Post-Silicon Verification for Cache Coherence

Andrew DeOrio, Adam Bauserman and Valeria Bertacco
Dept. of Electrical Engineering and Computer Science
University of Michigan, Ann Arbor
{*awdeorio, adambb, valeria*}*@umich.edu*

***Abstract*— Modern processor designs are extremely complex and difficult to validate during development, causing a growing portion of the verification effort to shift to post-silicon, after the first few hardware prototypes become available. Extremely slow simulation speeds during pre-silicon verification result in functional errors escaping into silicon, a problem that is further exacerbated by the growing complexity of the memory subsystem in multi-core platforms. In this work we present CoSMa, a novel technology offering high coverage functional post-silicon validation of cache coherence protocols in multi-core systems. It enables the detection and diagnosis of functional errors in the memory subsystem by recording at runtime a compact encoding of the operations occurring at each cache line and checking their correctness at regular intervals. We leverage the system's existing memory resources to store the required activity, thus minimizing area overhead. When the system is finally ready for customer shipment, CoSMa can be completely disabled, eliminating any performance or memory overhead. We reproduce in our experiments a set of coherence protocol bugs based on published errata documents of commercial multi-core designs, and show that CoSMa is highly effective in detecting them.**

I. Introduction

As chip multi-processor (CMP) systems grow more common, errors in cache coherence have become leading sources of bugs found in the post-silicon debugging phase. Errata from multi-core designs, such as the Core 2 Duo [1], indicate that at least 10% of escaped bugs were due to such errors. Increasing design complexity limits the design space that can be effectively verified before tapeout, requiring more effort to be invested in the post-silicon phase. As a result, post-silicon debugging has been a growing component of a chip's time-to-market, up to 35% at the 90nm technology node [2]. In this landscape, verification engineers are faced with the challenge of identifying errors, determining their root causes, and then correcting them. Finding the source of an error is especially difficult in post-silicon validation due to the lack of observability of internal circuit nodes which, in contrast, is widely available when debugging in pre-silicon simulation-based methodologies.

This situation is further exacerbated by modern multi-core processor systems, where multiple processor nodes share one or more memory blocks through an interconnect network. The communication between these nodes and the central memory is regulated by cache coherence and consistency protocols. While cache coherence protocols may appear simple at the abstract level, described by a simple finite state machine (FSM), the implementation of a protocol in hardware is complex, requiring many additional intermediate states to be taken into account. For example, a conceptual 5-state MOESI protocol is implemented in the GEMS simulation framework [3] with a 15-state FSM in the level 1 (L1) cache controllers and a 60-state FSM in the level 2 (L2) controller. The problem is worsened by the complex interactions that may occur when several of these controllers are instantiated in the same system. Due to the complexity of even basic cache coherence models and the slow simulation speed of pre-silicon verification, large portions of the design space go unverified at the pre-silicon stage and an increasing number of functional errors escape to silicon. The focus of our work is to leverage early silicon to speed up the verification process, thereby providing the level of coverage necessary to ensure the correctness of coherence protocols in multi-core systems.

A. Our Contribution

CoSMa is a novel solution that focuses on the post-silicon verification of cache coherence protocols for multi-core designs, enabling high coverage verification while incurring a near-zero area impact. The novelty offered by our contribution is a distributed cache activity history storage mechanism, and an accompanying online validation algorithm that determines the correctness of observed events. The history storage mechanism monitors the shared memory operations at each local cache during regular system operation, logging activity in a compact form for a period of time. Then, at regular intervals, the systems stops normal execution and enters a special interrupt mode, during which we rely on the cores to check that shared memory management mechanisms have maintained correct coherence. Memory coherence is a property of memory management which ensures that any access to a given location returns the latest value to that location. Memory coherence checks are achieved through a variant of a string matching algorithm (hence the name CoSMa, Coherence String Matching). To store the activity history, we temporarily reallocate some of the system's native resources (typically half of the L1 and L2 caches) to be controlled by CoSMa, allowing us to keep the area overhead at a bare minimum (Figure 1). These resources store a footprint of the coherence protocol activity, and possibly processor and timing information. CoSMa offers two variants of this mechanism, one optimized for minimal performance overhead, the other for high coverage. Furthermore, we provide the ability to completely disable CoSMa, so that when the system is ready for production, all cache storage is released and made available for mainstream computation.

Our experimental results show that CoSMa is effective in detecting bugs of the memory subsystem in multi-core designs, and that the performance overhead when CoSMa is active ranges from 1% to 23%. Moreover, when CoSMa is disabled there is no performance impact on the system. From an area cost standpoint, CoSMa requires the addition of a few

hardware logging mechanisms, devised to require minimal design effort and to be mostly decoupled from the underlying system. In our evaluation on an OpenSPARC T1 processor, these checkers would constitute only a small fraction of a percent of the overall design area.

II. Background and Related Work

The cache coherence protocol in a multi-processor system defines the behavior of read and write operations to and from individual processors and a shared memory subsystem, so that each processor can access the latest value written to a memory location. Different protocols usually define the states in which an individual cache line can be at any point in time. Much research has been dedicated to the correctness of cache coherence protocols, ranging from abstract model analysis, to simulation techniques, to runtime solutions.

