of time. Meanwhile, Atomic CResp does not suffer as much from false positives, because it release the mutex earlier.

Putting the top and bottom parts of Figure 5 together,

we see that the atomic MOEFSI’s incur about 1% slowdown (top-right figure) for every 3–5 cycles of added latency (bottom figure). The sensitivity of performance to small mutex overheads highlights that our very low latency data interconnect requires a very low latency atomic substrate.

Overall, our results indicate that Atomic CResp performs better than Atomic DResp. However, Atomic CResp’s performance is paid for by a slight increase in coherence transitions for queueing of forwarded requests, but, given these tradeoffs, we believe Atomic CResp is the more reasonable and complexity-effective policy of the two.

2) *Sensitivity Analysis:* The mutex overhead is largely dependent on the size of the mutex pool (to avoid false positives) and the circulation time of a free mutex (to promptly seize free mutexes). Below we study how these two factors affect cache miss latency.

a) *Mutex Number.:* Figure 6(a) shows how adding more mutexes, between 512–4096 (or 2–16 waveguides), decreases the mutex overhead. The figure also shows how an unlimited number of mutexes would perform. Atomic DResp is highly sensitive to the pool size, because its mutex-hold times are large. The effect is largely due to queueing delay as responses contend for a limited set of mutex resources. Atomic CResp is less sensitive and has less than 2% overhead across most data points.

b) *Mutex Latency.:* Varying the latency of a mutex’s revolution period also affects the system, as shown in Figure 6(b). The minimum circulation time of a

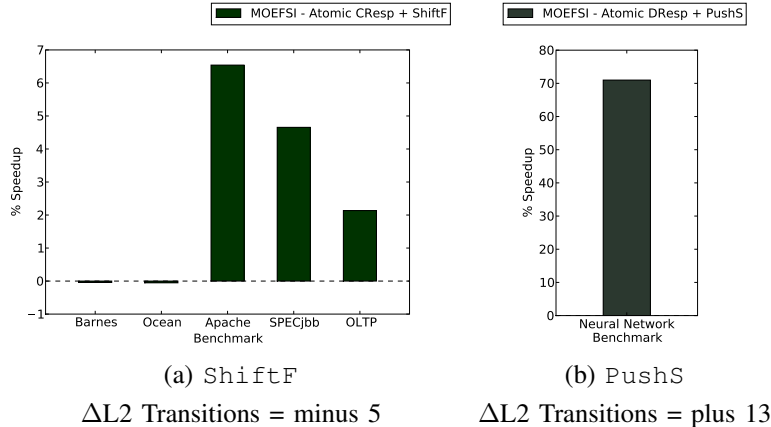


Fig. 7: Atomic Optimization Speedup. Speedup relative to non-atomic MOEFSI, and Δ L2 Transitions are relative to atomic MOEFSI.

mutex is set by light’s propagation in the topology. The maximum circulation time can be arbitrarily extended by adding buffers to the dateline, or another fixed point. We varied the circulation time from 0 (instant) to 16 cycles. Again, Atomic DResp has consistently higher overheads than Atomic CResp. Atomic DResp experiences mutex overheads in the $[0, 11]\%$ range, while Atomic CResp is in the $[0, 4]\%$ range. For the points we looked at, varying this latency has a smaller impact than varying mutex count. Still, the latency remains non-negligible.

3) *PushS and ShiftF Results:* Figure 7 show the performance of our PushS and ShiftF optimizations (from Section III-B and implemented as extensions to Atomic DResp and CResp).

ShiftF improves performance up to 6%, mainly by preserving an on-chip sourcer of data through the F state, improving performance for subsequent read and write misses. Interestingly, in terms of complexity, our implementation actually slightly *reduces* the number of transitions over the Atomic CResp design we began with. The reasons are as follows: When there are sharers on a chip, a miss in Atomic CResp may source the data from memory *or* from a remote cache. A miss in ShiftF may *only* source from a remote cache (if there are sharers, ShiftF guarantees a single F or O sharer). This effect removes some of the possible responses a miss is able to see, thereby shrinking the state-event space.

