VERIFICATION AND PERFORMANCE OF THE DENOVO CACHE COHERENCE PROTOCOL

BY

RAKESH KOMURAVELLI

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2010

Urbana, Illinois

Adviser:

Professor Sarita Adve

ABSTRACT

With the advent of multicores, parallel programming has gained a lot of importance. For parallel programming to be viable for the predicted hundreds of cores per chip, shared memory programming languages and environments must evolve to enforce disciplined practices like "determinism-by-default semantics" and ban "wild shared-memory behaviors" like arbitrary data races and potential non-determinism everywhere. This evolution can not only benefit software development, but can also greatly reduce the complexity in hardware. DeNovo is a hardware architecture designed from the ground up to exploit the opportunities exposed by such disciplined software models to make the hardware much simpler and efficient at the same time.

This thesis describes an effort to formally verify and evaluate the DeNovo cache coherence protocol. By using a model checking tool, we uncovered three bugs in the protocol implementation which had not been found either in the testing phase or in the simulation runs. All of these bugs were caused by errors in translating the high level description into the implementation. Surprisingly, we also found six bugs in a state-of-the-art implementation of the widely used MESI protocol. Most of these bugs were hard to analyze and took several days to fix. We provide quantitative evidence that DeNovo is a much simpler protocol by showing that the DeNovo protocol has about 15X fewer reachable states when compared to MESI when using the Murphi model checking tool for verification. This translates to about 20X difference in the runtime of the tool. Finally, we show that this simplicity of the DeNovo protocol does not compromise performance for the applications we evaluated. On the contrary, for some applications, DeNovo achieves up to 67% reduction in memory stall time and up to 70% reduction in network traffic when compared to MESI.

To mom, dad and Anu

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Sarita V. Adve, for her guidance and support during my research. I am really fortunate to have an advisor with great motivation, enthusiasm and immense knowledge. I hope to become a better researcher under her guidance in the course of my Ph.D. study.

The development of the DeNovo coherence protocol and its performance evaluation was in collaboration with Byn Choi and Hyojin Sung and I am greatly thankful to them. This thesis is based on a paper submitted recently for publication [22]. Byn Choi led the implementation of the word-based protocol, Hyojin Sung led the performance evaluation of some of the applications and I led the implementation of the line-based protocol and verification of the word-based protocol.

I am also greatly thankful to other collaborators of the DeNovo project, Vikram S. Adve, Robert Bocchino, Nima Honarmand, Robert Smolinski and Nick Carter, for their valuable inputs during the course of this project.

I would like to thank Ching-Tsun Chou from Intel for his insights and help with the verification of the DeNovo protocol.

I am also thankful to Sebastian Burckhardt from Microsoft for providing the source code for his project on verifying the Token coherence protocol which greatly helped me in learning the Murphi tool for verification.

I would also like to acknowledge that my work has been supported by the Intel and Microsoft funded Universal Parallel Computing Research Center at Illinois.

Most importantly, I would like to thank all my friends for their constant support. In particular, I deeply appreciate Thushara Chakkath, Amit Jain and Aditya Agrawal for their care and keeping me sane which helped me overcome setbacks and stay focused on my graduate study.

Finally, I am indebted to my parents and my younger brother for their understanding and encouragement to do the best in all matters of life.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

With the advent of multicores and with the predictions of hundreds of cores per chip in about a decade [16], parallel programming has gained a lot of importance. Shared-memory is arguably the most widely used general-purpose multicore parallel programming model. While shared-memory provides the advantage of a global address space, it is known to be difficult to program, debug, and maintain [39]. Specifically, unstructured parallel control, data races, and ubiquitous non-determinism make programs difficult to understand, and sacrifice safety, modularity, and composability. At the same time, designing performance-, power-, and complexity-scalable hardware for such a software model remains a major challenge. Current designs for large-scale shared-memory systems rely on directory-based cache coherence protocols for scalability [40], which are notoriously complex [2] and hard to scale and an active area of research [58, 29, 43, 53, 44]. More fundamentally, a satisfactory definition of memory consistency semantics (i.e., specification of what value a shared-memory read should return) for such a model has proven elusive, and a recent paper makes the case for rethinking programming languages and hardware to enable usable memory consistency semantics [4].

The above problems have led some researchers to promote abandoning shared-memory altogether (e.g., [39]). An alternative view is that these problems are not inherent to a global address space paradigm, but instead occur due to undisciplined programming models that allow arbitrary reads and writes for implicit and unstructured communication and synchronization. This results in "wild shared-memory" behaviors with unintended data races and non-determinism and implicit side effects that make programs hard to understand, debug, and maintain. The same phenomena result in complex hardware that must assume that any memory access may trigger communication, and performance- and power-inefficient hardware that is unable to exploit communication patterns known to the programmer but obfuscated by the programming model.

There has been much recent software work on disciplined shared-memory programming models with

explicit and structured communication and synchronization to address the above problems for both deterministic and non-deterministic algorithms [6]; e.g., Ct [25], CnC [17], Cilk++ [12], Galois [37], SharC [9], Kendo [48], Prometheus [8], Grace [10], Axum [27], and Deterministic Parallel Java (DPJ) [15, 14].