At an abstract level, the state of a local cache line can be described by a relatively simple FSM; verifying the correctness of the abstract model is a well-explored area of research [4, 5, 6, 7, 8]. The implementation of the model into a concrete system, on the other hand, can be problematic, requiring many additional intermediate states. As a result, the representation of the corresponding FSM for the whole system quickly becomes impractical, since it corresponds to the product of all the local FSMs. Pre-silicon verification of the implementation-level is undertaken in [9], where the authors use a constrained-random testing approach to increase coverage of a simulated design. The slow simulation speed and large state space precludes this approach from covering many of the possible cases, thus significant portions of the design go unverified.

Runtime solutions have emerged in response to the inadequacies of pre-silicon verification. Cantin *et al.* [10] propose the addition of a simple cache controller to each processor, as well as an extra communication bus connecting the controllers. This supplementary system runs in parallel with the existing cache coherence system, verifying its operation. Another recent solution [11, 12] uses signatures to represent the state of the cache lines, periodically aggregating the data to check that a number of global invariants are satisfied. These solutions have the advantage of tracking faults through the entire lifespan of a system at a minimal performance degradation, but at the expense of conspicuous hardware overhead. In the domain of post-silicon validation, solutions must have extremely limited hardware impact, so as not to affect the cost and power envelope of the system after customer release. On the other hand, the performance impact can be more noticeable, as long as it can be completely eliminated after product release.

The novelty of our solution lies in a high coverage, near-zero area impact solution which exploits the high performance of hardware prototypes to overcome the limitations of pre-silicon validation. Compared to runtime solutions, CoSMa has much lower area overhead (orders of magnitude) and can detect functional design errors.

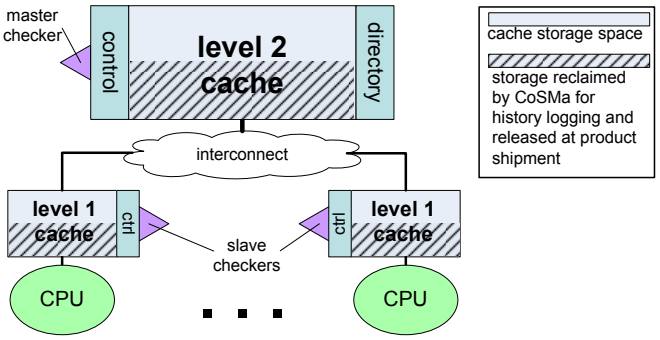

Fig. 1. **CoSMa is a distributed solution** with small checkers residing on each cache controller, the master on the L2 and a slave on each L1. This configuration is agnostic to the interconnect network type, and relies on it for communication between master and slave checkers. Storage for the coherence history is temporarily allocated only during verification and is shown with hashed lines.

III. CoSMa components

In a CoSMa design, a multi-core processor is augmented with a special mode of operation, called CoSMa mode, to be activated only during post-silicon validation. While in CoSMa mode, time is organized into epochs: each epoch includes a phase of normal execution, while CoSMa logs activity in the background, followed by a checking phase, during which coherence is checked (Figure 2) between local caches and the central memory. For simplicity, from now on we make the assumption that only the L1 caches are local and that the L2 cache is centralized and used to enforce coherence. However, CoSMa could be also be deployed in situations where coherence is enforced at other levels of the memory hierarchy.

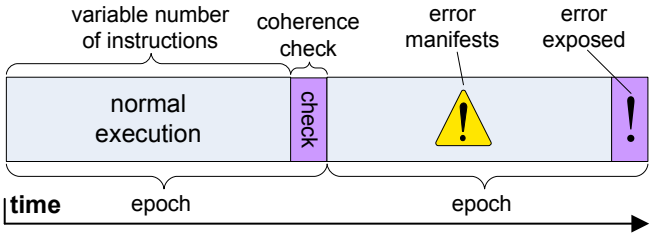

Fig. 2. **Checks are performed periodically**; the system executes normally until log storage resources have been exhausted, at which point cores are stalled and the checking phase begins.

At the beginning of the checking phase, all cores are stalled and pending memory requests are completed. Then the **master checker**, situated alongside the L2 controller (Figure 1), broadcasts the coherence history of each valid line over the existing interconnect network. **Slave checkers** at each L1 cache look up the address of the broadcasted line and verify that their local history is compatible with the global history by executing a checking algorithm on the local core. If the check passes, the local coherence history in the L1 cache is cleared; otherwise, the system stops so that the fault can be debugged. After all valid lines in the L2 have been broadcasted, the L1 slave checkers examine their own caches, ensuring that all valid lines have been checked. Finally, the master checker synchronizes the slaves and normal operation resumes.

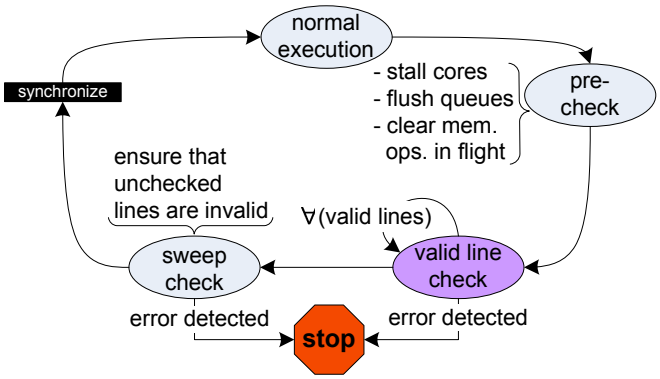

Fig. 3. **A slave checker** is connected to each L1 cache, and operates in two phases: first checking valid lines against messages from the master checker and then ensuring that no valid line has been missed in the check.

