Improving Multiple-CMP Systems Using Token Coherence

Michael R. Marty[1], Jesse D. Bingham[2], Mark D. Hill[1], Alan J. Hu[2], Milo M.K. Martin[3], David A. Wood[1]

[1]Computer Sciences Department
University of Wisconsin-Madison
{mikem, markhill, david}@cs.wisc.edu

[2]Department of Computer Science
University of British Columbia
{jbingham, ajh}@cs.ubc.ca

[3]Dept. of Comp. & Information Science
University of Pennsylvania
milom@cis.upenn.edu

Abstract

Improvements in semiconductor technology now enable Chip Multiprocessors (CMPs). As many future computer systems will use one or more CMPs and support shared memory, such systems will have caches that must be kept coherent.

Coherence is a particular challenge for Multiple-CMP (M-CMP) systems. One approach is to use a hierarchical protocol that explicitly separates the intra-CMP coherence protocol from the inter-CMP protocol, but couples them hierarchically to maintain coherence. However, hierarchical protocols are complex, leading to subtle, difficult-to-verify race conditions. Furthermore, most previous hierarchical protocols use directories at one or both levels, incurring indirections—and thus extra latency—for sharing misses, which are common in commercial workloads.

In contrast, this paper exploits the separation of correctness substrate and performance policy in the recently-proposed token coherence protocol to develop the first M-CMP coherence protocol that is flat for correctness, but hierarchical for performance. Via model checking studies, we show that flat correctness eases verification. Via simulation with micro-benchmarks, we make new protocol variants more robust under contention. Finally, via simulation with commercial workloads on a commercial operating system, we show that new protocol variants can be 10-50% faster than a hierarchical directory protocol.

1 Introduction

The increasing number of transistors per chip now enable *Chip Multiprocessors (CMPs),* which implement multiple processor cores on a chip. CMP-based designs provide high-performance, cost-effective computing for workloads with abundant thread-level parallelism, such as commercial server workloads.

Smaller-scale *Single-CMP (S-CMP)* systems, such as Stanford Hydra [16, 15] and Sun MAJC [37], use a single CMP along with DRAM and support chips. Larger-scale *Multiple-CMP (M-CMP)* systems, such as Piranha [5] and IBM Power4 [36], combine multiple CMPs to further increase performance. Because all of these systems use shared memory (to preserve operating system and application investment), a key challenge for M-CMP systems is implementing correct and high-performance cache coherence protocols. These protocols keep caches transparent to software, usually by maintaining the *coherence invariant* that each block may have *either one writer or multiple readers*. S-CMP systems are conceptually straightforward, in part because designers can leverage the large body of literature on Symmetric Multiprocessors (SMPs) [11] and maintain coherence with traditional non-hierarchical *snooping protocols* (which rely on a logical bus) or *directory protocols* (which track cached copies at memory).

M-CMPs present a greater challenge, because they must maintain both *intra-CMP* coherence and *inter-CMP* coherence. Ideally, an M-CMP protocol would exploit locality by separately optimizing for the low-latency, high-bandwidth intra-CMP communication as well as the higher-latency, lower-bandwidth inter-CMP communication. One approach uses a *hierarchical protocol* to separate the intra-CMP coherence protocol from the inter-CMP protocol, but couples them hierarchically to maintain the coherence invariant. This approach can apply experience with non-CMP hierarchical protocols [7, 13, 14, 19, 21, 38] to CMPs [5, 36]. Figure 1 illustrates (a) a CMP node and (b) an M-CMP with a hierarchical coherence protocol.

Hierarchical coherence presents at least two challenges. First, even non-hierarchical coherence protocols are difficult to implement correctly. Coupling two protocols into a hierarchy creates additional transient states and protocol corner cases, significantly increasing verification complexity [3, 6]. Races occur both among messages within each CMP (e.g., processor requests to readable/writable blocks, writebacks, invalidations, acknowledgments) and between CMPs (e.g., forwarded requests, data messages, and acknowledgments). These messages lead to many transient states in L1 caches, L2 caches, and directory controllers, particularly with opti-

Proceedings of the 11th Int'l Symposium on High-Performance Computer Architecture (HPCA-11 2005)
1530-0897/05 $20.00 © 2005 IEEE

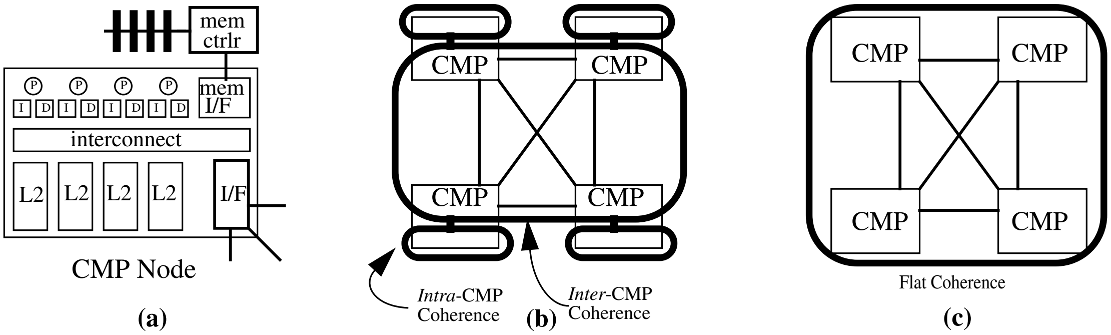

Figure 1. A CMP Node with Two Alternative Multiple CMP (M-CMP) Systems. Part (a) expands a CMP node with processors, private L1 caches, shared L2 cache banks, on-chip interconnect, global interconnect interface, and interface to off-chip memory controller with DRAM. Part (b) symbolizes a direct-interconnect M-CMP system with coherence maintained via an *intra*-CMP protocol interacting with a *inter*-CMP protocol. Part (c) symbolizes an M-CMP with a logically-flat coherence protocol, such as *TokenCMP* developed in this paper.

mizations (e.g., all MOESI states). Second, many previous hierarchical protocols—non-CMP [7, 13, 14, 19, 21, 38] or CMP [5]—use directories at one level of the hierarchy (an exception is Power4 [36]). Directory protocols require indirections (and thus additional latency) on sharing misses common in many commercial workloads [4]. Section 2 presents our base M-CMP system, which uses directory protocols for both intra-CMP and inter-CMP coherence.

In contrast, *token coherence* [23, 25] is well suited to the present challenge, because it explicitly separates the *correctness substrate* from a *performance policy.* The correctness substrate uses token counting to guarantee that, at any time, a memory block can have a single writeable copy (with all tokens), multiple read-only copies (with one or more tokens), or is not cached (all tokens at memory). To avoid starvation, it uses *persistent requests* that are remembered at other nodes until the requestor garners sufficient tokens.

