DACOTA: Post-silicon validation of the memory subsystem in multi-core designs

Andrew DeOrio, Ilya Wagner and Valeria Bertacco
University of Michigan; Ann Arbor, MI
{awdeorio, iwagner, valeria}@umich.edu

Abstract

The number of functional errors escaping design verification and being released into final silicon is growing, due to the increasing complexity and shrinking production schedules of modern processor designs. Recent trends towards chip multiprocessors (CMPs) are exacerbating the problem because of their complex and sometimes non-deterministic memory subsystems, prone to subtle but devastating bugs. This deteriorating situation calls for high-efficiency, high-coverage results in functional validation, results that are be achieved by leveraging the performance of post-silicon validation, that is, those verification tasks that are executed directly on prototype hardware. The orders-of-magnitude faster testing in post-silicon enables designers to achieve much higher coverage before customer release, but only if the limitations of this technology in diagnosis and internal node observability could be overcome.

In this work, we unlock the full performance of post-silicon validation through Dacota, a new high-coverage solution for validating memory operation ordering in CMPs. When activated, Dacota reconfigures a portion of the cache storage to log memory accesses using a compact data-coloring scheme. Logs are periodically aggregated and checked by a distributed algorithm running in-situ on the CMP to verify correct memory operation ordering. When the design is ready for customer shipment, Dacota can be deactivated, releasing all cache storage, and only leaving a small silicon area footprint, less than 0.01% (three orders of magnitude smaller than previous solutions). We found experimentally that Dacota is effective in exposing memory subsystem bugs, and it delivers its high coverage capabilities at a 26% performance slowdown (only during validation) for real-world applications.

1. Introduction

Verification of today's complex microprocessors has become the bottleneck of the design cycle. Despite massive pre-silicon verification efforts, chips are still released with devastating bugs. Errors in the memory subsystem are becoming increasingly common, comprising at least 10% of the escaped and reported bugs in the Intel Core 2 Duo [1]. Memory coherence and consistency, which provide guarantees as to the order of memory operations, are significant sources of escaped bugs and are likely to become more error-prone as designs move from buses towards complex, non-deterministic interconnects. The system-level properties related to memory operation policies are difficult to verify due to the vast state space they encompass and their decentralized enforcement. As technology moves towards large CMP systems, such as the TILE64 [3] and Polaris [23] microprocessors, the verification problem worsens.

To address these shortcomings, post-silicon functional validation has emerged as a new complementary approach, promising to bridge the gap between failing pre-silicon verification efforts and the correctness requirements of multi-core systems. Applied to early silicon prototypes, post-silicon validation enables high coverage as a result of fast execution speeds. However, current post-silicon techniques, such as logic analyzers [24], on-chip assertions [22] and scan chains [5] are plagued by limited internal observability. This precludes existing techniques from adequately validating system-wide properties such as memory coherence and consistency, for which it is extremely difficult to detect and diagnose bugs from the system's external interface.

1.1. Contributions

This work introduces a novel solution to detect functional errors in the memory ordering of CMPs in post-silicon validation. Our solution, called Dacota (Data-coloring for COnsistency Testing and Analysis), offers the benefits of high validation coverage and debugging support at a very small performance impact and near-zero area overhead. Enabled only during post-silicon validation, it incorporates a simple in-hardware activity logging mechanism that observes selected system activity during program execution. Periodically, a software-based validation algorithm examines the logs to detect violations in the ordering of memory operations, indicative of an error in memory coherence or consistency.

When Dacota is enabled, an activity logging mechanism located at each level one (L1) cache stores a compact en-

coding of memory accesses. The caches are temporarily reconfigured to include an *access vector* associated with each line: the access vectors contain a counter "color" value, incremented with each store operation to the line and used to disambiguate accesses to the same cache line during execution. In addition, after each load and store, individual CMP cores log the address and color values of the access in an *activity log*, which is also maintained in the local cache. Thus, the logs record the history of memory accesses in program order for each individual core. When local cache storage is exhausted, activity logs are aggregated and validated by a software-based algorithm, leveraging the existing processor cores for computation. Validation is performed by building a graph from the activity log where vertices represent memory accesses and edges indicate the observed ordering between them. Correct memory ordering is then checked by inspecting the graph for cycles. By leveraging existing cache storage and CPU computation resources, Dacota incurs an extremely small silicon area overhead. Finally, Dacota can be completely disabled upon product shipment, leading to zero performance impact to the end user.

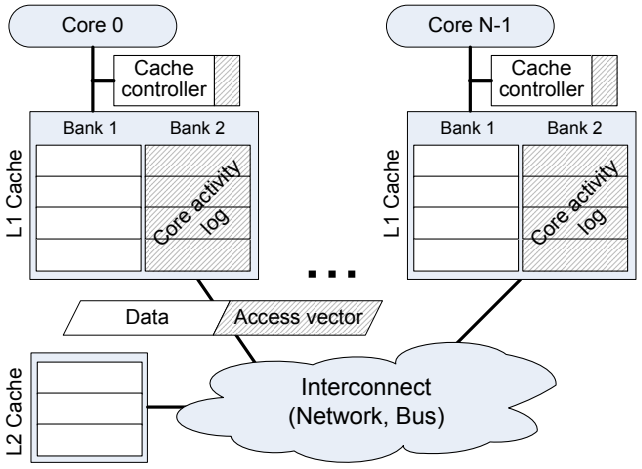

Figure 1. CMP reconfiguration for Dacota validation. Cache lines are partitioned to include an *access vector* tracking the order of memory accesses. A portion of each cache is reclaimed and used as *activity log* storage for load/store operations. Finally, the cache controllers are augmented to include supporting hardware.

2. Dacota Overview

Dacota's architecture is embedded in the CMP design to be validated and requires minimal hardware modifications. A schematic of its components is shown in Figure 1. We assume a generic CMP architecture where multiple simple processing elements (cores), each with private L1 caches, are connected via an on-chip interconnect fabric to a shared L2 cache. When Dacota is enabled, the system is reconfigured so that a portion of the cache resources are reserved for Dacota's *core activity logs* (hashed blocks in the L1 caches of Figure 1). The portion of the cache used by Dacota is configurable, a simple implementation allocates half of the cache to core logs, and the other half to normal data storage.

Processor activity is organized into *epochs* (Figure 2), where program execution alternates with Dacota's checking phase, which can be divided into log aggregation, graph construction and policy validation. During normal program execution, Dacota monitors activity in the background by updating and transmitting access vectors along with data, and logging snapshots of the vectors. When log resources are exhausted, program execution stops, data in transit is allowed to reach its destination, and all data portions of the caches are frozen. All cores then drain their activity logs into a dedicated region of un-cacheable memory (*log aggregation*). Next, each core in the system builds a *consistency graph* representing the ordering of memory operations. This consistency graph is examined by the *policy validation algorithm* to expose memory ordering errors. If the analysis exposes an error, information from the activity logs can be leveraged to support subsequent diagnosis and debugging. Otherwise, activity logs and access vectors are cleared and the next epoch may begin.