A. Operation

During each epoch, **normal execution** (Figure 3) extends until history storage has been exhausted. When the checking phase begins, first, a **pre-check** stage allows buffers and queues to empty and completes outstanding memory operations, ensuring consistent L1 and L2 cache states. Checking is organized into two phases: a valid line check that matches valid L2 lines against their L1 counterparts, and a sweep check, where each slave ensures that all of its valid lines have been checked during the previous phase. During a **valid line check**, the master checker broadcasts a message for each valid line in the L2. Each slave checker does a lookup in its L1 upon receiving each message and checks its contents against the message. This check ensures that the coherence history in the L1 cache is compatible with the history from the L2. The cache line is marked as checked and the next lookup takes place. When all the valid lines in the L2 have been exhausted, the system proceeds to **sweep check**: each slave checks that any line not checked in the previous phase is marked invalid. Assuming an inclusive L2 cache, all lines in the L1 cache must also reside in the L2 cache. Thus, if the sweep check discovers an unchecked and valid line, CoSMa determines that it is an errant line. If an error is detected at any time during the check, an exception is raised and debug information is readily available for the verification team in the log storage. Finally, the **synchronize** stage, synchronizes the slaves, waiting for all to complete. At this point, the next epoch begins.

IV. Checking Algorithms

CoSMa is equipped with two variants of the checking algorithm responsible for reconciling coherence operations. They are always executed in-situ on the processor cores themselves and operate on corresponding L1 and L2 cache lines. The first algorithm (called Low-overhead) is optimized for low overhead and minimal perturbation to the system under test; its corresponding histories are compact, encoded sequences of cache coherence states. However, this low overhead algorithm may have an impact on coverage. The second algorithm targets high coverage and has a higher performance impact. Here, cache line histories also include timing and processor ID information. Note that both algorithms are free of false positives.

The algorithms operate on two cache line histories corresponding to the address under consideration, one from the L2 and the other from the local L1. A history is a sequence of coherence states representing changes to the cache line over a period of time, also including timestamp and processor ID information in the case of the high coverage algorithm. By comparing them, CoSMa's algorithm determines whether they are compatible with coherence shared memory management; incompatible histories indicate a bug.

The check procedures analyze two histories corresponding to the same cache line, the global history from the L2 and the local history from the L1. These two state histories are **compatible** if there exists at least one valid sequence of memory operations that could have generated both histories. The basis of our algorithms is that in a correctly functioning cache coherence scheme, the state of each valid L1 line must agree with the corresponding L2 line. Below, we discuss the algorithms in detail using the `MESI` protocol as a running example. With minor modifications, the algorithms can be applied to any other protocol. The `MESI` states are: `M` indicating a dirty (modified) version of the data is present, `E` representing exclusive access, *i.e.*, the line is clean and not shared by any other L1, `S` meaning the line is clean but shared between two or more L1 caches, and `I` indicating that the line is invalid or not present in the cache.

A. Low-overhead Algorithm

The low-overhead algorithm is optimized for maximal runtime between checks and minimal system perturbation. It is capable of determining compatibility with a minimal amount of information, as it relies only on the observed sequence of high-level `MESI` states, (`M, E, S, I`). The algorithm is divided into three sections: sequence compression, partitioning and matching.

1. Sequence Compression. Repeated sequences of states `M` or `E` alternating with the Invalid state are reduced to one occurrence of `M` or `E`. For example, the sequences `MIM`, `MIMIM`, *etc.*, would all reduce to `M`. This accounts for the case where an L2 line remains unmodified while different L1 caches contend for it. Note that it would be possible to perform this step on the fly during normal execution.

2. Partitioning accounts for the observation that activity in an L1 cache will be reflected in the L2 history, but may not always appear in other L1 histories. Specifically, invalid states seen in the L1 history may correspond to any type of activity in the L2: while one L1 cache is invalid for a particular address, another L1 may be operating on that data. On the other hand, each sequence of non-invalid L1 states must have a corresponding segment somewhere in the L2 history. Thus, we partition the L1 history at each occurrence of an `I` state.

3. Matching between the partitioned L1 history and the L2 history can now take place. Each sequence partition must be found within the L2 history, with all partitions matching in the same order as they appear in the L1 history. As soon

as a match for each partition is found, the algorithm returns successfully; failure to find a match indicates an error in the cache coherence protocol.

Example. Consider two cache line histories, one corresponding to the L1 and the other to the L2.