1.1 Software-Hardware Co-Design

DeNovo is a hardware architecture designed from the ground up assuming a disciplined software, aiming for a more performance-, power-, and complexity-scalable hardware. In this thesis, we currently focus on deterministic codes for three reasons: (1) there is a growing view that deterministic algorithms will be common, at least for client-side computing [6]; (2) focusing on these codes allows us to investigate the "best case," i.e., the potential for gains from exploiting strong discipline; and (3) these investigations will form a basis on which we develop the extensions needed for other classes of codes in the future; in particular, extensions to support disciplined non-determinism as well as legacy software and programming models using "wild shared memory." Synchronization mechanisms involve races and are used in all classes of codes; here, we assume special techniques to implement them (e.g., hardware barriers, queue based locks, etc.) and postpone their detailed handling to future work.

We use Deterministic Parallel Java (DPJ) [15] as an exemplar of the emerging class of deterministic-by-default languages (Chapter 2), and use it to explore how hardware can take advantage of strong disciplined programming features. Specifically, we use three features of DPJ that are also common to several other projects: (1) structured parallel control; (2) data-race-freedom, and guaranteed deterministic semantics unless the programmer explicitly requests non-determinism (called determinism-by-default); and (3) explicit specification of the effects of shared-memory accesses; e.g., which (possibly non-contiguous) regions of memory will be read or written in a parallel section.

Most of the disciplined models projects cited above also enforce a requirement of structured parallel control (e.g., a nested fork join model, pipelining, etc.), which is much easier to reason about than arbitrary (unstructured) thread synchronization. Most of these, including all but one of the commercial systems, *guarantee* the absence of data races for programs that type-check. Coupled with structured parallel control, the data-race-freedom property guarantees determinism for several of these systems. We also note that data races are prohibited (although not checked) by existing popular languages as well; the emerging C++ and C memory models do not provide *any* semantics with any data race (benign or otherwise) and Java

provides extremely complex and weak semantics for data races only for the purposes of ensuring safety. The information about side effects of concurrent tasks is also available in other disciplined languages, but in widely varying (and sometimes indirect) ways. Once we understand the types of information that is most valuable, our future work includes exploring how the information can be extracted from programs in other languages.

There are several ways in which the above language features can be used to drive more efficient and simpler memory hierarchy designs. In this thesis, we focus on the cache coherence protocol, a central feature of multicore memory hierarchy design. Although current directory-based protocols are more scalable than snooping protocols, they suffer from several limitations:

Performance and power overhead: They incur latency and traffic overhead impacting both performance and power; e.g., they require invalidation and acknowledgement messages (which are strictly overhead) and require indirection through the directory for cache-to-cache transfers.

Verification complexity: They are notoriously complex and difficult to verify since they require dealing with subtle races and many transient states [46, 26]. Moreover, their fragility often discourages implementors from adding optimizations to previously verified protocols. Thus, although researchers have proposed several optimizations to mitigate the invalidation, acknowledgement, and indirection overheads discussed above, the optimizations usually require dealing with new states and races, further exacerbating the verification complexity.

State overhead: Directory protocols incur high directory storage overhead to track sharer lists. Several optimized directory organizations have been proposed, but also require considerable overhead and/or excessive network traffic and/or complexity. These protocols also require several coherence state bits due to the large number of protocol states (e.g., ten bits in [53]). This state overhead is amortized by tracking coherence at the granularity of cache lines. This can result in performance/power anomalies and inefficiencies when the granularity of sharing is different from a contiguous cache line (e.g., with false sharing).

Researchers continue to propose new directory organizations and protocol optimizations to address one or more of the above limitations [58, 29, 43, 38, 1]; however, to our knowledge, all of these approaches degrade one or more of complexity, performance/power, and storage overhead. More recently, there have been projects that do away with coherent caches altogether, most notably the 48 core Intel Single-Chip Cloud Computer [32], pushing significant complexity in the programming model.

This thesis presents, verifies and evaluates the DeNovo protocol that targets all the above limitations of the directory protocols for large core counts, driven by a disciplined shared-memory programming model.

1.2 Contributions of this Thesis

The DeNovo hardware takes advantage of the language-level annotations designed for concurrency safety and efficiently represents and uses them in hardware for improving complexity and efficiency. Two simple insights underlie our design. First, structured parallel control and knowing which memory regions will be read or written enable a cache to take responsibility for invalidating its own stale data. Such self-invalidations remove the need for a hardware directory to track sharer lists and to send invalidations and acknowledgements on writes. Second, data-race-freedom eliminates concurrent conflicting accesses and corresponding transient states in coherence protocols, eliminating a major source of complexity. The simple DeNovo protocol assumes equal address, communication and coherence granularity. This is the granularity at which data-race-freedom is ensured, which is a word for our applications.

The specific contributions of this thesis are as follows[1]:

Protocol verification: In this thesis, we verified the simple DeNovo protocol for correctness by specifying an abstract model in the Murphi model checking tool [24]. In the process, we discovered three bugs in the protocol implementation which had not been found either in the testing phase or in the simulation runs. All of these bugs were very simple to fix and turned out to be mistakes in translating the high level description of the protocol into the implementation (i.e., their solutions were already present in our internal high level description of the protocol).