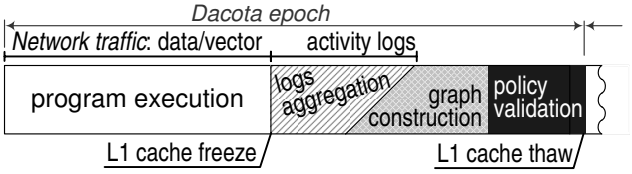

Figure 2. Dacota execution flow. When Dacota is enabled, normal benchmark execution progresses with data and access vectors transferred together and while activity is logged in the background. When log resources are exhausted, they are aggregated and analyzed by a graph-based algorithm. If an error is found, the logs are presented to the user for diagnosis, otherwise execution resumes.

When Dacota is active, each cache line is partitioned to include additional information alongside the data block, in an *access vector* that records the number and the order of store operations issued to the line. Each core modifying the data in the cache line also updates the corresponding access vector. Traveling throughout the system with its data block, the access vector records the order of store operations to cache lines. A snapshot of the access vector is also stored in the local cache's *core activity log* upon each memory operation, along with the address. These logs are later analyzed to validate the ordering of memory operations. When required to support specific consistency models (such as Weak Consistency), Dacota may also log special memory synchronization instructions.

When logging resources are exhausted, Dacota's analysis algorithm validates the ordering of memory operations from the contents of the activity logs. The algorithm builds a consistency graph from the aggregated core activ-

ity logs; vertices represent memory accesses, while directed edges indicate operation sequencing as perceived by different cores. Analysis of the consistency graph determines if a memory ordering error has occurred with respect to the memory consistency model. Errors manifest as a loop in the graph. The analysis engine can also expose coherence violations if they are manifest in the consistency graph or through incompatible access vectors in the activity log. Dacota's analysis algorithm is executed entirely in software on the existing CPU resources of the CMP.

3. Activity Logging

The activity logging system in Dacota records the order of shared memory accesses observed by different cores. To this end, we maintain both an access vector attached to each cache line, as well as activity logs residing in local caches. The information in the access vectors and logs is updated concurrently with program execution and incurs no performance overhead during this phase. However, when the log storage resources are exhausted, normal execution is suspended and the logs are aggregated and analyzed by the policy validation engine, discussed in Section 4.

3.1. Access vector

Dacota logs the order of accesses to cache lines using a scheme based on data coloring. Each cache line is partitioned into two parts: one for program data and the other configured as an *access vector*; these travel together through the interconnect and caches of the CMP. Each core has a dedicated entry in the vector, updated when that core performs a store access to the cache line. An additional entry in the vector is reserved for a counter tracking the total number of stores to the line since the beginning of the epoch. Figure 3 shows several examples of access vectors, with the counter entry shown in gray.

At the beginning of an epoch, the counter and all entries of the vector are initialized to zero. With each issued store, Dacota automatically increments the counter and copies its value to the vector entry associated with the issuing core. Updates to the counter are accomplished automatically by Dacota hardware and do not require read-modify-write operations to be issued by the CPU. A saturated counter triggers the end of the program execution phase in an epoch, a necessary measure to ensure counter uniqueness. The end result is an access vector with monotonically increasing counter values indicating the chronological order in which cores modified a line.

To illustrate how access vectors operate, we show several examples in Figure 3.a-d. Part 3.a shows the vector associated with an unmodified cache line: all of its N entries and the counter have a zero value. Figure 3.b shows the result of three stores (as indicated by the counter value 3) issued by

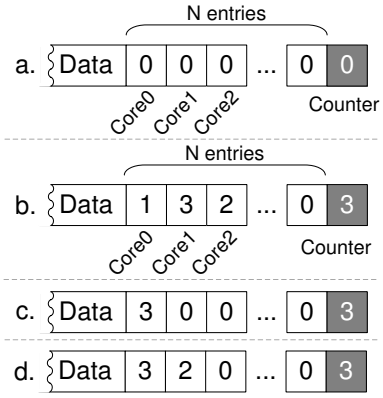

Figure 3. Dacota access vector. An access vector associated with each cache line is used to track the order of store operations. **a.** Access vector at the beginning of an epoch. **b.** Access vector indicating modifications (in order) by Core0, Core2 and Core1. **c.** and **d.** Examples of aliasing due to multiple stores by the same core. In c. the order can be inferred because of the other entries being at 0; in d. the complete order cannot be recovered.

cores 0, 2, and 1, in this order. Initially, the line starts with a vector as in 3.a, and after Core0's store, the counter and the first entry are updated to 1. When Core2 issues a store, the counter is incremented to 2 and this value is copied into the third entry. Finally, the store by Core1 changes the access vector to its final state shown in Figure 3.b. Observe that in this example the vector uniquely identifies the order of all store operations to the address block. In contrast, Figure 3.c shows a situation where three store operations have occurred, yet we can only determine which core executed the last one. In this example, the fact that all other entries are at zero indicates that Core0 is indeed responsible for all the stores. In the last example in Figure 3.d, we cannot determine the unique order of the accesses to the cache line, since we cannot determine whether Core0 or Core1 issued the first store operation. Note, however, that if we had a snapshot of this access vector before the third store operation was issued, then we would be able to establish the sequence of operations precisely.

To manage access vector updates, we make a small addition to the cache controllers (hashed areas in Figure 1). This hardware component is responsible for incrementing the counter and updating the vector entries. At the cost of increased hardware complexity, it would be possible to eliminate the counter entry from the access vector and simply retrieve the counter value by extracting the highest value in the access vector entries. This alternative approach incurs higher area cost, but could be interesting for system where the resources for the access vector are extremely limited.

3.2. Core Activity Log

While access vectors record the order of accesses to individual cache lines, *activity logs* record the order of accesses

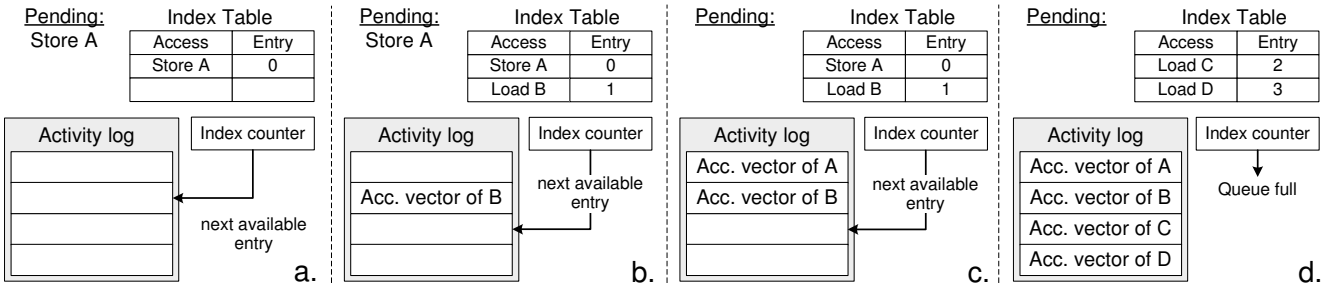

Figure 4. Dacota activity log operation. The activity log records snapshots of access vectors in program order, while data dependencies are exposed by the access vectors' content. The index table connects accesses that have not yet completed to their corresponding log entry. **a.** Store to line *A* is issued and the counter and the index table are updated. **b.** Load to line *B* is issued and completes before Store *A*. **c.** Store to line *A* completes and the updated vector is logged. **d.** After loads to C and D the log fills and the policy validation algorithm runs.

among different cache lines. Stored in a reconfigured portion of the caches, activity logs record a series of access vector snapshots. When a core issues a load or a store, Dacota copies the updated access vector to the activity log, together with the type of access (load/store) and the cache line tag. The activity log is maintained as a queue and entries are allocated in program order, but copied in order of completion, which may differ from program order. By leveraging the order and contents of the activity log, Dacota can later reconstruct the order of memory operations perceived by each core. To reconfigure Dacota's portion of the L1 cache as a queue, we augment it with a simple up-counter that cycles through the allocated ways and sets. The tag array stores a portion of the location's address, while the data block stores spill-over address bits and the instantaneous value of the access vector associated with the line. In addition, since the order of completion of memory operations may be different from program order, we add a small index table for conversion between outstanding memory accesses and entries in the cache. Because we only need to index the outstanding memory accesses, the table can be quite small.