A token coherence performance policy, on the other hand, uses unacknowledged *transient requests* to seek tokens and data. It can optimize common patterns, use complex predictors, and re-issue transient requests that don't find sufficient tokens. In all cases, the correctness substrate continues to provide safety (by token counting) and avoids starvation (by eventually issuing a persistent request). The original token coherence performance policy [25], however, is not well-suited for M-CMP systems because it assumes flat glueless multiprocessors with private caches.

This paper extends token coherence to M-CMP systems by developing *TokenCMP.* TokenCMP provides coherence in a manner that is *flat for correctness (*Section 3*), but direct and hierarchical for performance* (Section 4). We demonstrate the following three contributions:

- **Simplicity.** TokenCMP can be shown correct by verifying only a flat correctness substrate. Via model checking, we show the effort required is comparable or less than verifying a flat directory protocol, which is known to be much simpler than verifying hierarchical directory protocols (Section 5).
- **Robustness.** Under contention, the original token coherence proposal used a persistent request mechanism that could become a performance bottleneck. We extend the original starvation avoidance mechanism with persistent read requests and a distributed activation mechanism. We use microbenchmarks to show that TokenCMP variants can handle highly-contended blocks more robustly (Section 7).
- **Performance.** We evaluate the performance of TokenCMP versus a hierarchical directory protocol. We simulate three commercial workloads interacting with a commercial operating system on an M-CMP system using four 4-way CMPs (Section 6). We show TokenCMP can be 9-50% faster than a hierarchical directory protocol (Section 8).

2 Base M-CMP System & DirectoryCMP

In this paper, we compare TokenCMP against a base M-CMP system that uses directory protocols, not snooping, for both inter-CMP and intra-CMP coherence (similar to Piranha [5]). This approach allows both on- and off-chip interconnects to be unordered and fully-connected to reduce latency.

Figure 1 shows block diagrams of a CMP node and the 4-CMP system evaluated in this study. Each CMP contains four processors, private L1 instruction and data caches, shared L2 cache banks, an on-chip interconnect, a global interconnect interface, and an interface to an off-chip memory controller, which in turn connects to DRAMs. Using a separate memory controller allows the CMP to dedicate more pins for the global interconnect and support greater memory bandwidth and capacity.

Our base system uses an MOESI-based hierarchical directory protocol called *DirectoryCMP*. Each L2 cache bank maintains an intra-CMP directory to track copies in L1 caches. The intra-CMP directory controller maintains coherence with messages among the on-chip caches, and it interfaces with the inter-CMP directory protocol.

In the inter-CMP directory protocol, each memory controller maintains an inter-CMP directory to track which CMP nodes cache a block, but not which caches within the CMP hold the block. The inter-CMP directory maintains coherence with messages between itself and the appropriate L2 cache bank at each CMP.

The intra- and inter-CMP directories and protocols cooperate to maintain M-CMP coherence. An L1 miss sends a coherence request to the appropriate on-chip L2 bank. Depending on the intra-CMP directory state and the L2 cache state, a response is directly returned, or the request is forwarded to on-chip L1 caches or to the inter-CMP directory (located at the home memory controller for the block). As responses return through the hierarchical network, they update the appropriate cache and directory state.

In designing DirectoryCMP, we made choices that improved runtime at the expense of additional control messages. DirectoryCMP implements a migratory sharing optimization [10, 34], in which a cache holding a modified cache block invalidates its copy when responding with the block, thus granting the requesting processor read/write access to the block (even if the request was only for read access). This optimization substantially improves performance of many workloads with read-modify-write sharing behavior. To moderately reduce DirectoryCMP complexity and enable the use of optimizations such as migratory sharing, the protocol uses per-block busy states at both the intra-CMP and inter-CMP directories to defer requests to blocks with outstanding requests. It also uses three-phase writebacks, at both levels, to defer and coordinate writebacks with other requests.

3 Flat Correctness Substrate

Token coherence provides safety (all actions correct) and starvation avoidance (take appropriate actions) with a *correctness substrate* that is separate from a *performance policy* [25]. This section describes token coherence's correctness substrate and our enhancements to the substrate for shared caches and better performance robustness under contention.

3.1 Safety

Token coherence's correctness substrate directly enforces the coherence invariant—that each block can have *either a single writer or multiple readers*—with the simple mechanism of counting tokens. Each block always has *T tokens*, one of which is distinguished as the *owner token*. Tokens are stored at and exchanged among processor nodes and memory using $1+\lceil \log_2 T \rceil$-bit token counts. A processor node with all T tokens may write a block, while a node with one or more tokens may read a block. To allow all processor nodes to share a block, T should be at least as large as the number of nodes. Messages with the owner token must contain valid data, while messages with only non-owner tokens may omit data (to save bandwidth).

The original scheme implicitly assumes that once tokens are given to a node, it is straightforward to maintain coherence within the node. This assumption is reasonable for a uniprocessor node, but it is not true for M-CMPs, in which each node has multiple processors with private L1 caches and a shared L2 cache.

Fortunately, we can enforce safety in M-CMP systems simply by passing tokens among caches, rather than among nodes. Thus, each cache—L1 data, L1 instruction, and unified L2 bank—essentially acts like a "node". A processor may read (fetch) a block if it's L1 data (instruction) cache has at least one token; it may write a block if its L1 data cache has all T tokens. Caches exchange data and tokens following the original token coherence rules. A block may be simultaneously cached in at most T caches. Fortunately, doubling T (e.g., to accommodate more caches than processors) adds only 1 bit to the token count.

This flat correctness substrate provides processors within an M-CMP the same simple safety checks to enforce the coherence invariant that existed in the original token coherence protocol. All the additional complexity introduced by an M-CMP's physical hierarchy (e.g., finding a block locally within a CMP when possible) is handled by a performance policy (Section 4).

3.2 Starvation Avoidance

Because token counting guarantees safety, performance policies may use unacknowledged transient requests to aggressively seek tokens without the ordering restrictions imposed by conventional protocols. This provides the flexibility to optimize hierarchical performance, but also means that transient requests may miss in-flight tokens. To ensure this situation does not lead to starvation, the correctness substrate issues a *persistent request* when a processor fails to acquire sufficient tokens within a time interval. The substrate *activates* at most one persistent request per block. Coherence components remember all *activated* persistent requests and forward all tokens for the block—those tokens currently present and received in the future—to the initiator of the request. When the initiator has sufficient tokens, it performs a memory operation (*e.g.*, a load or store instruction) and *deactivates* its persistent request. In addition, the system must provide a starvation-free mechanism for activating persistent requests. For glueless multiprocessors, Martin et al. [25] employ arbiters and fair queuing to select one active persistent request per memory controller.