L2 history:	IMISMIM	initial
L1 history:	IMIMIMISMI	initial
L1 history:	IMISMI	after sequence compression
L1 history:	| M | SM |	after partitioning (|)
L2 history:	I ***M*** I ***SM*** I M	match found, matched
L1 history:	| ***M*** | ***SM*** |	partitions are in italics

Initially, these two histories reflect a period of contention among L1 caches for write access to the address, reflected by the sequence of MIMIM states in the L1 history. The sequence compression step reduces this sequence to M, which allows for the match against the single M state in the L2 cache. This is followed by shared reads before going into another period of write access, reflected by the SM section of the L1 and L2 histories. The L1 history is partitioned, (shown with "|"), and then matched. A match between the L1 and L2 histories is found, thus no bug is flagged in this example.

B. High Coverage Algorithm

Designed to maximize coverage, this algorithm is capable of detecting any errant activity related to a cache line's coherence state. When the coverage algorithm is in operation, the history storage mechanism logs timestamp and processor ID information in addition to the MESI state. This information is in the form: <cycles in intermediate state, processor ID, MESI state, cycles in MESI state>. Timestamps are maintained by a counter in each cache controller, which are periodically synchronized. The coverage algorithm is significantly simpler than the low-overhead algorithm since it has concrete timestamps to work with. This algorithm processes a pair of L1 and L2 histories linearly, aligning the two histories by their timestamps. Recorded MESI states in the L2 history that match the processor ID of the current L1 history are then matched; M, E and S states must exhibit a one-to-one correspondence between the two histories. An I in the L2 history must match an I in the L1 history, but not the reverse. When the L1 history is in the invalid state, the L2 history is simply checked to ensure that no recorded states within the time period are marked with the current processor ID.

C. Implementation and Complexity

CoSMa's history checking algorithms take two coherence histories as input, one from an L1 cache and the other from the L2 cache. The coverage algorithm processes each history linearly, while the low-overhead algorithm performs its check recursively. When the histories have been checked for compatibility, both algorithms return a judgment free of false positives.

Coverage Algorithm. The coverage algorithm aligns L1 an L2 histories using timestamps. Comparisons are made through matching processor IDs. Since the histories are traversed once from beginning to end, the complexity of the coverage algorithm is linear with the number of history entries.

Low-overhead Algorithm. The low-overhead algorithm considers a subset of an L1 and an L2 history, attempting to match the first partition found in the L1 against the remaining portion of the L2 history. When the algorithm is called with only one L1 history partition, it reduces to simple substring matching: the sequence must be found within the L2 history in order for them to be compatible. If no matches are found, CoSMa has found a coherence error. The worst-case complexity of the proposed algorithm is dependent on the characteristics of its inputs. To describe the inputs of the algorithm, let us define the following parameters: d is the length of the L2 history, p is the number of partitions in the L1 history, and b is a constant expressing the amount of activity within the L2 history (the average number of states traversed in the L2 history before finding each L1 history partition). To estimate complexity, we construct a tree to model the execution of the algorithm. The branching factor, *i.e.*, the average number of recursive calls made at each step, is $\frac{d}{b}$. The search depth, or the maximum depth of the recursion tree, is p. The worst-case number of recursive calls can then be approximated by the number of nodes in this tree, given by the expression $\left(\frac{d}{b}\right)^{p+1}$. This complexity is the worst-case bound and can only occur when the algorithm detects a bug, that is, when the entire tree must be traversed to conclude that no valid match exists. We note that the algorithm's complexity is exponential, growing particularly fast with the number of partitions in the L1 history. As a result, we must be careful to handle those rare worst-case situations that could have a significant impact on overall performance. Worst-case complexity in CoSMa is managed by performing a runtime estimate before calling the compatibility algorithm. If it exceeds a certain threshold, the corresponding cache line goes unchecked. In practice, we found that the performance of the algorithm is always much better than its theoretical bound, leading to an overall solution where history checks have very little impact on performance overhead.

D. Strengths and Limitations

CoSMa's checking algorithms allow the trade off of coverage and performance overheads. The high coverage algorithm and accompanying history storage mechanism are capable of catching all coherence errors encountered by a workload that result in incompatible MESI states. Even bugs that result in subtle timing variations can be caught by this method, since timing information is retained in the history.

On the other hand, the low-overhead algorithm is capable of running for much longer before a check is initiated, decreasing perturbation of the system under test. Because it does not retain timing information, this history storage mechanism is not guaranteed to catch every possible coherence failure. In practice, many cache coherence bugs will manifest as an incorrect sequence of cache line states, and therefore can be caught. For example, errors in cache controller logic may cause a line to transition into an incorrect state. Worse yet, messages on the interconnect

may be lost or corrupted, preventing a request or response from reaching its destination. Silent corruption of cache state bits may also occur, unexpectedly changing state even when a line is not accessed. However, the low-overhead algorithm cannot catch bugs that result only in subtle timing variations because it does not store timing information. For example, consider a RAW (read-after-write) scenario in a CMP system. A faulty L1 cache controller could exhibit a delayed response to an invalidate request, allowing stale data to be read by the processor when the line should have been invalidated. The cache line eventually transitions to the invalid state, so a correct history is present when the next check is performed. Despite its evasion of the low-overhead algorithm, this example is caught by the coverage algorithm.