PushS improves performance by anticipating read misses to a block and forcing the S state data into prospective caches. We found that our scientific and commercial workloads did not benefit from this optimization, because they did not have cyclic patterns of many readers and a single writer. We wrote a neural

network micro-benchmark that exhibits this behavior and captures the potential of PushS, improving performance over non-atomic MOEFSI by 70%. The shared read-write pattern is traditionally expensive to implement and is one reason why programmers use locks sparingly. Our PushS protocol addresses this cost with a simple high-performance implementation.

In summary, we can see that while Atomic Coherence does exact some overhead over the baseline racy protocols, but the addition of simple enhancements can mitigate this penalty. Recall that even our most complex atomic protocol (PushS) is substantially less complex than our baseline MOEFSI protocol, containing 52% fewer transitions (61 vs. 127).

VII. CONCLUSION

In this paper, we advocate nanophotonic support for building high-performance simple atomic protocols.

We propose Atomic Coherence, a framework that decouples race resolution from coherence protocol processing, leading to protocols that only need to follow expected, mostly sequential paths to satisfy coherence requests, and eliminating all unexpected, complex, and difficult-to-verify race transitions from the protocol state machine.

Atomic Coherence not only simplifies protocol design, but also enables straightforward implementations of aggressive protocols, yielding substantial performance improvements (up to 7% for ShiftF and 70% for PushS).

REFERENCES

- [1] J. Archibald and J.-L. Baer, “Cache coherence protocols: evaluation using a multiprocessor simulation model,” *ACM Trans. Comput. Syst.*, vol. 4, no. 4, pp. 273–298, 1986.