Figure 4.a-d shows an example of activity log operation. First, a local core issues a store to location A, allocating an entry for it in the log and recording the mapping in the index table. Before the store completes, a load to address B is issued and completed (Figure 4.b), logging the access vector of line B. When the store completes in Figure 4.c, the vector of line A is copied to its pre-allocated entry. When the log fills (Figure 4.d), a signal is asserted and the policy validation algorithm is invoked.

In developing Dacota, we observed that logging each memory access led to prohibitively large storage requirements. Thus, we optimized our design to log a load access only if it triggered a cache miss, either because the data block is not cached, or because the copy in the cache is obsolete due to modification by another core. No loss in coverage is incurred by this optimization, when operating under the simple assumption that (hit) accesses to local caches are serviced correctly.

3.3. Activity Logging Example

An example of Dacota's logging system in operation is presented in Figure 5. We assume a CMP with two cores, MESI coherence protocol and a load/store buffer that enforces strict program order. For this and subsequent examples we adopt the following notation for the access vector format: $c0|c1\ \underline{cnt}$, where $c0$ and $c1$ are the entries for Core0 and Core1, and $\underline{cnt}$ is the counter. For example, the access vectors in Figure 3.a and 3.d are recorded as $0|0\ \underline{0}$ and $3|2\ \underline{3}$, respectively. For simplicity we do not show the index table and the index counter of the activity log.

Initially, both L1 caches, as well as the logs, are empty (not shown). The first access (store to A) is issued by Core0 in Figure 5.a and brings line A into the L1 cache in the modified state. The access vector of the line is then updated: the counter is incremented and the new value (1) is copied into Core0's entry. The resulting value $1|0\ \underline{1}$ and the address of A is copied to the core's activity log. Subsequently, Core0 issues a load to B and Core1 issues a store to A (Figure 5.b). As a result, line B (with access vector $0|0\ \underline{0}$) is brought to the local cache and recorded in the activity log of Core0. Note that since a load operation is performed, the access vector is not updated. The consequences of Core1's store, on the other hand, are as follows: line A is invalidated in Core0's cache and it is moved to Core1 in modified state. The vector is transferred together with the data and is updated to $1|2\ \underline{2}$. The log of Core0 still preserves a log entry for a previous store to A, while the line itself is no longer in Core0's cache. Finally, in Figure 5.c Core0 issues a load to A and Core1 issues a store to B. The latter invalidates line B in Core0's cache, transfers it to Core1 and updates the vector to $0|1\ \underline{1}$. The former, puts line A in shared state and records its access vector in the activity log of Core0.

At the end of the execution we can use the logs to see how the order of operations was perceived by individual cores. For example, we observe that Core0 has the following log entries A:$1|0\ \underline{1}$, B:$0|0\ \underline{0}$, A:$1|2\ \underline{2}$. Since the last entry appears also in Core1, we can thus determine that when Core0 fetched line B, this was not yet modified by Core1.

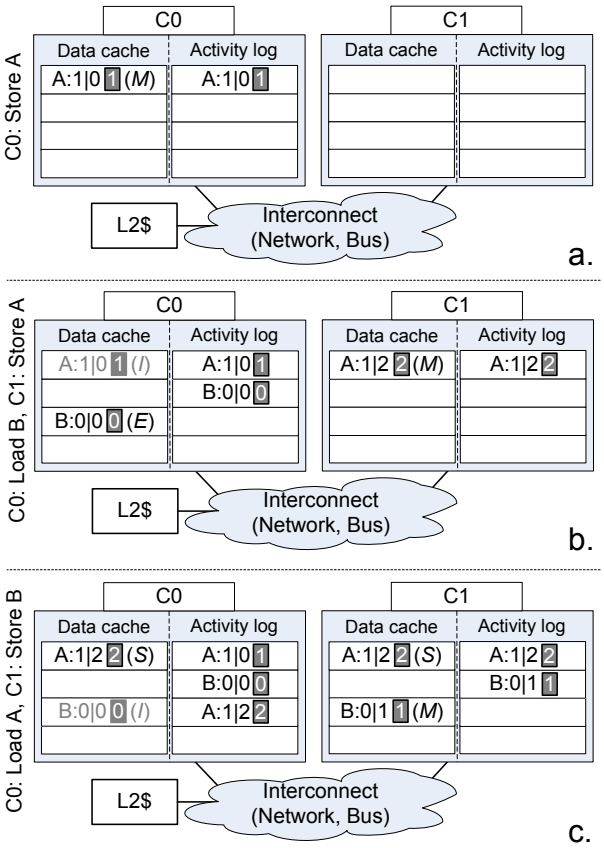

Figure 5. Example of Dacota activity logging system. The example is based on a system with 2 cores, MESI protocol and in-order execution of individual cores' memory operations. **a.** Core0 stores to A, updating and logging the access vector. **b.** Core0 loads B and updates the activity log. Core1 stores to A, updates its vector and logs it. **c.** Core0 loads A with the updated access vector, while Core1 modifies B, and writes the log.

4. Policy Validation Algorithm

Dacota's policy validation algorithm takes the activity logs as input, builds a directed graph representing the memory operation ordering, and checks the graph for errors. The algorithm is invoked every time log resources are exhausted, and begins with aggregating the access logs. This process overlaps with the graph construction process, which may begin as soon as the first logged data is available, and it is followed by policy validation through graph analysis. To minimize the area overhead of Dacota we implemented the checking algorithm in software running on the CMP's cores.

4.1. Access Log Aggregation

When a core detects that its log is full or the counter of the accessed line's vector reached the maximum value, it broadcasts a message requesting validation. Upon receipt of the message, all cores are required to stop execution and complete all pending memory operations. Then, the data portions of the caches are frozen, and the activity logs are transferred to un-cacheable memory, where they are accessible by all cores for graph construction and analysis.

4.2. Graph Construction

After activity logs are relocated to main memory, Dacota proceeds to build a directed graph, representing the ordering of memory operations. Vertices in the graph represent unique memory accesses issued during the epoch; while edges represent the ordering constraints specific to the consistency model adopted in the system. As graph construction progresses, Dacota also conducts a coherence invariant check using the access vectors of the individual lines.

```
Graph_Construction()
  Graph G, Coherence_Order_Map M
  Activity_Log L[0..N-1]
  Foreach core c in N
    Foreach entry e in L[c]
       If Exists M[Address(e)]
          Verify_Coherence(M,e)
       Add_Coherence_Order(M,e)
       Add_Vertex(e,G)
       Edges E = Ordering_Edges(L[c],e)
       Add_Edges(E,G)
    End
  End
```

Figure 6. Graph construction algorithm. The algorithm iterates through all history logs, generating vertices and ordering edges for the graph, and checking the coherence invariant.