While CoSMa is effective in detecting many types of coherence errors, some may still escape. Data corruption is one class of errors not covered, as this is often addressed through checksums or error correction codes. Additionally, the system is assumed to make forward progress; deadlocks or system hangs are not detected. Another type of bug that cannot be detected with CoSMa is one where an L1 cache controller ignores the current state of a cache line, illegally services a processor's request, and silently fails to change `MESI` state. This would preclude the history from being logged, thus when the check is performed, no mismatch would be detected. The class of bugs that results in performance degradation and yet provides correct results also dodges CoSMa's detection mechanism: incorrect behavior causing communication saturation or resource starvation while maintaining correct cache coherence states will not cause a mismatch.

V. System integration

We integrated CoSMa in a CMP system using `MESI` by, adding a master checker to the L2 cache controller and slave checkers to each L1 controller. CoSMa is designed to be integrated with minimal design effort, as we discuss in Section VI-E. Communication between master and slave checkers is achieved through the existing interconnect infrastructure. The area overhead for our solution is reported in Section VI-D, showing that reuse of existing hardware resources allows us to maintain a near-zero impact.

L2 Cache. The master checker is placed next to the shared L2 cache. Half of the cache lines are used as a coherence history buffer, thus any active line in the first half of the cache will use the corresponding line in the second half as a memory spool to hold coherence state changes. Note that different partitions are also possible. Each `MESI` state is encoded using two bits so, for the low-overhead algorithm, at most 256 history items can be stored in a typical 64-byte cache line. For the coverage algorithm, each history item occupies approximately 4 – 6 bytes (depending on workload); thus, a typical line can store 10 – 15 items. In this algorithm, 8 bits are allocated to the processor ID and 10 bits to the time spent in an intermediate state. In our experience, we found that a 10 bit (1024 cycle) timestamp was adequate 99% of the time. For the time spent in a `MESI` state, 14 bits were necessary to achieve the same accuracy for random stimulus tests, and 29 bits were needed for SPLASH2 benchmarks. When overflow occurs (1% of the time), a duplicate history item is appended and the counter restarted. To write into the history buffer, we augment the logic in the L2 so that history is logged in parallel with data; CoSMa leverages banked caches to avoid adding extra ports to the cache memory. A pointer stored in the cache tag register indicates where the next history entry should be stored. Finally, when any history line exceeds a specified maximum number of history items, a check is initiated.

L1 Caches. Each L1 cache controller is augmented with a slave checker and half of the cache storage is used for history logging. History items are appended in the same fashion as for the L2. In order to deal with evicted cache lines, CoSMa can be configured to perform individual cache line checks before any eviction. CoSMa can also be configured to allow evictions without checks, enabling decreased performance impact and decreased perturbation to the system. In this case, the L1 checker will clear the history when a line is replaced due to eviction, but not when a line is invalidated due to external invalidate requests, *i.e.*, when another processor has written newer data to the same cache line. As a result, history can be maintained even across periods when a cache line state is invalid.

Processor cores. Each processor core is augmented with a simple stall mechanism to prevent any new memory requests from being initiated while a coherence check is in progress. This requires minimal modifications, since the core is already required to stall under a number of conditions (*e.g.*, when waiting for a memory operation). Additionally, to minimize hardware overhead, the coherence history checking algorithm is executed on the existing CPU cores.

VI. Experimental Evaluation

To evaluate the performance and correctness of CoSMa, we implemented it using the GEMS simulation framework [3]. The core of the GEMS tool set is Ruby, a memory system simulator that can model a variety of coherence protocols connected through a network topology. We modeled a chip multi-processor system as described in Table I.

Processors	4-core CMP
Interconnect	Point-to-point switched (unordered)
L1 caches	1KB per core, write-back, 4-way
L2 cache	8KB shared, banked, inclusive
Line size	64 B
Memory size	4 GB
Coherence	Directory, MOESI protocol
L1/L2 hit latency	1 cycle / 18 cycles
Mem latency	92 cycles

TABLE I. **System model parameters** used to evaluate CoSMa.

For our coherence protocol, we chose the MOESI directory protocol. Note that MOESI can be treated exactly the same as `MESI` by our algorithm, since the Owner state is simply a variant of Shared: thus, when the history is being stored, an `S` is stored in place of an `O`. Other coherence protocols can also

be handled by using a minimal number of states to represent the history. Because of the limited speed of GEMS, cache sizes were selected to be relatively small to maximize stress on the coherence mechanism. Additionally, we employed SimpleScalar [13] to obtain a cycle-accurate estimation of the runtime overhead due to CPU time spent executing the checking algorithm. The number of history items we could store was dictated by the cache line size, 64B in our case. This allowed us to fit 256 history items for the low-overhead algorithm and 16 items for the high coverage algorithm.

A. Workloads

We evaluated the performance impact of CoSMa on two types of workloads: directed random stimulus and SPLASH2 [14] benchmarks. Three directed random testers were generated using Ruby's built-in microbenchmarks with the intent of maximizing sharing and stress on the cache coherence mechanism. The random-128k, random-256k and random-512k workloads are based on the comparison and swapping of atomic operations, similar to the behavior exhibited by software locks. Each of these workloads was configured to use different cache sizes, indicated by the name.

We also ran our experiments with a set of 10 SPLASH2 benchmarks, which helped us in characterizing the performance impact of CoSMa under more typical conditions. Since CoSMa affects only the performance of the memory system, we simulated the memory accesses corresponding to 10 million instructions on each core (a total of 40 million instructions were executed) in the heart of the benchmark. We generated a trace of these memory operations for each benchmark using the Simics simulator [15], then executed the trace using our Ruby and SimpleScalar setup.