To avoid starvation in an M-CMP system, we consider two persistent request mechanisms. One extends the arbiter-based scheme and the other uses a new distributed activation approach [23]. Both also include new mechanisms, discussed below, to improve worst-case performance on contended blocks.

Arbiter-based Activation. Extending the original arbiter-based persistent request mechanism to M-CMPs is straightforward, but requires each cache, not just each node, to remember active persistent requests. Using arbiter-based persistent requests provides flat starvation avoidance in M-CMP systems. Furthermore, the tables for storing active persistent requests are small (e.g., 384 bytes for 64 six-byte entries) and directly addressed.

However simple, the arbiter-based activation mechanism lacks *performance robustness*. That is, when performance gets bad, persistent requests tend to make it worse because the handoff from one persistent request to the next requires an indirect deactivate/activate exchange with the arbiter, increasing both latency and bandwidth consumption. Although this has little effect for well-tuned workloads, we seek an alternative mechanism that avoids surprises with more demanding applications.

Distributed Activation. The new scheme improves worst-case performance by using a distributed arbitration mechanism to directly forward contended blocks to the processor's requesting L1 cache with the next active persistent request. Each processor initiates at most one persistent request, and each cache remembers these persistent requests in a table (each table has one entry per processor). The table activates only the highest priority persistent request of those in the table seeking the same block. Priority among persistent requests is fixed (e.g., by processor number). When a cache receives a message deactivating a persistent request, it clears the corresponding table entry. When a processor deactivates its own persistent request, its local table "marks" all valid entries for the same block by setting a bit in the entry. A processor is allowed to issue a persistent request only when no marked entries for the desired block are present in its local persistent request table. The marking mechanism prevents a processors from continually issuing persistent requests that starve out another processor. This approach is loosely based on FutureBus [35] arbitration, which uses a fixed priority but groups processors into "waves" to prevent them from re-requesting bus access until all current wave members obtain access.

Distributed activation reduces the average persistent request latency by forwarding highly contended blocks directly between processors. For example, let processors P1, P2, and P3 seek block B with persistent requests. All three will remember each other's requests, but activate only the highest priority request, say, processor P1's. When P1 succeeds, it deactivates its request, activates P2's request, and sends block B to P2. When P2 is done, it sends block B directly to P3. In this way, the distributed scheme provides a minimum latency hand-off on highly-contended blocks (e.g., hot locks). Secondarily, locality can be enhanced by simply fixing processor priority so that least-significant bits vary for processors within a CMP and more-significant bits vary between CMPs. In particular, with this approach, highly-contended spin locks tend to dynamically perform much like complex hierarchical or reactive locks [20, 30].

Distributed activation is implemented with small tables, like the arbiter-based scheme, but with a content-addressable access. When tokens for block B arrive, the table is searched for an active persistent request for block B, and, if found, tokens are forwarded. When a new persistent request for block B arrives, the table is searched for an active persistent request to block B. The incoming request is inserted and made active (possibly forwarding tokens) depending on the priority of the requestor. Furthermore, implementation is straightforward and like a fully-associative translation lookaside buffer, but with a multi-cycle access time being acceptable.

Persistent Read Requests. The original persistent request mechanism always collects all tokens, regardless of whether the starving processor wants to read or write a block. This approach performs poorly for certain access patterns, such as a highly-contended test-and-test-and-set lock. We implement a new *persistent read request* that forces all caches to give up all but one token [23]. As long as the total number of tokens is

Table 1. TokenCMP Variants

Name	Performance Policy	# Transient Requests	Persistent Request Activation
TokenCMP-arb0	None: immediately issues persistent requests	0	Arbiter-based
TokenCMP-dst0	None: immediately issues persistent requests	0	Distributed activation
TokenCMP-dst4	Hierarchical protocol with 1 transient request & up to 3 retries	up to 4	Distributed activation
TokenCMP-dst1	Hierarchical protocol with 1 transient request, but no retries.	1	Distributed activation
TokenCMP-dst1-pred	Hierarchical protocol with predictor to choose immediate persistent request or a single transient request.	0 or 1	Distributed activation
TokenCMP-dst1-filt	Like TokenCMP-dst1 with filter on incoming external requests.	1	Distributed activation

greater than the number of caches, this approach (1) guarantees the requester will receive at least one token and (2) avoids stealing read permission away from other caches. The correctness substrate issues persistent reads when a processor fails to make progress on a load, and issues the original persistent request for stores and atomic memory operations.

Response Delay Mechanism. Highly-contended locks can result in the coherence protocol prematurely stealing permissions from the processor executing a critical section. To improve locking performance, all protocols implement a simple, non-speculative delay mechanism that ensures that a processor holds permissions for a block long enough to perform a short critical section (inspired by Rajwar et al. [31]). Adding a bounded delay does not affect starvation-avoidance guarantees.

3.3 Token and Data Transfer

The correctness substrate ensures that data and tokens are transferred without loss or corruption. Like most cache coherence protocols, this guarantee requires that the interconnection network eventually delivers each message accurately. The performance policy, discussed next, invokes the correctness substrate to reliably transfer data and tokens on its behalf.

4 Hierarchical Performance Policy

The safety and starvation-freedom guarantees provided by the correctness substrate enable aggressive performance policies. For example, a performance policy can optimize for common cases, without concern for the races that make conventional protocol optimizations so complex.

The original *TokenB* performance policy targets flat modestly-sized glueless multiprocessors with low-latency, high-bandwidth, unordered interconnects [25]. TokenB broadcasts *transient requests* to all nodes. Nodes respond to these transient requests with one or more tokens, however, a transient request may fail to collect sufficient tokens. In such situations, a processor re-broadcasts its transient request after a timeout threshold. TokenB monitors average response times to determine this timeout threshold, and it re-broadcasts a transient request up to three times. After a fourth timeout the substrate issues a persistent request. Pseudo-random backoff avoids lock-step retries.

TokenB is not well-suited for an M-CMP system. First, the timeout threshold does not account for the difference in response latency between local and remote caches. Second, broadcasting requests requires greater cache lookup bandwidth, since all L1 and L2 caches may hold tokens. Third, requests do not exploit locality; doing so can reduce both latency and inter-CMP bandwidth by finding tokens and data within the local CMP. Finally, it may be worthwhile to filter external requests arriving at a CMP to save the intra-CMP bandwidth of forwarding the requests to all L1 caches. TokenB does not consider these optimizations because it assumed a flat system.

Table 1 lists the TokenCMP variants evaluated in this study. TokenCMP-arb0 and TokenCMP-dst0 use no performance policy and never issue transient requests; instead, they rely on the correctness substrate to immediately issue a persistent request for all processor requests. We use these variants to stress robustness, but we do not recommend deploying them in real systems (because persistent requests are less efficient than transient requests). TokenCMP-arb0 uses the original arbiter-based persistent request activation mechanism, while TokenCMP-dst0 and all subsequent variants use the new distributed activation mechanism (described in Section 3.2).