The pseudocode for the graph construction algorithm is given in Figure 6.a. The algorithm iterates over each core and log entry, performing a preliminary check to verify that all store operations to an individual cache line are compatible with a unique ordering of events. In other words, the algorithm checks that for a single line all cores agree on the same order for write operations. For this coherence check, we employ a data structure that maps the line's address to a complete list of stores issued to this line. Each time the line's address is encountered in a log, its access vector is compared to the list to see if there are any violations. If the entries of the vector reveal an access that was not previously observed, the ID of the core that issued it is added to the list in the proper location. After this preliminary check, we use the information in the log entry to augment the graph for the consistency model. At the end, the graph is checked for loops, which are indicators of an error in memory operation ordering. Below we provide insights in the specific graph construction rules for a range of consistency policies.

Sequential Consistency. Sequential consistency requires that the order of operations in all cores is perceived uniquely throughout the system. For systems employing this model, loads and stores correspond to vertices in the

consistency graph; edges in the graph are of two types: *program order edges* and *address reference edges*. Program order edges are imposed by the order of entries within each activity log, while address reference edges are derived from operations issued to a same location by different cores and represent data dependencies (see Figure 7.b).

Total Store Ordering (TSO). This consistency model relaxes sequential consistency by allowing loads to complete before previously issued stores to a same location. In the graph construction for TSO, vertices are created only from store operations, while edges are derived from load information. *Program order* and *address reference edges* are constructed similarly to the Sequential Consistency model. In addition, *chronological edges* are derived from the values of access vectors returned by load operations and represent inferred orderings between stores to different locations. In particular, we identify the most recent store operation that modified the loaded location by inspecting its access vector and infer that any subsequent store to that same location had not yet been executed. We then retrieve the vertex corresponding to the most recent store operation issued by the core, which executed before the load (to any location), and draw an edge from this vertex to the one corresponding to inferred *next* store. In addition, we connect the last store that accessed the loaded location to the *next* store issued by this core. With reference to the example in Figure 7.c, the two chronological edges ($A_2 \rightarrow B_2$ and $B_1 \rightarrow C_1$) are derived precisely in this way.

Processor Consistency. This model requires a that the perceived order of store operations issued by a single core is the same throughout the entire system. However, there is no requirement for the interleaving of stores issued by different cores. For this model, each core builds its own graph using a process similar to that of TSO, but creating vertices only for stores issued by that core. Edges are generated from load operations issued by all cores, as in TSO.

Other consistency models. Weak consistency models require that only special instructions (such as memory barriers) are perceived in a unique order throughout the system, while the observed interleaving of accesses between the synchronization operations may be different for different cores. In this case, Dacota logs the synchronization instructions in addition to loads and stores, and uses them as vertices in graph construction. The edges between the operations are derived from log entries of ordinary accesses.

4.3. Graph Analysis

Consistency graphs in Dacota are constructed to reflect the order in which accesses performed by individual cores are perceived in the system. In order to find errors, Dacota searches the graphs for loops, employing a modified version of the Depth First Search (DFS) algorithm [11]. The algorithm retains the complexity of the underlying implementation of DFS [20], that is, to $O(E)$, where E is the number of edges in the graph. However, to ensure maximum performance, we aggressively apply transitive closure during graph construction, thus reducing the number of edges.

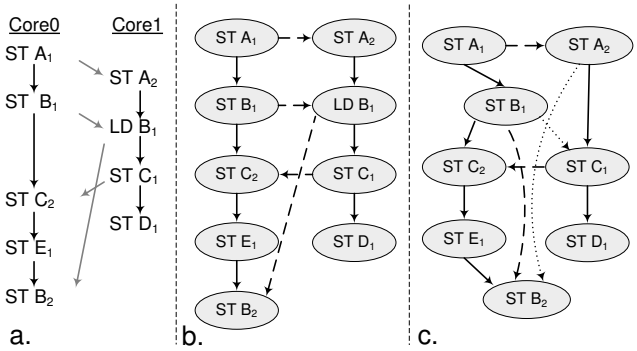

Figure 7. Graph construction example. a. Interleaved sequence of loads and stores issued by two cores. Black lines represent program order constraints, while gray lines represent data dependencies. **b.** Consistency graph for Sequential Consistency. Solid lines represent program ordering constraints, while dashed ones indicate address reference edges. **c.** Consistency graph for TSO. The additional dotted lines represent chronological edges.

Note that we do not use any additional hardware for implementing policy validation, thus dramatically reducing the silicon area overhead requirements of Dacota. This is a crucial feature distinguishing Dacota as a post-silicon solution from runtime approaches that also use graph analysis techniques. Moreover, it allows us to parallelize the analysis for common weaker consistency models where several distinct graphs must be constructed. Even for consistency policies that require the construction of a single graph, such as Sequential Consistency, Dacota exploits the cores of the CMP to parallelize the cycle detecting algorithm by starting from distinct graph vertices. To further boost the performance of Dacota during the policy validation phase, we reconfigure the storage previously occupied by the activity logs (now residing in main memory) to be used as regular cache space. When the check completes, caches are reconfigured once again to make space for the activity logs, the data cache is thawed and the next Dacota epoch begins.

4.4. Error Detection Examples

In this section we overview an example of a coherence error and a consistency error, showing how Dacota identifies them. Figure 8 demonstrates a case when the state of cache line A is not updated properly upon Core1 requesting it as exclusive. In a system operating correctly, Core0 should first invalidate the line in its own cache. Assume instead, as shown in Figure 8.b, that due to a design error, Core0 incorrectly modifies the line's state to shared (we as-

sume a MESI protocol). This, in turn, leads to the situation shown in Figure 8.c, where two modified copies of the line exist in different caches at the same time. Thus, the cores disagree on the order of the store operations that they issued to the line. Dacota detects this type of issues while analyzing the accesses to address A during graph construction.