We also used the Murphi model checking tool to compare the complexity of the simple DeNovo protocol with that of a conventional MESI protocol. For MESI, we used the implementation available in the Wisconsin GEMS simulation suite [42] as an example of a (publicly available) state-of-the-art, mature implementation. Although the GEMS protocol has been used by many researchers in many publications, we found six bugs. Most of these bugs involved subtle data races and took several days to debug and fix. In contrast, for the much less mature DeNovo, the three bugs that we discovered were very simple to fix. After the bugs were fixed, MESI showed 15X more reachable states compared to DeNovo, with a runtime of 173

[1]The work discussed in this thesis is done in collaboration with Byn Choi and Hyojin Sung. Byn Choi led the implementation of the simple DeNovo protocol and Hyojin Sung led the performance evaluation of some of the benchmarks. This thesis is based on a paper recently submitted for publication. A high level overview of the project was presented at HotPar 2010 [20].

seconds for MESI and 8.66 seconds for DeNovo. This provides quantitative evidence of the simplicity of the DeNovo protocol.

Practical DeNovo protocol: We also discuss the practical DeNovo protocol which enhances the simple DeNovo protocol to operate on a larger communication and address granularity, typically a cache line size from conventional protocols. We call this the line-based protocol. With this implementation of the line-based DeNovo protocol, it is now possible to make realistic comparisons with the standard MESI protocol which is run on current systems with conventional cache line sizes.

The performance results indicate that the execution time of the DeNovo line-based protocol is comparable or better than the MESI protocol. For some of the applications, false sharing and inclusivity advantages reduced memory stall time up to 67% and network traffic up to 70%. These benefits translate directly into power savings since they come from lower traffic and miss rates. To demonstrate the advantages of the line-based protocol, we also provide performance results comparing it with the simple protocol.

Overall, our results show that rethinking hardware from the ground up, driven by a disciplined programming environment, can help greatly reduce the complexity of the hardware with little or no effect on performance.

CHAPTER 2

BACKGROUND: DETERMINISTIC PARALLEL JAVA

DPJ is an extension to Java that enforces *deterministic-by-default* semantics via compile-time type checking [15, 14]. Using Java is not essential; similar extensions for C++ are underway. DPJ provides a new type and effect system for expressing important patterns of deterministic and non-deterministic parallelism in imperative, object-oriented programs. Non-deterministic behavior can only be obtained via certain explicit constructs. For a program that does not use such constructs, DPJ guarantees that if the program is well-typed, any two parallel tasks are *non-interfering*, i.e., do not have conflicting accesses. (Two accesses conflict if they reference the same location and at least one is a write.)

DPJ's parallel tasks are iterations of an explicitly parallel `foreach` loop or statements within a `cobegin` block; they synchronize through an implicit barrier at the end of the loop or block. Parallel control flow thus follows a scoped, nested, fork-join structure, which simplifies the use of explicit coherence actions in DeNovo at fork/join points. This structure defines a natural ordering of the tasks, as well as an obvious definition (omitted here) of when two tasks are "concurrent". It implies an obvious sequential equivalent of the parallel program (`for` replaces `foreach` and `cobegin` is simply ignored). DPJ guarantees that the result of a parallel execution is the same as the sequential equivalent.

In a DPJ program, the programmer assigns every object field or array element to a named "*region*" and annotates every method with read or write "*effects*" summarizing the regions read or written by that method. The compiler checks that (i) all program operations are type safe in the region type system; (ii) a method's effect summaries are a superset of the actual effects in the method body; and (iii) that no two parallel statements interfere. the effect summaries on method interfaces allow all these checks to be performed without interprocedural analysis.

For DeNovo, the effect information tells the hardware what fields will be read or written in each parallel "phase" (`foreach` or `cobegin`). This enables efficient software-controlled coherence mechanisms discussed in the following sections.

DPJ has been evaluated on a wide range of deterministic parallel programs. The results show that DPJ can express a wide range of realistic parallel algorithms; that its type system features are useful for such programs; and that well-tuned DPJ programs exhibit good performance [15].

CHAPTER 3

DENOVO COHERENCE AND CONSISTENCY

A shared-memory design must first and foremost ensure that a read returns the correct value, where the definition of "correct" comes from the memory consistency model. Modern systems divide this responsibility between two parts: (i) cache coherence, which is usually defined as ensuring that writes to the *same* location appear in the same total order to all cores, and (ii) various memory ordering constraints, which impose needed ordering among accesses to different (and the same) data. These are arguably among the most complex and hard to scale aspects of shared-memory hierarchy design. Disciplined models enable mechanisms that are potentially simpler and more efficient to achieve this function.

The deterministic parts of our software have semantics corresponding to those of the equivalent sequential program. A read should therefore simply return the value of the last write to the same location that is before it in the deterministic sequential program order. This write either comes from the reader's own task (if such a write exists) or from a task preceding the reader's task, since there can be no conflicting accesses concurrent with the reader (two accesses are concurrent if they are from concurrent tasks). In contrast, conventional (software-oblivious) cache coherence protocols assume that writes and reads to the same location can happen concurrently, resulting in significant complexity and inefficiency.