The last four TokenCMP variants have much in common. On an L1 miss, each broadcasts a coherence request only *within* its CMP to the appropriate on-chip L2 cache bank and other on-chip L1 caches. The local L1 caches check their tags. On a write request, an L1 replies with data (if it holds the owner token) and all its tokens. An L1 cache responds to a local read request if it has multiple tokens. If it has all tokens and has modified the data, it optimizes for migratory sharing by transferring the data and all tokens. Otherwise it responds with data and one token.

On an L2 miss, each CMP broadcasts the request to other CMPs. A CMP responds to external write requests by returning all tokens (and data if it holds the owner token). A CMP responds to external read requests only if it holds the owner token. To reduce the latency of a future intra-CMP request, read responses include *C* tokens (if possible), rather than the necessary 1 token, where *C* is the number of caches on a CMP node. A cache may also respond to a read request with all *T* tokens to optimize for migratory sharing.

In all cases, a cache only responds to external requests when it actually has tokens. In contrast, a traditional protocol may need to track or block pending requests, allowing it to create a queue of pending requests for a contended block.

Finally, the TokenCMP variants set their timeout threshold using responses from memory. We found that TokenB's approach of averaging in the latency of all responses led to a rapid burst of retries, because fast on-chip hits accounted for a substantial portion of the running average.

The last four TokenCMP variants differ as follows. TokenCMP-dst4 follows TokenB's approach of issuing three retries (four transient requests total) before resorting to a persistent request. In contrast, TokenCMP-dst1 uses a persistent request immediately after the initial transient request times out. This policy exploits the lower latency of the new distributed activation mechanism.

TokenCMP-dst1-pred adds a predictor to detect highly-contended blocks and immediately issue a persistent request to avoid a potential timeout. Our base predictor uses a four-way set-associative 256-entry table of 2-bit saturating counters (other configurations performed similarly). A counter is allocated and incremented when a transient request is retried. Counters are reset pseudo-randomly to allow adaptation to different phase behaviors.

Finally, TokenCMP-dst1-filt filters external transient requests to conserve intra-CMP bandwidth. Each L2 bank maintains an approximate directory of L1 sharers and forwards external transient requests to only those caches. This filtering can be approximate because the correctness substrate provides safety and prevents starvation (persistent requests are never filtered). This approach contrasts with previous coherence filters that could cause coherence violations if they filtered too many coherence requests [28, 32].

For our workloads and system size (16 processors in four 4-processor CMPs), however, we will see that TokenCMP-dst1-pred and TokenCMP-dst1-filt may not contribute enough to justify their implementation costs. Nevertheless, these and other ideas (e.g., multicast via destination set prediction [23, 24]) may be more valuable in larger systems.

5 Complexity Discussion & Results

Quantifying the design and verification complexity of a system is notoriously hard, because what really matters is the subjective complexity experienced by the human designers, rather than some easily measurable quantity. A clean, modular design might be larger in terms of lines of code or number of transistors, yet be far easier to understand, design, debug, and modify. We justify our claim of simplicity two ways. First, we provide concrete examples of how we subjectively found TokenCMP variants easier to design and modify. Second, we present objective results from model-checking experiments, which show that the correctness substrate shared by all TokenCMP variants has comparable model-checking complexity to a simplified, non-hierarchical version of DirectoryCMP in which all intra-CMP details are omitted.

Subjective Experience. As an example of the greater simplicity of TokenCMP, consider writebacks. Handling writebacks correctly is difficult in a flat coherence protocol and even harder in a hierarchical one. Traditional directory protocols often require two-phase or three-phase writebacks of dirty blocks to handle races and complications arising from protocol optimizations. The root of the complexity in these protocols is that all requests must find the pertinent copies of the block, even when they are in transit as part of a writeback operation. In contrast, writebacks are much simpler in protocols based on token coherence. When a cache needs to write back a block (dirty or otherwise), it simply sends tokens and (in some cases) data to either the L2 or memory; no extra messages or transient states at any caches or memory are required. A request that misses any in-flight tokens may be reissued, and the substrate will eventually invoke a persistent request to ensure that all misses eventually complete. This same property allows token coherence variants to more simply handle multiple concurrent requests to the same block.

Also, in our experience, TokenCMP is easier to change and debug than DirectoryCMP. For example, we can add or remove the migratory sharing optimization by changing the number of tokens returned in response to a read request. Adding this optimization to TokenCMP required only one additional state and a few small modifications to protocol finite state machines. Moreover, these changes are clearly correct, because they do not affect the correctness substrate. In contrast, implementing the migratory sharing optimization in a flat directory protocol was somewhat complex and doing so in a hierarchical directory protocol was even more challenging.

Model Checking. In addition to our subjective experience, we performed model-checking experiments in an effort to objectively quantify the relative complexity of TokenCMP and DirectoryCMP, as well as to increase our confidence in the correctness. Model checking is a technique for verifying properties of complex systems by exhaustively exploring the state space [9, 29]. Model checking has become almost routine for enhancing confidence (and finding bugs) in cache coherence protocols, and the literature is too vast to survey here comprehensively. We simply note a few facts. (1) Model checking provides exhaustive analysis, completely analyzing obscure corner cases and protocol interactions. It provides a thoroughness difficult to achieve via other analysis approaches. (2) The exhaustive analysis is also the Achilles' heel of model checking. Because the state space explodes exponentially, only very small or highly simplified configurations can be model checked successfully. With reasonably detailed protocol models, tiny configurations with only a few caches and a few blocks per cache are the limit of the state-of-the-art. (3) Despite those tiny configurations, model checking often finds bugs, e.g., McMillan and Schwalbe's seminal work for the Encore Gigamax [26] through Joshi et al.'s recent results on the EV6 and EV7 [17] (Braun, et al. [8] cites numerous other case studies.) (4) We are not aware of any published work that has reported model checking a detailed model of a hierarchical protocol *as a hierarchical protocol.* The model would simply be too large. Instead, previous work considered only one layer of the hierarchy at a time, manually abstracting away the other layers.

We used the TLA+ description language [18] and its TLC model checking tool [18, 40] to model and verify TokenCMP variants and a non-hierarchical simplification of DirectoryCMP that omits all intra-CMP details. We used standard techniques for simplifying the protocols to enable model checking, e.g., symmetry, down-scaling [12], data independence [39], etc. We verified three versions of the token coherence correctness substrate: (1) TokenCMP-arb, (2) TokenCMP-dst, and (3) TokenCMP-safety, a simplified TokenCMP, used only for easily verifying safety properties, that lacks any starvation-prevention mechanisms. We modeled only the correctness substrate and the interfaces used by any performance protocol. By allowing the model to nondeterministically invoke these interfaces, we verify the correct behavior of not just one performance protocol, but *all* possible performance protocols.