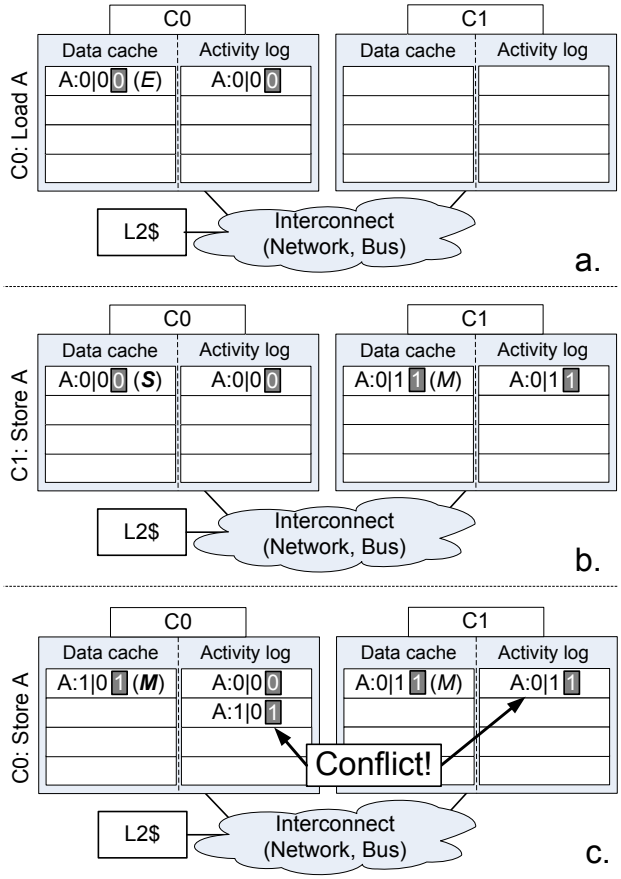

Figure 8. Coherence conflict example. a. A store to line A is executed by Core0. **b.** Core1 requests exclusive access to line A to perform a store, however, the cache of Core0 is erroneously updated. **c.** Core0's store to line A updates the access vector, leading to a conflict detected by Dacota at graph construction time.

A more complex fault, this time in memory consistency, is shown in Figure 9, where we assume Sequential Consistency. When Core1 issues a number of load operations, their execution order is erroneously reversed, possibly because of a non-deterministic interconnect or because of incorrect behavior of the memory controller. The load to A completes first, bringing an unmodified vector associated with the line into Core1's cache and activity log. Then, the stores issued by Core0 complete, and, finally, the load to line B completes, returning to Core1 the line's data and access vector. In this scenario, the reversal of the execution of the loads becomes visible because the perceived order of the stores to A and B is incompatible among the two cores, generating a loop in the consistency graph, which is detected by Dacota.

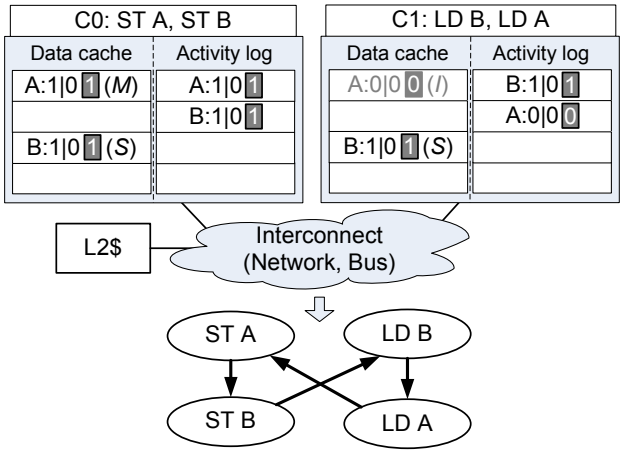

Figure 9. Sequential Consistency violation example. A load to A in Core1 is erroneously reordered to execute before the load to B and completes first. Then, stores from Core0 execute (invalidating A in Core1's cache), and, finally, the load to B in Core1 completes. The order in which accesses to A and B are observed by Core0 and Core1 in this case is different, which leads to a loop in the consistency graph built by Dacota, and thus an error is flagged.

4.5. Checking Algorithm Requirements

To minimize the area overhead of Dacota, its checking algorithm is designed to run in software, as opposed to dedicated hardware. Therefore, for our approach to be reliable, we require computational correctness from the cores, as well as their ability to access main memory. Main memory accesses use the same subsystem that Dacota is verifying, however, bugs in the memory subsystem are unlikely to cause analysis errors due to the absence of memory race conditions during Dacota analysis. The cores drain the activity logs into disjoint memory regions and then use this information without overwriting it. Moreover, since the logs are aggregated in the uncacheable memory, caches are not involved in the transfer, eliminating the chance that coherence or consistency bugs corrupt the analysis process.

5. Strengths and Limitations

While Dacota is effective in exposing a diverse range of functional errors in the memory subsystem of a CMP architecture, there are several limitations to its approach. First, its bug catching ability is dependent on the quality of the stimulus, that is, the program running on the system under test. More specifically, Dacota can only find bugs that are uncovered by the workload running on the system. For example, workloads without shared data will not uncover coherence or consistency related bugs, nor will small ones, which do not stress the memory system. Bugs that result in deadlock or system hang are also not flagged by Dacota, as we assume that forward progress is always maintained.

Bugs that do not affect execution semantics evade capture by our validation system. For instance, consider one

cache line shared by two cores, both in the shared state of a MESI protocol. If one processor silently (and incorrectly) modifies the line's state to exclusive the system becomes incoherent. Yet, if no store is performed to the line, the execution semantics of the program is not violated. A store, however, will make the error manifest, since it updates the activity log and the access vector. Furthermore, when Dacota is implemented with the optimization that activity is logged only on cache misses (Section 3.2), errors related to how cache hits are handled locally would not be detected by our approach. However, this aspect can be verified through properties local to a single cache and its core's memory interface by using traditional methods, allowing Dacota to focus on the validation of system-wide invariants.

5.1. Debugging with Dacota

One of the major strengths of Dacota is its support for the debugging effort. In addition to the detection of complex coherence and consistency bugs, Dacota can also help find the root cause of an error once it has been discovered. The activity logs present at the end of an epoch become a useful debugging resource, providing detailed information about which cache lines were accessed, which cores accessed them, and when stores occurred. With this information, an engineer can clearly see the activity of all the processor cores, as well as system-level events leading up to the bug. Moreover, since the data portions of the cores' L1 caches are frozen during Dacota analysis, the data residing in them can also be used to provide additional insights about the violation. Finally, Dacota can be augmented with hardware blocks to log the state of each core's internal registers at the end of an epoch, providing engineers with more info about the program execution. These features allow Dacota to be efficiently used for debugging of complex and non-deterministic CMP systems.

5.2. Design Considerations

In order to make Dacota a viable post-silicon solution, its design was driven by three primary goals: high coverage, low area overhead and debuggability. Performance was an additional consideration, as high execution speed is critical to our primary goal of high coverage. Early in the design phase, we considered attaching only a sequence counter to each cache line and storing these values in the activity log with each memory operation. While this minimalist setup eliminates the need for access vectors, we quickly found that the analysis algorithm corresponding to this storage mechanism was grossly inefficient. Since the sequence counter did not record the order of accesses to the line, the order had to be inferred from the logs of the other cores in the system, requiring the algorithm to walk the entire set of aggregated access logs during the construction of each vertex in the graph. With the logs placed in un-cacheable memory, the performance overhead of the graph construction process outweighed the savings in log storage space. Furthermore, this scheme lost writer identification information, thus significantly hampering debuggability. With the addition of the access vector, graph construction becomes much faster, as well as it provides additional debugging information. Our design decisions are validated by our experimental results, which indicate that longer periods of execution between checks do not necessarily lead to better performance for the graph analysis algorithm.

6. Experimental Evaluation

We evaluate the efficiency of Dacota in a simulated CMP system, determining its error coverage, performance and area impact. Investigating how different configurations affect Dacota, we explore several activity log lengths and their effect on consistency graph size and policy validation algorithm runtime. In addition, we measure how the amount of communication and computation overhead incurred by Dacota ranges across multiple benchmarks and log sizes.