To describe the DeNovo protocol, we first assume that the coherence granularity and address/communication granularity are the same. That is, the data size for which coherence state is maintained is the same as the data size corresponding to an address tag in the cache and the size communicated on a demand miss. This is typically the case for MESI protocols, where the cache line size (e.g., 64 bytes) serves as the address, communication, and coherence granularity. For DeNovo, the coherence granularity is dictated by the granularity at which data-race-freedom is ensured – a word for our applications. Thus, this assumption constrains

the cache line size. We henceforth refer to this as the word based version of our protocol. We relax this assumption in Section 3.2, where we decouple the address/communication and coherence granularities. We also enable sub-word coherence granularity in Section 3.3.

Without loss of generality, throughout we assume private and writeback L1 caches, a shared last-level on-chip L2 cache inclusive of only the modified lines in any L1, a single (multicore) processor chip system, and no task migration. The ideas here extend in an obvious way to deeper hierarchies with multiple private and/or cluster caches and multichip multiprocessors, and task migration can be accommodated with appropriate self-invalidations before migration. Below, we use the term *phase* to refer to the execution of all tasks created by a single parallel construct (foreach or cobegin).

3.1 DeNovo with Equal Address/Communication and Coherence Granularity

Broadly, coherence protocols can be classified as snooping or directory (or hybrids). Snooping protocols require broadcasts (or ordered networks), which makes them unscalable. Directory protocols avoid the use of broadcast through a level of indirection, but have other limitations as described in Chapter 1. DeNovo eliminates the drawbacks of conventional directory protocols as follows.

No directory storage or write invalidation overhead: In conventional directory protocols, a write acquires ownership of a line by invalidating all other copies, to ensure later reads get the updated value. The directory achieves this by tracking all current sharers and invalidating them on a write, incurring significant storage and invalidation traffic overhead. In particular, straightforward bit vector implementations of sharer lists are not scalable (a 1,000 core processor would require 1,000 bits for each unique cache line on chip). Several techniques have been proposed to reduce this overhead, but typically pay a price in significant increase in complexity and/or incurring unnecessary invalidations when the directory overflows. DeNovo eliminates these overheads by removing the need for ownership on a write. Data-race-freedom ensures there is no other writer or reader for that line in this parallel phase. DeNovo need only ensure that (i) outdated cache copies are invalidated before the next phase, and (ii) readers in later phases know where to get the new data.

For (i), each cache simply uses the known write effects of the current phase to invalidate its outdated data before the next phase begins. The compiler inserts self-invalidation instructions for each region with these

write effects (we describe how regions are conveyed and represented below). Each L1 cache invalidates its data that belongs to these regions with the following exception. Any data that the cache has read or written in this phase is known to be up-to-date since there cannot be concurrent writers. We therefore augment each line with a "touched" bit that is set on a read. A self-invalidation instruction does not invalidate a line with a set touched bit or that was last written by this core (indicated by the `registered` state as discussed below); the instruction resets the touched bit in preparation for the next phase.

For (ii), DeNovo requires that on a write, a core register itself at (i.e., inform) the shared L2 cache. The L2 data banks serve as the registry. An entry in the L2 data bank either keeps the identity of an L1 that has the up-to-date data (`registered` state) or the data itself (`valid` state) – a data bank entry is never required to keep both pieces of information since an L1 cache registers itself in precisely the case where the L2 data bank does not have the up-to-date data. Thus, DeNovo entails *zero overhead for directory (registry) storage*. Henceforth, we use the term L2 cache and registry interchangeably.

We also note that because the L2 does not need sharer lists, it is natural to not maintain inclusion in the L2 for lines that are not registered by another L1 cache – the registered lines do need space in the L2 to track the L1 id that registered them.

No transient states: The DeNovo protocol has three states in the L1 and L2 – `registered`, `valid`, and `invalid` – with obvious meaning. (The touched bit mentioned above is local to its cache and irrelevant to external coherence transactions.) Although textbook descriptions of conventional directory protocols also describe 3 to 5 states (e.g., MSI) [30], it is well-known that they contain many hidden transient states due to races, making them notoriously complex and difficult to verify [2, 54, 57]. For example, considering a simple MSI protocol, a cache may request ownership, the directory may forward the request to the current owner, and another cache may request ownership while all of these messages are still outstanding. Proper handling of such a race requires introduction of transient states into the cache and/or directory transition tables.

DeNovo, in contrast, is a true 3-state protocol with *no transient states*, since it assumes race-free software. The only possible races are related to writebacks. As discussed below, these races either have limited scope or are similar to those that occur in uniprocessors. They can be handled in straightforward ways, without transient protocol states (described below).