We verified that the protocols were free of deadlock and provided a serial view of memory, in which every load returns the value of the most recent store to the same location [27]. We also verified that the persistent request mechanisms in TokenCMP variants ensure that the system eventually satisfies all requests, under certain fairness constraints, e.g., messages are eventually delivered, and once a persistent request is satisfied it is eventually deactivated.

We were able to verify the correctness of all versions of TokenCMP for small configurations. The model-checking complexity was similar between TokenCMP-arb and the simplified, non-hierarchical version of DirectoryCMP. TokenCMP-dst was somewhat more computationally intensive to verify; TokenCMP-safety was somewhat less intense to verify because it omits any persistent request mechanism.

Furthermore, the number of non-comment lines of TLA+ descriptions is 383 lines for TokenCMP-arb and 396 lines for TokenCMP-dst, versus 1025 for the simplified, flat DirectoryCMP. Obviously, the size of the TLA+ descriptions is only an indirect complexity metric and depends on various modeling decisions and coding style. However, we feel that this metric accurately reflects the benefit of decoupling correctness from performance in shared memory protocols: the brevity of the token coherence TLA+ description stems from the fact that only the correctness substrate need be verified. The directory protocol does not afford such a reduction because there is no clean division between correctness and performance.

The model checking results highlight that, because the correctness substrate is flat, the TokenCMP approach is as model-checkable as a typical flat directory protocol, which is important because only flat protocols (or flat protocols manually sliced from hierarchical protocols) are currently model-checkable. Furthermore, because of token coherence's separation of correctness from performance, our model checking results apply immediately to *any* performance policy, including hierarchical ones. In contrast, to model check DirectoryCMP either we would need to model check a full, hierarchical M-CMP configuration, which is computationally intractable, or else we would have to resort to manual reasoning to justify abstracting away the intra-CMP protocol and hope that all corner cases have been handled correctly. We conclude that TokenCMP is simpler than a hierarchical directory protocol.

Table 2. Benchmark Descriptions

Locking Micro-benchmark. In this micro-benchmark, each processor thinks for 10 ns, acquires a random lock (different from the last lock acquired), holds the lock for 10 ns, and repeats until the total number of acquires performed by each processor reaches a pre-determined limit. Lock acquires use test-and-test-and-set [11] and contention is varied by changing the number of locks.

Barrier Micro-benchmark. This micro-benchmark models processors performing local work, waiting for a sense-reversing barrier [11], and repeating 99 more times. Local work takes 3000 ns with and without optional variability. When each processor reaches the barrier, it acquires a lock and increments a count in the same cache block. If the count is not maximum, the processor releases the lock and spins on a flag in another cache block. If the count is maximum, the processor zeros the counter, reverses the sense of the flag, and releases the lock. All processors now pass the barrier and begin the next work phase.

Apache: Static Web Content Serving. Web servers such as Apache are an important enterprise server application. We use Apache 2.0.43 configured to use a hybrid multi-process multi-threaded server model with 64 POSIX threads per server process. We use 800,000 requests to warm the system, 1000 requests to warm simulated hardware caches, and detailed simulations of 100 requests for our reported results.

OLTP: Online Transaction Processing. DB2 with a TPCC-like workload. The TPC-C benchmark models the database activity of a wholesale supplier. Our OLTP workload is based on the TPC-C v3.0 benchmark using IBM's DB2 v7.2 EEE database management system. We use 10,000 transactions to warm the system and database buffer pool, 500 transactions to warm simulated hardware caches, and detailed simulations of 100 transactions for our reported results.

SPECjbb: Java Server Workload. SPECjbb2000 is a server-side Java benchmark that models a 3-tier system, but its main focus is on the middleware server business logic. We use over a million transactions to warm the system, 100,000 transactions to warm simulated hardware caches, and detailed simulations of 2000 transactions for our reported results.

Table 3. Target System Parameters

Each CMP	
number of processors	4 per CMP
cache block size	64 Bytes
split L1 I & D caches	128kBytes, 4-way, 2ns
interconnect topology	directly connected
interconnect link bw	64 GBytes/sec
interconnect latency	2ns (one-way)
shared unified L2 cache	8MByte, 4-banks, 4-way, 7ns
memory/dir controllers	6ns latency
Each Dynamically Scheduled Processor	
clock frequency	2 Ghz
reorder buffer/scheduler	128/64 entries
pipeline width	4-wide fetch & issue
pipeline stages	11
direct branch predictor	1kBytes YAGS
indirect branch predictor	64 entry (cascaded)
return address stack	64 entry
Per-CMP Memory	
latency to mem controller	20ns (off-chip)
DRAM latency	80ns
memory bank capacity	1 GByte per bank
Between CMPs	
number of CMPs	4 (16 processors total)
interconnect topology	directly connected
interconnect link bw	16 GBytes/sec
interconnect link latency	20ns (including interface, wire, & sync.)

6 Methods

This section describes the commercial workloads, target M-CMP system assumptions, and simulation methods we use for our performance evaluation.

Benchmarks. We evaluate protocols with commercial workloads from an enhanced version of the *Wisconsin Commercial Workload Suite* [1]. As detailed in Table 2, we use locking and barrier micro-benchmarks, a static web serving workload (Apache), an online transaction processing workload (OLTP), and a Java middleware workload (SPECjbb). The macro-benchmarks execute on a simulated SPARC multiprocessor running Solaris 9, while the micro-benchmarks use a testing facility immune to operating system effects.

Target M-CMP System. We target an M-CMP system that uses four directly-connected CMPs, each containing four dynamically-scheduled SPARC processors (16 processors total) with a total system memory of 4GB. We focus on 16-processor systems because (1) most multiprocessor systems have a small or moderate number of processors, and (2) the commercial workloads on which we focus can less easily exploit scalable multiprocessing systems (in contrast to technical workloads). Figure 1a depicts the CMP node, while Table 3 provides additional assumptions.

Target M-CMP Coherence Protocols We evaluate several alternative M-CMP protocols:

- *DirectoryCMP* provides coherence hierarchically (Figure 1b) with the two-level directory protocol described in Section 2. We show results for both a DRAM directory and an unrealistic zero-cycle directory (*DirectoryCMP-zero*).
- *TokenCMP* variants are introduced in Section 4 and summarized in Table 1.
- *PerfectL2* provides an unimplementable lower bound. All L1 misses hit in an infinite L2 cache shared across all CMPs.