6.1. Experimental Framework

Our simulation framework was based on a CMP system modeled with the Wisconsin Multifacet GEMS memory sub-system simulator [14]. The CMP contained 16 cores, each with a 16 entry load/store buffer, 128KB L1 cache, a single 4MB L2 cache and a 4x4 on-chip mesh interconnect. The MOESI directory protocol was used as the coherence protocol, along with the Total Store Ordering consistency model. For several additional experiments, we evaluated systems with token coherence, as well as crossbar and switch-based interconnects. Dacota was implemented as a simulator plug-in, which included methods for access vector manipulation, core activity log management, the policy validation algorithm was implemented using the Boost Graph Library [20]. The runtime of the algorithms was calculated using the SimpleScalar architectural simulator [4].

The set of workloads used to evaluate Dacota was a combination of real world programs and random stimulus. We used the ten SPLASH2 benchmarks [25], considering sections of 10,000,000 instructions generated by the Virtutech Simics simulator [13]. In addition, to induce more stress on the memory subsystem, we created eight tests of directed random stimulus with varying degrees of data sharing, each containing 1,000,000 memory accesses. Three of these benchmarks used a fairly small address space that could fit into the cores' L1 caches without eviction, while the other five used significantly larger memory ranges and could not be fully contained in the L1 caches. Additionally, we used the GEMS built-in random test generator executing the "barrier" and "locks" patterns.

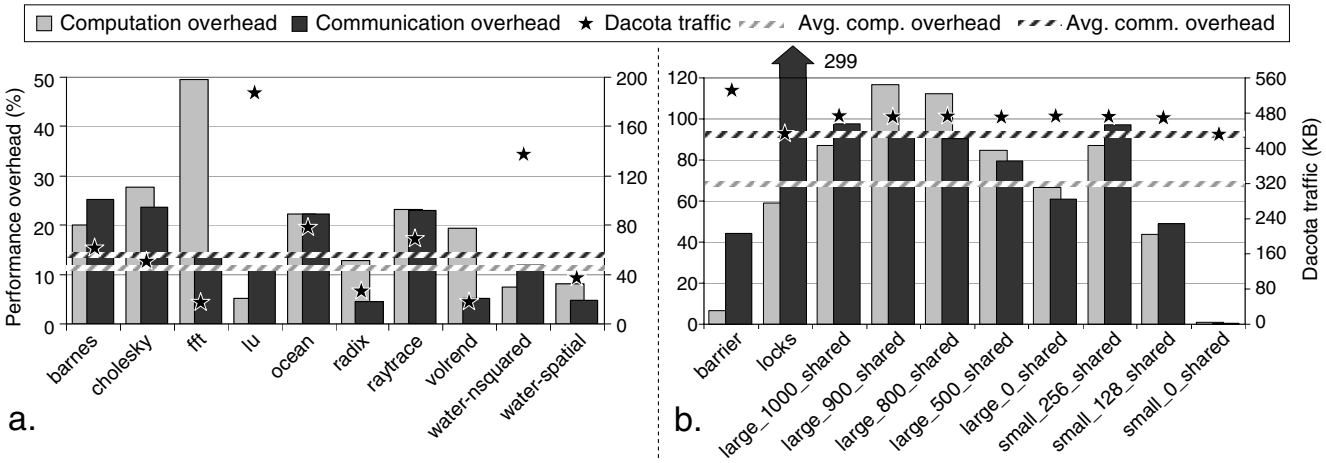

Figure 10. Dacota performance overhead and additional traffic for an activity log size of 256 entries. **a.** Overhead for SPLASH2 benchmarks. **b.** Overhead for random stimulus.

6.2. Design Error Coverage

In our first experiment, we introduced eight coherence and consistency related errors into our simulation model, inspired by known issues with industrial CMPs. We then ran our SPLASH2 and random stimulus benchmarks with Dacota enabled, recording the number of cycles required to discover the bug (Table 1). We found that the activity logging scheme of Dacota is capable of quickly finding complex coherence and consistency bugs.

Table 1. Design error coverage by Dacota.

Bug name	Description of the error	Avg. cycles to expose
shared_store	store to a shared line may not invalidate other caches	0.252M
invisible_store	store message may not reach all cores	1.32M
store_alloc_1	store allocation in any core may not occur properly	1.93M
store_alloc_2	store allocation in a single core may not occur properly	2.27M
reorder_1	invalid store reordering (all cores)	1.38M
reorder_2	invalid store reordering (by a single core)	2.82M
reorder_3	invalid store reordering (to a single address)	2.87M
reorder_4	invalid store reordering (to a single address by a single core)	5.61M

6.3. Performance Evaluation

We also investigated the computation and communication overhead of Dacota relative to normal program execution time. In this study, we assumed that one half of the L1 cache (64kB) was devoted to data and associated access vectors, and the activity log was limited to 256 entries. As shown in Figure 10, the performance overhead was well within the acceptable range for post-silicon solutions: approximately 26% for the SPLASH2 benchmarks (for comparison, overheads of 200-300% are perfectly acceptable in this domain). The overhead for random benchmarks is somewhat higher due to the nature of these tests, which were designed specifically to stress the memory subsystem. Our implementation of Dacota serializes graph construction and log transfers: in practice they could be overlapped, leading a smaller aggregate overhead than the one reported in Figure 10. Moreover, since Dacota can be disabled upon shipment, these performance overheads are only incurred during in-house analysis of a prototype.

In our next study, we varied the size of Dacota's activity logs, measuring communication and computation overheads. The results of this analysis for the SPLASH2 benchmarks and random tests are presented in Figures 11.a-b and 12.a-b. We found that the communication overhead remains nearly constant despite different queue sizes, regardless of workload. However, computation overhead seems to exhibit several interesting trends, as shown in Figure 11: for some benchmarks (*fft, lu, locks*), the ratio of analysis time to normal program execution time decreases as the activity log size grows. Interestingly, the computation overhead time for several other benchmarks, such as *cholesky, radix, etc.*, exhibits a local minimum at medium log sizes. This can be explained by the growing complexity of the consistency graph, which in turn results in an increased time to build and analyze it. We found through several analyses that the average aggregate overhead (computation and communication) is minimal when the activity log is 256 entries long.

In the next experiment, we ran the same benchmarks and activity log lengths, but varied the interconnect topology (crossbar or hierarchical switch) and used a different underlying coherence protocol (token coherence). Our results did not demonstrate any appreciable difference in Dacota's performance for different network topologies, however, changing the coherence protocol to token coherence resulted in an 8% reduction in communication overhead. We found that in this case, the ratio of Dacota traffic to normal system traf-

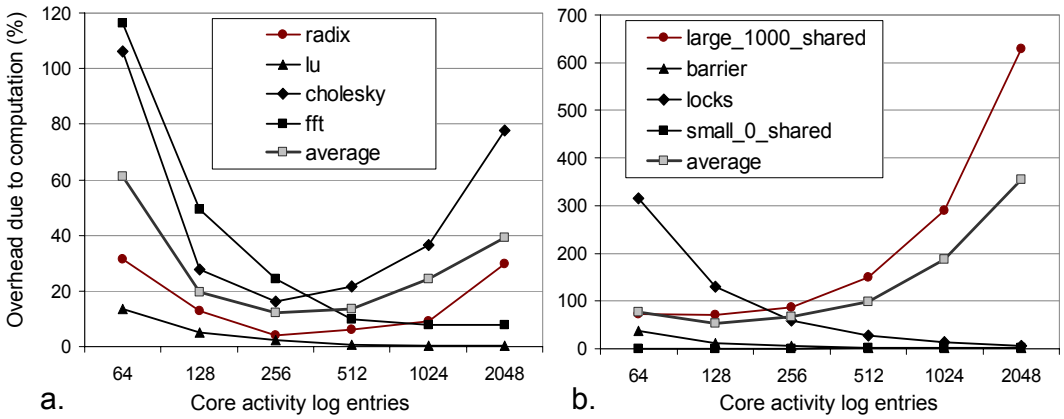

Figure 11. Dacota computation overhead. a. Overhead for SPLASH2 benchmarks (4 individual benchmarks and average of all 10 benchmarks is shown) **b.** Overhead for directed random tests (4 individual tests and average of all 10 tests is shown).