The full protocol: Table 3.1 shows the L1 and L2 state transitions and events for the full protocol. Note the

	$Read_i$	$Write_i$	$Read_k$	$Register_k$	Response for $Read_i$	Writeback
$Invalid$	Update tag; Read miss to L2; Writeback if needed	Go to $Registered$; Reply to core i; Register request to L2; Write data; Writeback if needed	Nack to core core k	Reply to core k	If tag match, go to $Valid$ and load data; Reply to core i	Ignore
$Valid$	Reply to core i	Go to $Registered$; Reply to core i; Register request to L2	Send data to core k	Go to $Invalid$; Reply to core k	Reply to core i	Ignore
$Registered$	Reply to core i	Reply to core i	Reply to core k	Go to *Invalid*; Reply to core k	Reply to core i	Go to *Valid*; Writeback

(a) **L1 cache of core i. $Read_i$ = read from core i, $Read_k$ = read from another core k (forwarded by the registry).**

	Read miss from core i	Register request from core i	Read response from memory for core i	Writeback from core core i
$Invalid$	Update tag; Read miss to memory; Writeback if needed	Go to $Registered_i$; Reply to core i; Writeback if needed	If tag match, go to $Valid$ and load data; Send data to core i	Reply to core i; Generate reply for pending writeback to core i
$Valid$	Data to core i	Go to $Registered_i$; Reply to core i	X	X
$Registered_j$	Forward to core j; Done	Forward to core j; Done	X	if i==j go to $Valid$ and load data; Reply to core i; Cancel any pending Writeback to core i

(b) **L2 cache**

Table 3.1 DeNovo cache coherence protocol for (a) private L1 and (b) shared L2 caches. Self-invalidation and touched bits are not shown here since these are local operations as described in the text. Request buffers (MSHRs) are not shown since they are similar to single core systems.

lack of transient states in the caches.

Read requests to the L1 (from L1's core) are straighforward – accesses to valid and registered state are hits and accesses to invalid state generate miss requests to the L2. A read miss does not have to leave the L1 cache in a pending or transient state – since there are no concurrent conflicting accesses (and hence no invalidation requests), the L1 state simply stays invalid for the line until the response comes back.

For a write request to the L1, unlike a conventional protocol, there is no need to get a "permission-to-write" since this permission is implicitly given by the software race-free guarantee. Thus, a writer can always immediately update the line in its L1 cache. If the cache does not already have the line registered, it must issue a registration request to the L2 to notify that it has the current up-to-date copy of the line and set the registry state appropriately. Since there are no races, the write can *immediately* set the state of the cache to registered, without waiting for the registration request to complete. Thus, *there is no transient or pending state for writes either*.

The pending read miss and registration requests are simply monitored in the processor's request buffer, just like those of other reads and writes for a single core system. Thus, although the request buffer technically has transient states, these are not visible to external requests – external requests only see stable cache states. The request buffer also ensures that its core's requests to the same location are serialized to respect uniprocessor data dependences, similar to a single core implementation (e.g., with MSHRs). The memory model requirements are met by ensuring that all pending requests from the core complete by the end of this parallel phase (or at least before the next conflicting access in the next parallel phase).

The L2 transitions are also straightforward except for writebacks which require some care. A read or registration request to data that is invalid or valid at the L2 invokes the obvious response. For a request for data that is registered by an L1, the L2 forwards the request to that L1 and updates its registration id if needed (if the request is a new registration). For a forwarded registration request, the L1 always acknowledges the requestor and invalidates its own copy. If the copy is already invalid due to a concurrent writeback by the L1, the L1 simply acknowledges the original requestor and the L2 ensures that the writeback is not accepted (by noting that it is not from the current registrant). For a forwarded read request, the L1 supplies the data if it has it. If it no longer has the data (because it issued a concurrent writeback), then it sends a negative acknowledgement (nack) to the original requestor, which simply resends the request to the L2. Because of race-freedom, there cannot be another concurrent write, and so no other concurrent writeback, to the line.

Thus, the nack eventually finds the line in the L2, without danger of any deadlock or livelock. The only somewhat less straightforward interaction is when both the L1 and L2 caches want to writeback the same line concurrently, but this race also occurs in uniprocessors.

Conveying and representing regions in hardware: A key research question is how to represent regions in hardware for self-invalidations. Language-level regions are usually much more fine-grain than may be practical to support in hardware. For example, when a parallel loop traverses an array of objects, the compiler may need to identify (a field of) *each object* as being in a distinct region in order to prove the absence of conflicts. For the hardware, however, such fine distinctions would be expensive to maintain. Fortunately, we can coarsen language-level regions to a much smaller set without losing functionality in hardware. The key insight is as follows. For self-invalidations, we need regions to identify which data could have been written in the current phase. It is not important to distinguish which core wrote which data. In the above example, we can thus treat the entire array of objects as one region.

Alternately, if only a subset of the fields in each object in the above array is written, then this subset aggregated over all the objects collectively forms a hardware region. Thus, just like software regions, hardware regions need not be contiguous in memory – they are essentially an assignment of a color to each heap location (with orders of magnitude fewer colors in hardware than software). Hardware regions are not restricted to arrays either. For example, in a traversal of the spatial tree in an n-body problem, the compiler distinguishes different tree nodes (or subsets of their fields) as separate regions; the hardware can treat the entire tree (or a subset of fields in the entire tree) as an aggregate region. Similarly, hardware regions may also combine field regions from different aggregate objects (e.g., fields from an array and a tree may be combined into one region).