Simulation Infrastructure. We simulate target M-CMPs with the Virtutech Simics full-system functional execution-driven simulator [22] and a perfor-

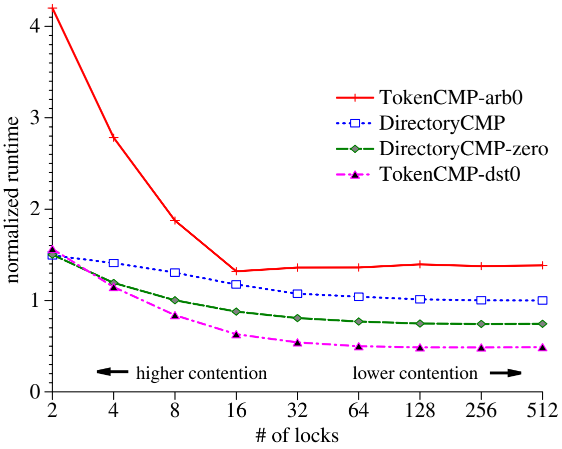

Figure 2. Locking micro-benchmark results using only persistent requests

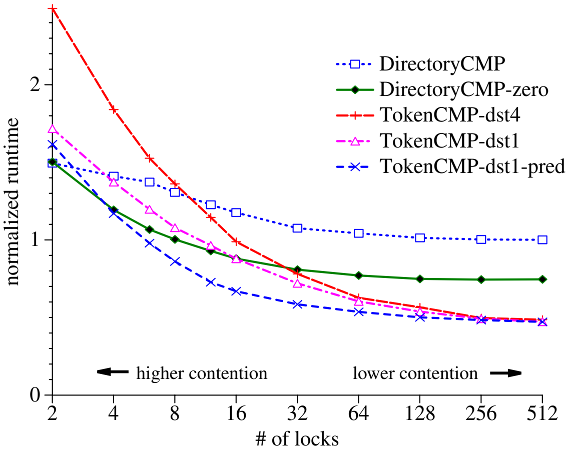

Figure 3. Locking micro-benchmark results with both transient and persistent requests

mance simulation infrastructure used to simulate memory hierarchies and out-of-order processors [1]. We pseudo-randomly perturb simulations and calculate error bars as described by Alameldeen et al. [2]. Improvements in the next section are statistically significant with 95% confidence when error bars do not overlap. We extended this infrastructure to model an M-CMP's physical hierarchy and specified both DirectoryCMP and TokenCMP variants in a table-driven language for protocol specification [33].

7 Robustness Results

The distributed persistent request mechanism (Section 3.2) tries to improve performance robustness. We evaluate how well it does this using the locking and barrier micro-benchmarks and performance policies that use *only* persistent requests. Figure 2 shows runtime (smaller is better), normalized to DirectoryCMP with 512 locks, for 16 processors as the number of locks varies from 2 (high contention) to 512 (low contention). The middle two lines show DirectoryCMP with a realistic directory and an unrealizable zero-cycle directory. The other two lines show TokenCMP variants that use only persistent requests. We see that the original arbiter method (TokenCMP-arb0) performs worse than DirectoryCMP, while the new distributed method (TokenCMP-dst0) performs comparably or better than the directory variants. Not shown, TokenCMP-arb0 performs even worse when highly-contended locks map to the same arbiter, while the distributed method is immune to where locks map.

Although TokenCMP-dst0 has good runtime for this micro-benchmark, its exclusive use of persistent requests is not well suited for macro-benchmarks, in part, because of the traffic of broadcasting activate and deactivate messages to all caches. Instead, our goal is to develop protocols that are both (1) robust for contended micro-benchmarks and (2) perform well for macro-benchmarks.

Figure 3 shows runtime results for the various TokenCMP performance policies (normalized to DirectoryCMP with 512 locks). For low contention (e.g., 512 locks), the results show that (1) TokenCMP variants perform well and (2) TokenCMP outperforms DirectoryCMP. This result occurs because the requested lock is often in an L1 cache in another CMP, causing many directory indirections in DirectoryCMP.

As contention increases, TokenCMP variants differ. TokenCMP-dst4 is not robust, because it wastes time issuing three retries that often fail before issuing a successful persistent request. TokenCMP-dst1 does better, and comparable to directory variants, by issuing a persistent request immediately after an initial transient request fails. Finally, TokenCMP-dst1-pred does better by using persistent requests immediately in high contention and acting like TokenCMP-dst1 in low contention. Not shown, TokenCMP-dst1-filt performs identically to TokenCMP-dst1.

To further exercise robustness, we also compare protocols using the barrier micro-benchmark from Table 2. Results in Table 4 show runtimes (normalized to DirectoryCMP) for the various protocols, in which the work each processor does between barriers takes either a constant 3000 ns (middle column) or has some uniform variability (right). These results (and results using other parameters not shown) corroborate locking micro-benchmark results that TokenCMP-arb0 and TokenCMP-dst4 should be avoided (results highlighted in bold).

In summary, TokenCMP-dst1, TokenCMP-dst1-pred, and TokenCMP-dst1-filt all provide robust performance even under high contention.

Table 4. Barrier Micro-benchmark Runtime (Normalized to DirectoryCMP)

	Work between barriers	
Protocol	3000 ns fixed	3000 ns + U(-1000,+1000)
TokenCMP-arb0	**1.40**	**1.29**
TokenCMP-dst0	0.94	0.91
DirectoryCMP	1.00	1.00
DirectoryCMP-zero	0.95	0.93
TokenCMP-dst4	**1.15**	1.01
TokenCMP-dst1	0.99	0.95
TokenCMP-dst1-pred	0.96	0.93
TokenCMP-dst1-filt	0.99	0.95

8 Performance Results

This section evaluates TokenCMP performance using commercial workloads, presenting runtime and intra-CMP and inter-CMP bandwidth results.

Runtime. Figure 6 displays runtime results for the macro-benchmarks from Table 2, normalized to DirectoryCMP. Hash marks for a perfect global L2 cache bound the possible improvement, while DirectoryCMP is shown with both a DRAM directory and an unrealistic 0-cycle directory. We find:

- **The TokenCMP variants perform significantly better than DirectoryCMP.** In particular, TokenCMP-dst1 is faster than DirectoryCMP (with DRAM directory) by 50% for OLTP, 29% for Apache, and 10% for SpecJBB.[1]
- **All TokenCMP variants perform similarly**. This implies that contention is modest and changes to improve robustness did not hurt. Persistent requests occur rarely—less than 0.3% of L1 misses for all workloads and protocols.
- **TokenCMP-dst1 is best.** It is more robust than TokenCMP-dst4 with similar macro-benchmark performance. The cost of TokenCMP-dst1-pred's predictor and TokenCMP-dst1-filt's filter are not justified for these workloads and system sizes.