- [2] H.-m. D. Toong, S. O. Strommen, and E. R. Goodrich, II, "A general multi-microprocessor interconnection mechanism for non-numeric processing," in *CAW '80*. New York, NY, USA: ACM, 1980, pp. 115–123.
- [3] K. Lepak, "Exploring, defining, and exploiting recent store value locality," Ph.D. dissertation, University of Wisconsin - Madison, 2003.
- [4] D. Abts, Y. Chen, and D. J. Lilja, "Heuristics for complexity-effective verification of a cache coherence protocol implementation," Protocol Implementation, Laboratory for Advanced Research in Computing Technology and Compilers, Tech. Rep., 2003.
- [5] R. Joshi, L. Lamport, J. Matthews, S. d. Tasiran, M. Tuttle, and Y. Yu, "Checking cache-coherence protocols with TLA+," *Form. Methods Syst. Des.*, vol. 22, no. 2, pp. 125–131, 2003.
- [6] D. Abts, S. Scott, and D. J. Lilja, "So many states, so little time: Verifying memory coherence in the Cray X1," in *IPDPS '03*, 2003.
- [7] "Intel Core2 Extreme Processor X6800 and Intel Core2 Duo Desktop Processor E6000 and E4000 Sequence," p. 37, 2008. [Online]. Available: <http://www.intel.com/design/processor/specupdt/313279.htm>
- [8] M. Zhang, A. R. Lebeck, and D. J. Sorin, "Fractal coherence: Scalably verifiable cache coherence," in *MICRO 43*, 2010.
- [9] E. Hagersten and M. Koster, "WildFire: A scalable path for SMPs," in *HPCA '99*, 1999.
- [10] A. Singhal, D. Broniarczyk, F. Cerauskis, J. Price, L. Yuan, C. Cheng, D. Doblar, S. Fosth, N. Agarwal, K. Harvey, E. Hagersten, and B. Liencres, "Gigaplane: A high performance bus for large SMPs," *Hot Interconnects IV*, pp. 41–42, 1996.
- [11] J. Laudon and D. Lenoski, "The SGI Origin: A ccNUMA highly scalable server," in *ISCA '97*, 1997, pp. 241–251.
- [12] M. M. K. Martin, M. D. Hill, and D. A. Wood, "Token coherence: decoupling performance and correctness," in *ISCA '03*, 2003, pp. 182–193.
- [13] J. R. Goodman, "Using cache memory to reduce processor-memory traffic," in *ISCA '83*, 1983, pp. 124–131.
- [14] K. Strauss, X. Shen, and J. Torrellas, "Uncorq: Unconstrained snoop request delivery in embedded-ring multiprocessors," in *MICRO 40*, 2007, pp. 327–342.
- [15] D. B. Gustavson, "The scalable coherent interface and related standards projects," *IEEE Micro*, vol. 12, no. 1, pp. 10–22, 1992.
- [16] A. Raghavan, C. Blundell, and M. M. K. Martin, "Token tenure: PATCHing token counting using directory-based cache coherence," in *MICRO 41*, 2008, pp. 47–58.
- [17] D. Vantrease, R. Schreiber, M. Monchiero, M. McLaren, N. P. Jouppi, M. Fiorentino, A. Davis, N. Binkert, R. G. Beausoleil, and J. H. Ahn, "Corona: System implications of emerging nanophotonic technology," in *ISCA '08*, 2008, pp. 153–164.
- [18] L. Lamport, M. Sharma, M. Tuttle, and Y. Yu, "The wildfire challenge problem," 2001.
- [19] D. D. Corso and H. Kirmman, *Microcomputer Buses and Links*. Orlando, FL, USA: Academic Press, Inc., 1986.
- [20] M. R. Marty and M. D. Hill, "Virtual hierarchies to support server consolidation," in *ISCA '07*, 2007, pp. 46–56.
- [21] E. E. Bilir, R. M. Dickson, Y. Hu, M. Plakal, D. J. Sorin, M. D. Hill, and D. A. Wood, "Multicast snooping: a new coherence method using a multicast address network," in *ISCA '99*, 1999, pp. 294–304.
- [22] A. Kägi, D. Burger, and J. R. Goodman, "Efficient synchronization: let them eat QOLB," *SIGARCH Computer Architecture News*, vol. 25, no. 2, pp. 170–180, 1997.
- [23] A. Moshovos, "RegionScout: Exploiting coarse grain sharing in snoop-based coherence," in *ISCA '05*, 2005, pp. 234–245.
- [24] J. Huh, J. Chang, D. Burger, and G. S. Sohi, "Coherence decoupling: making use of incoherence," in *ASPLOS-XI*, 2004, pp. 97–106.
- [25] A.-C. Lai and B. Falsafi, "Memory sharing predictor: the key to a speculative coherent DSM," in *ISCA '99*, 1999, pp. 172–183.
- [26] N. Eisley, L.-S. Peh, and L. Shang, "Leveraging on-chip networks for data cache migration in chip multiprocessors," in *FACT '08*, 2008, pp. 197–207.
- [27] L. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese, "Piranha: A scalable architecture based on single-chip multiprocessing," in *ISCA-27*, 2000, pp. 282–293.
- [28] H. Hum and J. Goodman, "Forward state for use in cache coherency in a multiprocessor system," Dec 2002, US Patent App. 10/325,069.
- [29] R. N. et al, "Large-scale photonic integrated circuits for long-haul transmission and switching," *J. Opt. Netw.*, vol. 6, no. 2, pp. 102–111, 2007. [Online]. Available: <http://jon.osa.org/abstract.cfm?URI=JON-6-2-102>
- [30] A. Kovsh, I. Krestitnikov, D. Livshits, S. Mikhlin, J. Weimert, and A. Zhukov, "Quantum dot laser with 75nm broad spectrum of emission," *Optics letters*, vol. 32, no. 7, pp. 793–795, 2007.
- [31] D. Vantrease, N. Binkert, R. Schreiber, and M. H. Lipasti, "Light speed arbitration and flow control for nanophotonic interconnects," in *MICRO 42*, 2009, pp. 304–315.
- [32] J. L. Carter and M. N. Wegman, "Universal classes of hash functions (extended abstract)," in *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, 1977, pp. 106–112.
- [33] M. Cianchetti, J. Kerekes, and D. Albonese, "Phastlane: a rapid transit optical routing network," in *ISCA-36*, 2009.
- [34] Y. Pan, J. Kim, and G. Memik, "Flexishare: Energy-efficient nanophotonic crossbar architecture through channel sharing," in *HPCA-16*, 2010.
- [35] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *ISCA*, Jun 1995, pp. 24–36.
- [36] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, M. D. Hill, D. A. Wood, and D. J. Sorin, "Simulating a \$2M commercial server on a \$2K pc," *Computer*, vol. 36, pp. 50–57, February 2003.
- [37] 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 *PLDI '05*, 2005, pp. 190–200.
- [38] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *SIGARCH Comput. Archit. News*, vol. 33, no. 4, pp. 92–99, 2005.