B. Performance Results

The performance overhead observed when CoSMa is active ranged from less than 1% to 23%, averaging to 7% for the SPLASH2 benchmarks and to 10% for the random workloads. Figure 4 plots the overheads of running CoSMa. We observe that the low-overhead algorithm consistently resulted in a lower runtime impact than the coverage algorithm.

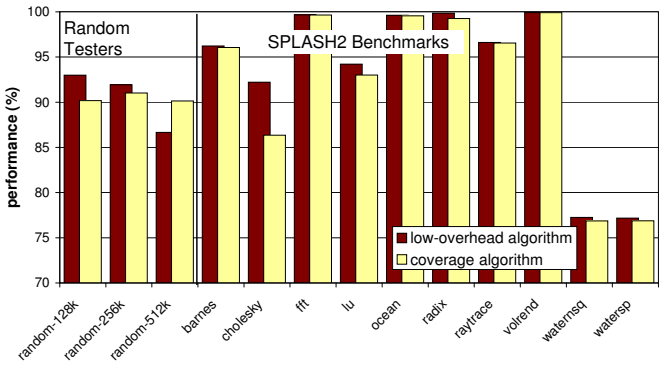

Fig. 4. Performance impact of CoSMa mode. 100% indicates no performance degradation.

This performance overhead has two major contributors: network delay to transfer the history logs from the L2 to the L1, and CPU time to execute the matching algorithm.

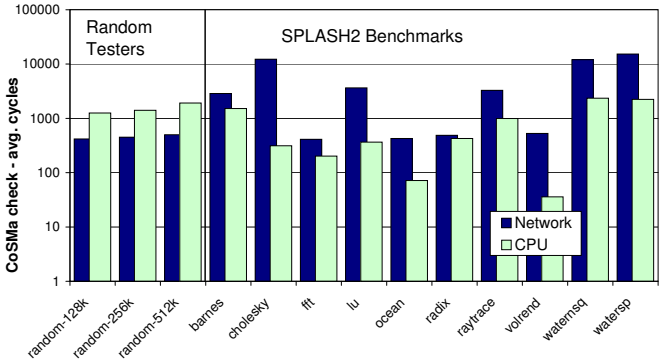

Fig. 5. Low-overhead algorithm: network and CPU overhead. Overall performance impact is determined by the greater of the two.

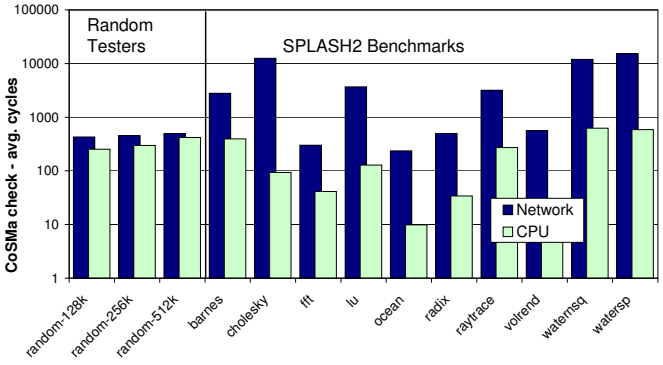

Fig. 6. Coverage algorithm: network and CPU overhead. Overall performance impact is determined by the greater of the two.

Network delivery and computation are pipelined; as the computation for one set of histories progresses, a new set is in flight on the network. Moreover, all processors execute the matching algorithm in parallel, further streamlining the process. Figures 5 and 6 show the average latency due to network and CPU computation for one check period. Since these occur in parallel, in each case the higher of the two bars represents the total performance impact. Both coverage and low-overhead algorithms are evaluated, and we found that the cycles required to transfer histories through the network was the dominant factor most of the time. All SPLASH2 benchmarks were network bounded, this was due to their large working sets: more histories had to be sent across the network for a check. We observed CPU-dominated performance impact when running the low-overhead algorithm on random stimulus workloads. In this case, many long complex histories are generated due to high contention for a concentrated set of addresses, leading to many partitions. A key insight gained from these results is that the exponential complexity of the low-overhead algorithm rarely affects the runtime of the system. In fact, in our experiments, we limited the runtime of the low-overhead algorithm to 10,000 cycles to avoid an exponential escalation for those cases close to the worst-case complexity bound. However, we found experimentally that none of the testbenches reached this.

We also investigated the relation between performance overhead and storage space (see Figures 7 and 8) for varying proportions of CoSMa vs. computation storage. With more history storage, the system can leverage longer epochs, and

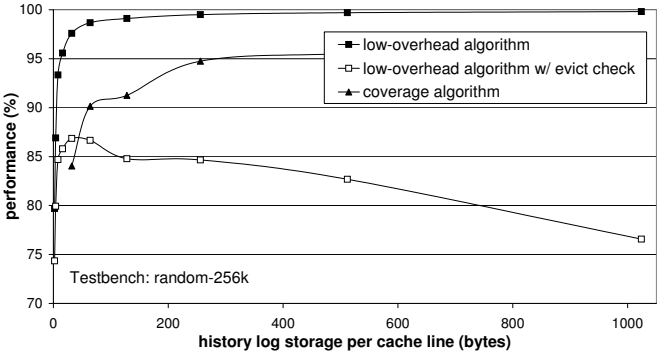

Fig. 7. **Performance as a function of history log size - Random stimulus.** Both the low-overhead and coverage algorithms are shown. A variant of the low overhead algorithm which performs a check at each eviction is also shown.