The compiler can easily *summarize* program regions into coarser hardware regions as above and insert appropriate self-invalidation instructions. The only correctness requirement is that the self-invalidated regions must cover all write effects for the phase. For performance, these regions should be as precise as possible. For example, fields that are not accessed or read-only in the phase should not be part of these regions. Similarly, multiple field regions written in a phase may be combined into one hardware region for that phase, but if they are not written together in other phases, they will incur unnecessary invalidations.

During final code generation, the memory instructions generated can convey the region name of the address being accessed to the hardware; since DPJ regions are parameterizable, the instruction needs to point

to a hardware register that is set at runtime (through the compiler) with the actual region number. When the memory instruction is executed, it conveys the region number to the core's cache. A straightforward approach is to store the region number with the accessed data line in the cache. Now a self-invalidate instruction invalidates all data in the cache with the specified regions that is not `touched` or `registered`.

The above implementation requires storing region bits along with data in the L1 cache and matching region numbers for self-invalidation. A more conservative implementation can reduce this overhead. At the beginning of a phase, the compiler conveys to the hardware the set of regions that need to be invalidated in the *next* phase – this set can be conservative, and in the worst case, represent all regions. Additionally, we replace the region bits in the cache with one bit: `keepValid`, indicating that the corresponding data need not be invalidated until the end of the *next* phase. On a miss, the hardware compares the region for the accessed data (as indicated by the memory instruction) and the regions to be invalidated in the next phase. If there is no match, then `keepValid` is set. At the end of the phase, all data not `touched` or `registered` are invalidated and the `touched` bits reset as before. Further, the identities of the `touched` and `keepValid` bits are swapped for the next phase. This technique allows valid data to stay in cache through a phase even if it is not `touched` or `registered` in that phase, without keeping track of regions in the cache. The concept can be extended to more than one such phase by adding more bits and if the compiler can predict the self-invalidation regions for those phases.

Example: Figure 3.1 illustrates the above concepts. Figure 3.1(a) shows a code fragment with parallel phases accessing an array, S, of structs with three fields each, X, Y, and Z. The X (respectively, Y and Z) fields from all array elements form one DeNovo region. The first phase writes the region of X and self-invalidates that region at the end. Figure 3.1(b) shows, for a two core system, the L1 and L2 cache states at the end of Phase 1, assuming each core computed one contiguous half of the array. The computed X fields are `registered` and the others are invalid in the L1's while the L2 shows all X fields registered to the appropriate cores.

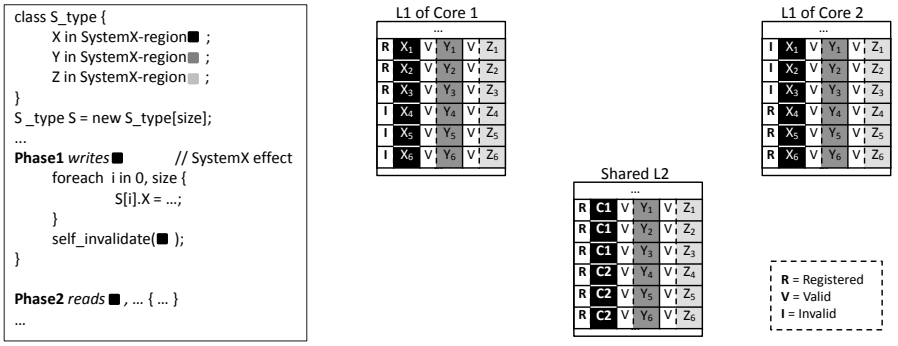

Figure 3.1 (a) Code with DeNovo regions and self-invalidations and (b) cache state after phase 1 self-invalidations at the beginning of phase 2. X_i represents $S[i].X$. Ci in L2 cache means the word is registered with Core i. Initially, all lines in the caches are in `valid` state.

3.2 DeNovo with Address/Communication Granularity > Coherence Granularity

To decouple the address/communication and coherence granularity, our key insight is that any data marked `valid` or `registered` can be copied over to any other cache in `valid` state (but not as `touched`). Additionally, for even further optimization, we make the observation that this transfer can happen without going through the registry/L2 at all (because the registry does not track sharers). Thus, no serialization at a directory is required. When (if) this copy of data is accessed through a demand read, it can be immediately marked `touched`. The presence of a demand read means there will be no concurrent write to this data, and so it is indeed correct to read this value (`valid` state) and furthermore, the copy will not need invalidation at the end of the phase (`touched` copy). The above copy does not incur false sharing (nobody loses ownership) and, if the source is the non-home node, it does not require extra hops to a directory.

With the above insight, we can easily enhance the word-based DeNovo protocol from the previous section to operate on a larger communication and address granularity; e.g., a typical cache line size from conventional protocols. However, we still maintain coherence state at the granularity at which the program guarantees data race freedom; e.g., a word. On a demand request, the cache servicing the request can send an entire cache line worth of data, albeit with some of the data marked invalid (those that are `invalid` or `registered` in the L2). The requestor can then merge the valid words in the response message with its copy of the cache line (if it has one), marking all of those words as `valid` (but not `touched`).

Note that if the L2 has a line valid in the cache, then an element of that line can be either valid (and hence sent to the requestor) or registered (and hence not sent). Thus, for the L2, it suffices to keep just one

coherence state bit at the finer (e.g., word) granularity with a line-wide valid bit at the line granularity.[1] As before, the id of the registered core is stored in the data array of the registered location.