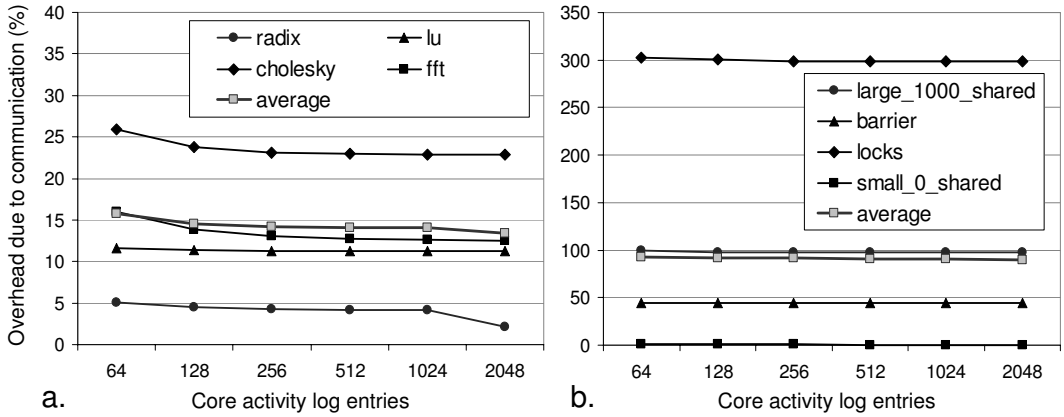

Figure 12. Dacota communication overhead. a. Overhead for SPLASH2 benchmarks (4 individual benchmarks and average of all 10 benchmarks is shown) **b.** Overhead for directed random tests. (4 individual tests and average of all 10 tests is shown).

fic is lower due to the larger number of control messages required to implement the token coherence scheme.

Finally, we measured the number of cycles between invocations of the policy validation algorithm. The average time between analyses (in simulation cycles) is presented in Figure 13.a for SPLASH2 benchmarks and Figures 13.b and 13.c for the random tests. We found that the time between checks grows with increasing activity log size. An interesting observation presents itself in the *barrier* benchmark in Figure 13.b, where the periodicity of Dacota checks remains constant past 512 log entries. The plateau in this benchmark occurs because the check is triggered by access vector counter saturation, not due to the filling of the log. Another special case is illustrated in Figure 13.c by benchmark *small_0_shared*. Small random benchmarks in our experiments have address spaces that fit completely in the L1 caches of individual cores. Since this particular benchmark has no shared data, lines cached by the cores are never invalidated or evicted. Thus, with the optimizations discussed in Section 3.2, the core activity logs never fill up throughout the entire execution of the benchmark, and Dacota analysis is invoked only once, after the benchmark's completion.

6.4. Area Evaluation

To analyze the area impact of Dacota, we implemented the additional hardware required by our solution in Verilog HDL. The module included a block for updating the counter and access vector, a state machine for activity log management and an index table for conversion between program order and performance order. The module was synthesized with Synopsys Design Compiler targeting a TSMC 90nm library. The area for a control module, one of which is added to each processor core in a CMP system, is 5,216μm^2. For comparison, an OpenSPARC T1 [12] chip occupies approximately 378mm^2. Placing a Dacota module on each of the 8 cores in this design results in an overhead of 0.01%. This low overhead is largely due to Dacota's reuse of existing hardware structures, such as cache storage.

In order to put Dacota's area overhead into perspective, we compared its storage requirements to two runtime solutions: [7] and [15]. Since Dacota reconfigures the caches for temporary storage, it requires very little additional hardware, a 2B counter and a 32B table at each node, totaling 544B for our 16-processors system. A conservative esti-

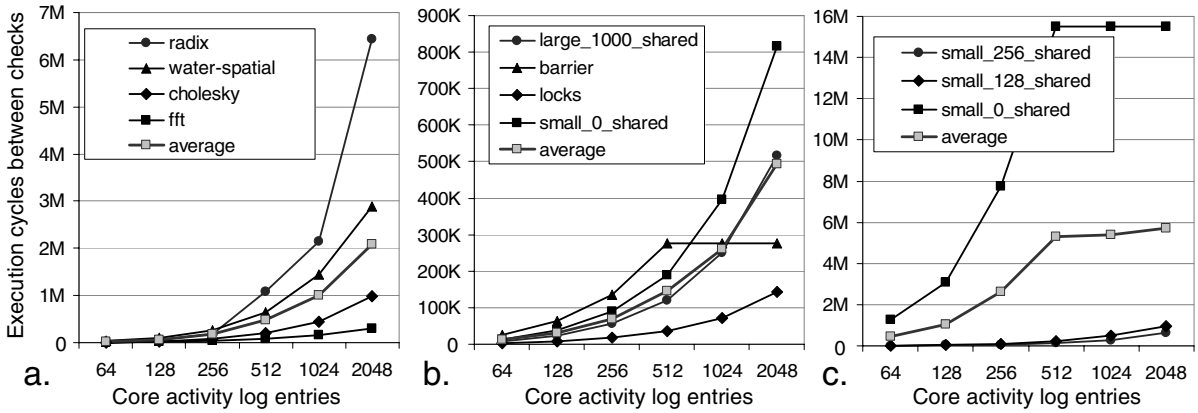

Figure 13. Number of simulation cycles between checks. Policy validations are invoked by Dacota when an activity logs fills or an access vector counter reaches the maximum value. **a.** Number of simulation cycles between Dacota checks for SPLASH2 benchmarks (4 benchmarks and the average of all 10 benchmarks is shown) **b.** Number of simulation cycles between Dacota checks for *barrier, locks* and larger random benchmarks with different degrees of sharing. **c.** Number of simulation cycles between Dacota checks for smaller random benchmarks with different degrees of sharing. With activity log length greater than or equal to 512 entries, the benchmark with no sharing fits entirely into a single Dacota epoch.

mation of the added storage in Chen *et al.*, [7] results in 171kB for a similar 16 node system. Since the functionality of this solution is dependent on checkpointing by SafetyNet [21], an additional 432kB are required, bringing the total to 603kB. In the case of Meixner, *et al.*, [15], 486kB are required by the technique, totaling to 918kB when SafetyNet is included. These estimates are extremely conservative, as they do not consider the area required by logic implementing the checkers. Conservative estimates notwithstanding, the Dacota solution is approximately 1,135 times smaller than [7] and 1,728 times smaller than [15].