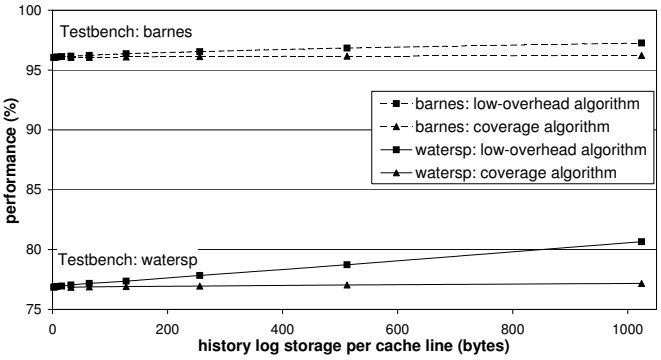

Fig. 8. **Performance as a function of history log size - SPLASH2.** We plot the results for two benchmarks at the extremes of observed performance.

hence, as we validate experimentally, more history storage space leads to lower performance overhead. An exception to this is the variant of the low-overhead algorithm where a cache line is checked upon each eviction (Figure 7). In this case, more storage caused a performance decrease due to the large number of eviction checks. Overall, the low-overhead algorithm leads to longer epochs and better performance.

C. Exposing Bugs

We evaluated CoSMa's ability to find cache coherence bugs using a set a injected functional bugs in our simulation platform. seven bugs, inspired by published escaped bug reports of the Intel Core 2 Duo [1] (see Table II), were injected into the Ruby cache controllers.

concurrent-writes	Concurrent writes to the same non-dirty cache line causes dropped messages.
write-unaligned	Writing two bytes across a cache line boundary causes message misordering.
dropped-message	Messages are randomly dropped.
delayed-message	Messages are randomly delayed.
two-stores	Two consecutive stores to different addresses cause the first to be delayed.
stores-dropped	Two stores cause one to be dropped.
delayed-writes	Some messages for stores are delayed

TABLE II. **Cache coherence bugs injected** in our CMP model.

Each bug was injected individually and the Ruby tester was run with a variety of workloads, including a random tester and the SPLASH2 benchmarks. When an error was detected by the coherence checker, the simulation was terminated. Each of the bugs was eventually detected by CoSMa. In Table III we show the number of cycles required to expose them on the random tester with both the low-overhead and coverage algorithms. On average, the low-overhead algorithm required approximately 1 million cycles to discover injected bugs, while the coverage algorithm took approximately 100,000 cycles. Table IV shows similar information for the SPLASH2 benchmarks. The coverage algorithm was able to find all 7 bugs with all workloads, while the low-overhead algorithm was unable to expose 2 of them. The bugs evading the low-overhead algorithm were delayed-message and delayed-writes, both the result of subtle timing perturbations.

Bug	Low-overhead Algorithm	Coverage Algorithm
concurrent-writes	1,823k	48k
write-unaligned	1,047k	48k
dropped-message	778k	213k
delayed-message	not detected	46k
two-stores	1,682k	355k
stores-dropped	780k	124k
delayed-writes	not detected	46k

TABLE III. **Execution cycles required to expose bugs with random stimulus**. The coverage algorithm could expose all bugs in about 126k cycles on average. Low-overhead could not detect two of the bugs.

When SPLASH2 benchmarks were run on systems including cache coherence bugs, we noticed that a significantly higher number of cycles was required compared to the random tests. As with the random tester, the coverage algorithm was able to discover every bug, while the low-overhead algorithm was unable to expose the two bugs involving delayed messages. It is interesting to note that with the coverage algorithm, these timing bugs were exposed by a large number of the benchmarks (6 out of 10), the highest of rate in our experiment. This indicates that application workloads are sensitive to coherence errors involving timing perturbations. We also noted that in several cases the coverage algorithm took more cycles to uncover a bug than the low-overhead algorithm. Upon closer investigation we found that this was due to a higher number of benchmarks uncovering the bug compared to low-overhead, but at a later time.

D. Area Overhead

To estimate the area cost of CoSMa, we implemented the design in Verilog, then performed synthesis using Synopsys Design Compiler with a 90nm TSMC target library. The resulting area for the master checker was $7508\mu m^2$, while each slave occupied $165\mu m^2$. For comparison, the Sun OpenSPARC T1 processor [16] has a total area of $378mm^2$, with its 8 cores each contributing $11mm^2$. The combined area impact on this processor by adding 1 master checker and 8 slaves is $8828\mu m^2$, only 0.002% of the total OpenSPARC chip area.