This is analogous to sector caches – cache space allocation (i.e., address tags) is at the granularity of a line but there may be some data within the line that is not valid (indicated by the smaller granularity coherence state). This combination effectively allows exploiting spatial locality without any false sharing, similar to multiple writer protocols of software distributed shared memory systems [34].

3.3 Flexible Coherence Granularity

Although the applications we studied did not have any data races at word granularity, this is not necessarily true of all applications. Data may be shared at byte granularity, and two cores may incur conflicting concurrent accesses to the same word, but for different bytes. A straightforward implementation would require coherence state at the granularity of a byte, which would be significant storage overhead (1 bit per byte at the L2 and several bits at the L1, see Section 3.4). [2] Although previous work has suggested using byte based granularity for state bits in other contexts [41], we would like to minimize the overhead.

We focus on the overhead in the L2 cache since it is typically much larger (e.g., 4X to 8X times larger) than the L1. We observe that byte granularity coherence state is needed only if two cores incur conflicting accesses to different bytes in the same word in the same phase. Our approach is to make this an infrequent case, and then handle the case correctly albeit at potentially lower performance. In disciplined languages, task allocation to cores can be made under compiler/runtime control. Since the compiler is aware of the granularity of regions, it can orchestrate task scheduling granularity so that byte granularity regions are allocated to tasks at word granularities when possible. (Note that even in the MESI protocol, the above byte based sharing can result in undesirable false sharing related performance anomalies).

For cases where the compiler (or programmer) cannot avoid word granularity data races, we require the compiler to indicate such regions to the hardware. Hardware uses word granularity coherence state. For byte-shared data such as the above, it "clones" the cache line containing it in four places: place i contains the ith byte of each word in the original cache line. If we have at least four way associativity in the L2

[1] This requires that if registration request misses in the L2, then the L2 obtain the full line from main memory. Our implementation mistakenly does not bring the line in; however, we do not believe this affects performance or complexity and will fix it in the future.

[2] The upcoming C and C++ memory models and the Java memory model do not allow data races at byte granularity; therefore, we also do not consider a coherence granularity lower than that of a byte.

cache (usually the case), then we can do the cloning in the same cache set. The tag values for all the clones will be the same but each clone will have a different byte from each word, and each byte will have its own coherence state bit to use (essentially the state bit of the corresponding word in that clone). This allows hardware to pay for coherence state at word granularity while still accomodating byte granularity coherence when needed, albeit with potentially poorer cache utilization in those cases.

3.4 Storage Overhead

We next compare the storage overhead of DeNovo to other common directory configurations.

DeNovo overhead: At the L1, DeNovo needs state bits at the word granularity. We have three states and one touched bit (total of 3 bits). We also need region related information. In our applications, we later show that we need at most 20 hardware regions – 5 bits. These can be replaced with 1 bit by using the optimization of the `keepValid` bit discussed in Section 3.1. Thus, we need a total of 4 to 8 bits per 32 bits or 64 to 128 bits per L1 cache line. At the L2, we just need one valid and one dirty bit per line (per 64 bytes) and one bit per word, for a total of 18 bits per 64 byte L2 cache line or 3.4%. If we assume L2 cache size of 8X that of L1, then the L1 overhead is 1.56% to 3.12% of the L2 cache size.

In-cache full map directory. We conservatively assume 5 bits per 64 byte cache line at the L1 (4 bits for 11 stable+transient states and 1 dirty bit). With full map directories, each L2 line needs a bit per core for the sharer list. This implies that DeNovo overhead for just the L2 is better for more than a 13 core system. If the L2 cache size is 8X that of L1, then the total L1+L2 overhead of DeNovo is better at greater than about 21 (with `keepValid`) to 30 cores.

Duplicate tag directories. To reduce directory overhead without losing information, L1 tags can be duplicated at the L2. However, this requires a very high associative lookup; e.g., 64 cores with 4 way L1 associativity requires a 256 way associative lookup. As discussed in [58], this design is not scalable to even low tens of cores systems.

Tagless directories and sparse directories. The tagless directories work uses Bloom filter based directory organization [58]. Their directory storage requirement appears to be about 3% to over 5% of L1 storage for core counts ranging from 64 to 1K cores (the numbers are gleaned from a log scale graph and hence imprescise). This does not include any coherence state overhead from either L1 or L2 which we include in our calculations above. Further, this organization is lossy in that larger core counts require extra invalidations

and protocol complexity.

Many sparse directory organizations have been proposed that can drastically cut directory overhead at the cost of sharer list precision, and so come at a significant performance cost especially at higher core counts [58].

3.5 Related Work

The starting point for our work is that current shared-memory programming models are unsustainable for both software and hardware for the era of mass-scale parallel programming, motivating more disciplined shared-memory models. With such models as drivers, we rethink the cache coherence protocol.

Perhaps philosophically the most closely related line of work has been in the software distributed shared memory literature where the system exploits expected software behavior (data-race-freedom) to allow large granularity communication (virtual pages) without false sharing (e.g., [5, 34, 35, 11, 19, 13]). Many of these techniques rely on heavyweight mechanisms like virtual memory management, and have struggled to find an appropriate high-level programming model. Our work starts with a high-level model that includes both data race freedom guarantees and data sharing information.