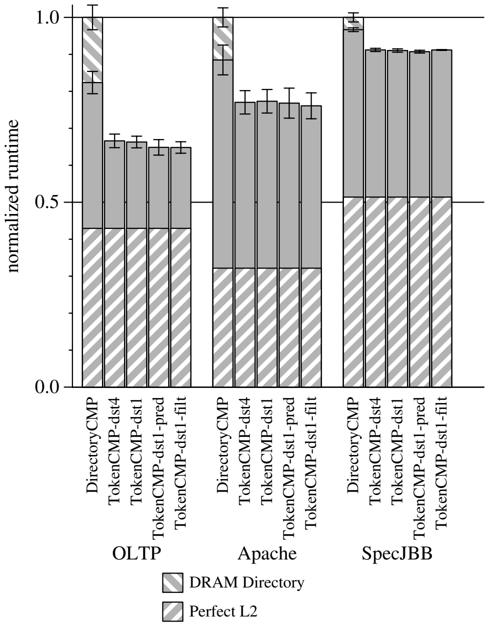

Figure 6. Runtime of commercial workloads

Inter-CMP Bandwidth. For our parameters, inter-CMP traffic generates little queuing delay. Nevertheless, to examine possible effects for other assumptions, we plot inter-CMP traffic in Figure 7a and break it down by message type. Results are in bytes and normalized to traffic of DirectoryCMP. Data messages are 72 bytes and control messages 8 bytes. Results show TokenCMP variants generate somewhat *less* traffic than DirectoryCMP. We initially believed this result incorrect, because TokenCMP uses broadcast between nodes. Nevertheless, further investigation supported the result by revealing that DirectoryCMP can send more control messages than TokenCMP. Consider, for example, a sequence in which a CMP obtains an exclusive copy of a block from remote memory, updates it, and writes it back to memory. With TokenCMP, a CMP sends three request messages to the other CMPs, receives a data message, and then sends a data writeback message. With DirectoryCMP, a CMP sends a request message, receives a data message, sends an unblock message (used to reduce the implementation complexity), requests a writeback, gets a writeback grant, and sends a data writeback message. A total of 168 bytes for TokenCMP and 176 bytes for DirectoryCMP.

In a system with more CMPs, TokenCMP traffic results will be worse (unless multicast with destination set predictions is employed [24]). Our directory protocol also expends messages to increase performance and manage complexity whereas other implementations may choose different tradeoffs. Regardless, the current TokenCMP has reasonable traffic characteristics for modest numbers of CMPs per system.

Intra-CMP Bandwidth. Intra-CMP traffic also generates little queuing delay for our assumptions. However we plot intra-CMP traffic in Figure 7b and break it down by message type. To first order, all protocols use similar intra-CMP bandwidth. As expected, TokenCMP protocols expend more traffic for request messages (both internal and external) due to broadcast. Unexpectedly, DirectoryCMP uses more traffic for response data because of an artifact of the strictly hier-

[1] X% faster = runtime(DirCMP)/runtime(TokenCMP) - 1

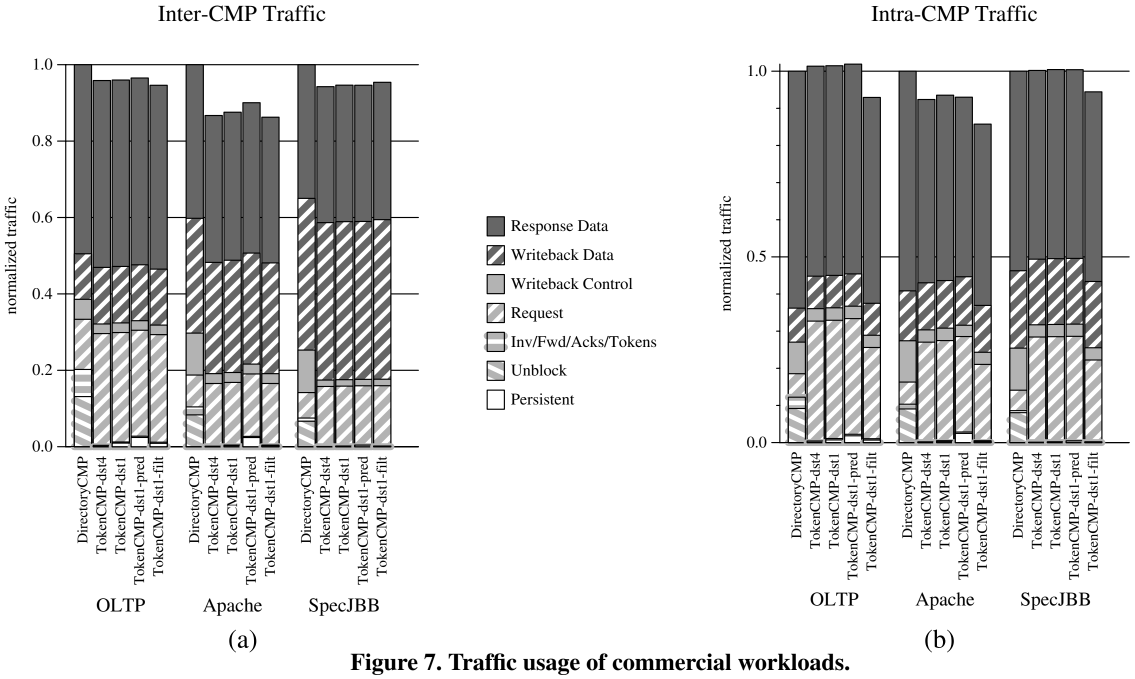

Figure 7. Traffic usage of commercial workloads.

archical DirectoryCMP implementation: data responses must be handled by the L2 cache (intra-CMP directory). For example, an L1 cache responding to a forwarded request from an external CMP, must transfer the data to the L2 cache where it may collect other invalidation acknowledgments. Only then does the L2 send the data to the requesting chip which in turn sends the data to the requesting processor. In contrast, in TokenCMP an L1 cache directly sends the forwarded request to the requesting processor, using a single data message on the on-chip interconnect.

As introduced in Section 4, TokenCMP-dst1-filt uses an approximate directory of L1 sharers to filter external transient requests. Figure 7b shows that the filter reduces intra-CMP bandwidth 6-8%, but the utilization is sufficiently low that this does not affect runtime.

9 Conclusions

Few papers have considered implementing coherence in systems with multiple chip multiprocessors (M-CMPs). Those that do implement hierarchical protocols (e.g., [5, 36]). This paper advocates using token coherence to obtain TokenCMP protocols that have flat correctness properties, but exhibit hierarchical performance characteristics. We found TokenCMP variants easier to verify than hierarchical directory protocols. We improved token coherence performance robustness under high-contention. Finally, we showed commercial workloads can run significantly faster on M-CMP systems using TokenCMP variants instead of a hierarchical directory protocol.

Acknowledgments

We thank Virtutech AB, the Wisconsin Condor group, and the Wisconsin Computer Systems Lab for their help and support. We thank Brad Beckmann, Kevin Moore, the Wisconsin Multifacet group, and the Wisconsin Computer Architecture Affiliates for their comments on this work.