7. Related Work

The space of memory subsystem verification can be divided into three major approaches: pre-silicon, post-silicon and runtime. The set of pre-silicon solutions boasts an extensive body of work, most notably several formal approaches [2, 10, 9, 17], which use mathematical techniques to reason about coherence and consistency. While formal verification has the advantage of exploring all scenarios in the design state space, it is limited to small designs and to abstractions. Moreover, while formal verification of abstract representations has been accomplished with success, its results are inadequate, as the implementations of these abstract models are much more complex and cannot be validated. In contrast, simulation-based approaches are able to consider the full system-level implementation of a design, as in [26], where the authors leverage constrained-random simulation to boost the coverage of a design. Unfortunately, the slow speed of pre-silicon simulation critically limits the coverage of this approach.

Runtime solutions to the problem of memory subsystem correctness have emerged to ensure the correctness of multi-core designs deployed in the field. Meixner *et. al.* [15] approach the problem by adding checkers to verify memory consistency. Another runtime solution, proposed by Chen *et. al.* [7], leverages the constraint graph-based approach developed in [6, 19]. This solution adds dedicated observation hardware at each core, periodically analyzing the collected information with a centralized hardware checker. While runtime solutions provide effective protection against soft errors, their ability to recover from functional errors is not guaranteed, since the same errant hardware is also used to recover from the bug. Furthermore, the silicon area required to implement these solutions is significant, three orders of magnitude more than Dacota (Section 6.4), thus precluding the use of runtime solutions for post-silicon validation. On the other hand, Dacota targets functional errors, and operates under a different design philosophy optimized for the post-silicon regime: zero performance cost to the end user and near-zero area cost to the designer.

Work in the sphere of execution replay [16, 18, 27] is relevant to Dacota's logging approach. These works log memory activity and values in order to provide deterministic execution replay. Dacota differs from these approaches in that it does not store data values. Furthermore Dacota only involves caches for this task, while related logging approaches necessarily involve main memory.

Recent work in post-silicon verification targets cache coherence in CMP systems [8]. In this work, the authors describe an alternate mode of a operation for cache controllers which allow the reconfiguration of system resources to store a history of cache coherence operations at each node. The information is periodically aggregated and checked by a software algorithm running on the processor cores. Dacota is similar to it in that it leverages built-in storage resources and it uses the available processor cores to perform software

checking. However, it differs in the capabilities it provides: while [8] is only capable of validating the implementation of the coherence protocol, Dacota can identify errors in both cache coherence and memory consistency.

8. Conclusions

This work presents Dacota, a novel solution for high-coverage post-silicon validation of memory ordering in CMP systems. When enabled by the verification team, Dacota stores sequence information about issued memory operations, periodically aggregating this information to perform a software-based policy validation. The validation algorithm is implemented purely in software to minimize the area impact of our solution and executes on existing processor resources. Leveraging approximately 6 orders of magnitude performance advantage over pre-silicon simulation, Dacota's post-silicon approach is able to offer significantly higher coverage compared to pre-silicon approaches. The average performance overhead of our solution (compared to execution with error detection disabled) is 26% for SPLASH2 benchmarks. Moreover, through reuse of preexisting hardware resources, Dacota is able to incur an area penalty of less than 0.01% for a commercial design, such as the OpenSPARC T1 system, and over three orders of magnitude area advantage over popular runtime solutions.

We found that Dacota is effective in detecting subtle consistency and coherence bugs, showing its promise as a solution to the problem of validating the order of memory operations in CMP systems. Furthermore, Dacota enables post-silicon debugging support, providing invaluable information to the validation team. Before shipment, the checking functionality of our solution can be disabled, completely eliminating performance degradation to the end user.

References

[1] *Intel Core 2 Duo and Intel Core 2 Solo Processor for Intel Centrino Duo Processor Technology Specification Update*, Sept. 2007.

[2] D. Abts, S. Scott, and D. Lilja. So many states, so little time: Verifying memory coherence in the Cray X1. In *Proc. of IPDPS*, Apr. 2003.

[3] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 processor: A 64-core soc with mesh interconnect. In *Proc. of ISSCC*, Feb. 2008.

[4] D. Burger and T. Austin. The SimpleScalar toolset, version 3.0. http://simplescalar.com.

[5] M. Bushnell and V. Agrawal. *Essentials in Electronic Testing*. Springer, 2000.

[6] H. Cain, M. Lipasti, and R. Nair. Constraint graph analysis of multithreaded programs. In *Proc. of PACT*, Sept. 2003.

[7] K. Chen, S. Malik, and P. Patra. Runtime validation of memory ordering using constraint graph checking. In *Proc. HPCA*, February 2008.

[8] A. DeOrio, A. Bauserman, and V. Bertacco. Post-silicon verification for cache coherence. In *Proc. ICCD*, 2008.

[9] D. Dill, A. Drexler, A. Hu, and C. Yang. Protocol verification as a hardware design aid. In *Proc. ICCD*, 1992.

[10] S. German. Formal design of cache memory protocols in IBM. *Formal Methods in System Design*, 22(2), 2003.

[11] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1968.

[12] 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*, 42(1), Jan. 2007.

[13] 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*, 35(2), Feb 2002.

[14] 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. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[15] A. Meixner and D. Sorin. Dynamic verification of memory consistency in cache-coherent multithreaded computer architectures. In *Proc. DSN*, pages 73–82, 2006.

[16] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. In *Proc. ISCA*, pages 284–295, 2005.

[17] F. Pong and M. Dubois. Verification techniques for cache coherence protocols. *ACM Computing Surveys*, 29(1), 1997.

[18] M. Prvulovic. CORD: cost-effective (and nearly overhead-free) order-recording and data race detection. *Proc. HPCA*, pages 232–243, Feb. 2006.

[19] D. Shasha and M. Snir. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.*, 10(2):282–312, 1988.

[20] J. Siek and L.-Q. Lee. BOOST graph library. www.boost.org/doc/libs/release/libs/graph.

[21] D. Sorin, M. Martin, M. Hill, and D. Wood. Safetynet: improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *Proc. ISCA*, 2002.

[22] G. J. van Rootselaar and B. Vermeulen. Silicon debug: Scan chains alone are not enough. In *Proc. ITC*, 1999.

[23] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, A. Singh, T. Jacob, S. Jain, V. Erraguntla, C. Roberts, Y. Hoskote, N. Borkar, and S. Borkar. An 80-tile sub-100-w teraflops processor in 65-nm cmos. *IEEE Journal of Solid-State Circuits*, 43(1):29–41, Jan 2008.

[24] L. Whetsel. An IEEE 1149.1 based logic/signature analyzer in a chip. In *Proc. ITC*, 1991.

[25] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The SPLASH-2 programs: characterization and methodological considerations. In *Proc. ISCA*, 1995.

[26] D. Wood, G. Gibson, and R. Katz. Verifying a multiprocessor cache controller using random test generation. *IEEE Design & Test*, 7(4), 1990.

[27] M. Xu, R. Bodik, and M. D. Hill. A "flight data recorder" for enabling full-system multiprocessor deterministic replay. In *Proc. ISCA*, pages 122–135, 2003.