Bug	Low-overhead Algorithm		Coverage Algorithm	
	cycles to expose bug	# tests exposing	cycles to expose bug	# tests exposing
concurrent-writes	20M	2	14M	2
write-unaligned	20M	2	14M	2
dropped-message	39M	3	149M	4
delayed-message	missed	0	373M	6
two-stores	39M	3	237M	4
stores-dropped	20M	2	362M	5
delayed-writes	missed	0	339M	6

TABLE IV. **Execution cycles required to expose bugs with SPLASH2 benchmarks.** The same two bugs of Table III could be exposed only by the coverage algorithm even with the SPLASH2 workload.

E. Case Study: CoSMa Integration

CoSMa is designed for easy integration into an existing cache coherence system with minimal design effort. This held true in our experience with a student team implementing CoSMa as part of a class project. The project involved a quad-core alpha processor with L1 and L2 caches maintaining coherence with the `MESI` protocol. The team began by implementing basic `MESI` cache controllers, and added CoSMa capabilities once the basic controllers were operational.

L1 Cache Controller. Modifications to the L1 cache controller included the addition of two states to read and write the history log. Transitions were added to facilitate entry addition to this history: a `MESI` operation proceeds by reading the requested cache line and history, modifying the `MESI` state if the cache line tag matches, appending a log entry to the history, and finally storing the new `MESI` state and history. By taking advantage of banked caches, it was straight-forward to implement the concurrent writing of data and history log.

L2 Cache Controller. The L2 cache controller was augmented to update cache line histories when servicing requests from L1 caches, as well as to orchestrate a coherence check. Similar to the L1 cache, the L2 controller required two additional states to facilitate writing history entries to the cache portion allocated for history storage. Additionally, when a check request comes from an L1 cache controller, the L2 controller was modified to send a message back to each L1, signaling the system to pause normal operation. Once all the L1 controllers have acknowledged, a second broadcast goes out from the L2 controller to initiate the check process.

VII. Conclusions

CoSMa introduces a new framework optimized for the unique challenges of post-silicon cache coherence validation of complex multi-processors. It offers a specialized mode of operation, called CoSMa mode. While in CoSMa mode, the processor stores a compact encoding of a cache line's state history and periodically checks that the history observed at local cache is compatible with that of the second level cache. We developed two variants of the compatibility checking algorithm, one optimized for high coverage, the other for minimal system perturbation. We found these algorithms to be very successful at detecting a set of realistic cache coherence bugs with a performance degradation ranging from 1% to 23%. By leveraging existing hardware, we were able to implement our solution with minimal area overhead, only 0.002%. While related work in runtime verification [11] offers low performance overhead (5-10% communication bandwidth), the area overhead is significant. CoSMa operates under a new set of design constraints, offering observation at full speed, high coverage and minimal system perturbation. It is a distributed solution that does not require pervasive communication during data collection. It can be disabled upon product release, thus exhibiting zero performance degradation to the end user.

Acknowledgments

The authors would like to thank Matthew Fojtik, Bradley Dobbie and Andy Lin for their implementation of CoSMa in a quad-core Alpha design and the valuable feedback provided as a result of this effort.

References

[1] *Intel Core 2 Duo and Intel Core 2 Solo Processor for Intel Centrino Duo Processor Technology Specification Update*, September 2007.
[2] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A reconfigurable design-for-debug infrastructure for SoCs," in *Proc. DAC*, 2006.
[3] M. Martin, D. Sorin, B. Beckmann, M. Marty, M. Xu, A. Alameldeen, K. Moore, M. Hill, and D. Wood, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, 2005.
[4] F. Pong and M. Dubois, "Verification techniques for cache coherence protocols," *ACM Computing Surveys*, vol. 29, no. 1, 1997.
[5] D. Dill, A. Drexler, A. Hu, and C. Yang, "Protocol verification as a hardware design aid," in *Proc. ICCD*, 1992.
[6] A. Pnueli, S. Ruah, and L. Zuck, "Automatic deductive verification with invisible invariants," *Lecture Notes in Computer Science*, 2001.
[7] S. German, "Formal design of cache memory protocols in IBM," *Formal Methods in System Design*, vol. 22, no. 2, 2003.
[8] E. Emerson and V. Kahlon, "Exact and efficient verification of parameterized cache coherence protocols," in *Proc. CHARME*, 2003.
[9] D. Wood, G. Gibson, and R. Katz, "Verifying a multiprocessor cache controller using random test generation," *IEEE Design & Test*, vol. 7, no. 4, 1990.
[10] J. Cantin, M. Lipasti, and J. Smith, "Dynamic verification of cache coherence protocols," in *Proc. ISCA*, 2001.
[11] A. Meixner and D. Sorin, "Error detection via online checking of cache coherence with token coherence signatures," in *HPCA*, 2007.
[12] D. Sorin, M. Hill, and D. Wood, "Dynamic verification of end-to-end multiprocessor invariants," in *Proc. DSN*, 2003.
[13] D. Burger and T. Austin, "The SimpleScalar toolset, version 3.0," http://simplescalar.com.
[14] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta, "The SPLASH-2 programs: characterization and methodological considerations," in *Proc. ISCA*, 1995.
[15] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *IEEE Computer*, vol. 35, no. 2, Feb 2002.
[16] A. Leon, K. Tam, J. Shin, D. Weisner, and F. Schumacher, "A power-efficient high-throughput 32-thread SPARC processor," *IEEE Journal of Solid-State Circuits*, vol. 42, no. 1, 2007.