This work is supported in part by the National Science Foundation (CCR-0324878, EIA/CNS-0205286, and CCR-0105721) and donations from Intel Corporation and Sun Microsystems. Hu is supported in part by a research grant from the Natural Sciences and Engineering Research Council of Canada, and Bingham is supported by a UBC Graduate Fellowship. Hill and Wood have a significant financial interest in Sun Microsystems.

References

[1] A. R. Alameldeen, M. M. K. Martin, C. J. Mauer, K. E. Moore, M. Xu, D. J. Sorin, M. D. Hill, and D. A. Wood. Simulating a $2M Commercial Server on a $2K PC. *IEEE Computer*, 36(2):50–57, Feb. 2003.

[2] A. R. Alameldeen and D. A. Wood. Variability in Architectural Simulations of Multi-threaded Workloads. In *Proceedings of the Ninth IEEE Symposium on High-Performance Computer Architecture*, pages 7–18, Feb. 2003.

[3] L. A. Barroso and K. Gharachorloo. Personal Communication, June 2003.

[4] L. A. Barroso, K. Gharachorloo, and E. Bugnion. Memory System Characterization of Commercial Workloads. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 3–14, June 1998.

[5] L. A. 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 *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 282–293, June 2000.

[6] L. A. Barroso, K. Gharachorloo, M. Ravishankar, and R. Stets. Managing Complexity in the Piranha Server-Class Processor Design. In *2nd Workshop on Complexity-Effective Design held in conjunction with the 27th International Symposium on Computer Architecture*, June 2001.

[7] J. M. Borkenhagen, R. D. Hoover, and K. M. Valk. EXA Cache/Scalability Controllers. In *IBM Enterprise X-Architecture Technology: Reaching the Summit*, pages 37–50. International Business Machines, 2002.

[8] T. Braun, A. E. Condon, A. J. Hu, K. S. Juse, M. Laza, M. Leslie, and R. Sharma. Proving Sequential Consistency by Model Checking. In *International High-Level Design, Validation, and Test Workshop*. IEEE, Nov. 2001.

[9] E. Clarke and E. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of the Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, May 1981. Springer-Verlag.

[10] A. L. Cox and R. J. Fowler. Adaptive Cache Coherency for Detecting Migratory Shared Data. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 98–108, May 1993.

[11] D. E. Culler and J. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann Publishers, Inc., 1999.

[12] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol Verification as a Hardware Design Aid. In *International Conference on Computer Design*. IEEE, Oct. 1992.

[13] K. Gharachorloo, M. Sharma, S. Steely, and S. V. Doren. Architecture and Design of AlphaServer GS320. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 13–24, Nov. 2000.

[14] E. Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the Fifth IEEE Symposium on High-Performance Computer Architecture*, pages 172–181, Jan. 1999.

[15] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The Stanford Hydra CMP. *IEEE Micro*, 20(2):71–84, March-April 2000.

[16] L. Hammond, B. A. Nayfeh, and K. Olukotun. A Single-Chip Multiprocessor. *IEEE Computer*, 30(9):79–85, Sept. 1997.

[17] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking Cache-Coherence Protocols with TLA+. *Formal Methods in System Design*, 22(2):125–131, March 2003.

[18] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addision-Wesley, 2002.

[19] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 148–159, May 1990.

[20] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–35, Oct. 1994.

[21] T. D. Lovett and R. M. Clapp. STiNG: A CC-NUMA Computer System for the Commercial Marketplace. In *Proceedings of the 23th Annual International Symposium on Computer Architecture*, May 1996.

[22] P. S. Magnusson et al. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, Feb. 2002.

[23] M. M. K. Martin. *Token Coherence*. PhD thesis, University of Wisconsin, 2003.

[24] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 206–217, June 2003.

[25] M. M. K. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[26] K. L. McMillan and J. Schwalbe. Formal Verification of the Gigamax Cache-Consistency Protocol. In *International Symposium on Shared Memory Multiprocessing*, pages 242–251. Information Processing Society of Japan, 1991.

[27] D. Mosberger. Memory Consistency Models. *Operating Systems Review*, 27(1):18–26, 1993.

[28] A. Moshovos, G. Memik, B. Falsafi, and A. Choudhary. JETTY: Filtering Snoops for Reduced Power Consumption in SMP Servers. In *Proceedings of the Seventh IEEE Symposium on High-Performance Computer Architecture*, Jan. 2001.

[29] J.-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in Cesar. In *5th International Symposium on Programming*, pages 337–351. Springer, 1981. Lecture Notes in Computer Science Number 137.

[30] Z. Radovi'c and E. Hagersten. Efficient Synchronization for Nonuniform Communication Architectures. In *Proceedings of SC2003*, Nov. 2003.

[31] R. Rajwar, A. Kägi, and J. R. Goodman. Improving the Throughput of Synchronization by Insertion of Delays. In *Proceedings of the Sixth IEEE Symposium on High-Performance Computer Architecture*, pages 168–179, Jan. 2000.

[32] S. L. Scott and J. R. Goodman. Performance of Pruning-Cache Directories for Large-Scale Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 4(5):520–534, May 1993.

[33] D. J. Sorin, M. Plakal, M. D. Hill, A. E. Condon, M. M. K. Martin, and D. A. Wood. Specifying and Verifying a Broadcast and a Multicast Snooping Cache Coherence Protocol. *IEEE Transactions on Parallel and Distributed Systems*, 13(6):556–578, June 2002.

[34] P. Stenström, M. Brorsson, and L. Sandberg. Adaptive Cache Coherence Protocol Optimized for Migratory Sharing. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 109–118, May 1993.

[35] D. M. Taub. Improved Control Acquisition Scheme for the IEEE 896 Futurebus. *IEEE Micro*, 7(3), June 1987.

[36] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 System Microarchitecture. *IBM Journal of Research and Development*, 46(1), 2002.

[37] M. Tremblay, J. Chan, S. Chaudhry, A. W. Conigliaro, and S. S. Tse. The MAJC Architecture: A Synthesis of Parallelism and Scalability. *IEEE Micro*, 20(6):12–25, Nov-Dec 2000.

[38] G. White and P. Vogt. Profusion: A Buffered, Cache Coherent Crossbar Switch. In *Proceedings of the 5th Hot Interconnects Symposium*, pages 87–96, Aug. 1997.

[39] P. Wolper. Expressing Interesting Properties of Programs in Propositional Temporal Logic. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 184–192, Jan. 1986.

[40] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In L. Pierre and T. Kropf, editors, *Proceedings of Correct Hardware Design and Verification Methods (CHARME '99)*, number 1703 in LNCS, pages 54–66. Springer-Verlag, 1999.