Motivated by the complexities and inefficiencies of scalable hardware coherent caches, some work has abandoned these ideas in favor of completely software controlled caches [36, 31] or even no global address space at all, at the cost of significant programming complexity. We claim that we can enjoy the benefits of a coherent global address space, if we move to disciplined programming models for programmer productivity, system simplicity, and scalable efficiency.

CHAPTER 4

PROTOCOL VERIFICATION

Cache coherence protocols are inherently complex. With numerous transient states and hard-to-cover race conditions, it is very difficult to find all the bugs during simulations of the protocols. Hence, formal methods like model checking are often employed to verify their correctness. Model checking is a technique to verify the properties of a system by exhaustive exploration of the state space [23, 52]. McMillan and Schwalbe's seminal work on model checking the Encore Gigamax protocol [45] was the first to apply model checking to verify cache coherence protocols. A general survey of various techniques to verify cache coherence protocols can be found at [51].

Complex systems often exhibit a lot of regularity and symmetry. Ip and Dill developed Murphi [24, 47, 33] which exploits these characteristics by grouping together similar states to verify a reduced state graph instead of the full one. This helps to greatly reduce the amount of time and memory used in verification. One major disadvantage of this tool is that the abstraction used by it is too coarse to prove liveness. Murphi is a widely used tool to formally verify cache coherence protocols; e.g., Sun RMO memory model [49], Sun S3.mp multiprocessor [50], Cray SV2 protocol [3], and Token coherence protocol [18]. In this thesis, we use Murphi for our protocol verification work.

We verified the word-based protocol of DeNovo and MESI. We derived the MESI model from the GEMS implementation and the DeNovo model directly from our implementation. To enable cross-phase interactions in both the protocols, we introduced the notion of phase boundary by modeling it as a sense reversing barrier and modeled data-race-fee guarantee for DeNovo by limiting conflicting accesses. We explain each of these models in detail below.

4.1 Abstract Model

To reduce the amount of time and memory used in verification, we modeled the processors, addresses, data values and regions as scalarsets, a datatype in Murphi, which takes advantage of the symmetry in these entitites while exploring the reachable states. A processor is modeled as an array of cache entries consisting of L1 state information along with protocol specific fields like the region field for DeNovo. L1 state is one of the 3 possible states for DeNovo or one of the 11 possible states for MESI. Similarly, L2 is also modeled as an array of cache entries each with L2 state information and other protocol specific details like sharer list for MESI. L2 state is one of the 3 possible states for DeNovo or one of the 18 possible states for MESI. Memory is just modeled as an array of addresses storing data values.

4.1.1 Data-race-free guarantee for DeNovo

To model the data-race-free guarantee from software for DeNovo, we used an additional data structure called $AccessStatus$. This maintains the current status ($read$, $readshared$, or $written$) and the core id of the last requestor for every address in the model. On any read, if it is the first access to this address, then the $status$ is set to $read$. If the $status$ is already set to $read$ and the requesting core is not the same as the last requestor, then $status$ is set to $readshared$. The $status$ is not modified if it is either $readshared$ or $write$ and the requesting core is the same as the last requestor. On the other hand, if the $status$ is $write$ and the requesting core is not the same as the last requestor, then this access is not generated in the model. Similary, on any write, if it is the first access to this address or if the requesting core is the same as the last requestor, then the $status$ is set to $write$. If the $status$ is either $readshared$ or the requesting core is not the same as the last requestor, then this access is not generated to adhere to the data-race-free guarantee. The $AccessStatus$ data structure is reset for all the addresses at the end of a phase.

4.1.2 Cross phase interactions

We modeled end-of-phase using a sense-reversing barrier implementation. This event can be triggered at any time i.e., with no condition. This event occurs per core and stalls the core preventing any more memory requests until (1) all the pending requests of this core are completed and (2) all other cores reach the barrier. In DeNovo, once a core reaches the barrier, we also modeled the self-invalidations by executing the self-invalidation instruction.

4.1.3 Addressing state space explosion

To keep the number of states explored tractable, as is common practice, we used a single address, single region (only for DeNovo), two data values, two cores with private L1 cache, a unified L2 and an in-cache directory (for MESI). We modeled an unordered full network with separate request and reply links. Both models allow only one request per L1 in the rest of the memory hierarchy. As we modeled only one address, we modeled replacements as unconditional events that can be triggered at any time.

4.2 Invariants

In this section, we discuss the invariants used for both MESI and DeNovo models. A deadlock occurs when all the entities in the system (all L1s and L2) become locked out from making any forward progress. Murphi checks for absence of deadlock by default and we do not have to explicitly specify any invariant for checking that.

4.2.1 MESI invariants

We used a total of four invariants to verify the MESI protocol. These invariants are based on prior work in verification of cache coherence protocols [45, 24].

Empty sharer list in Invalid state. This invariant asserts that the sharer list is empty when $L2$ transitions to $Invalid$ state. This makes sure that there are no $L1$s sharing the line after $L2$ replaces the line.

Empty sharer list in Modified state. This invariant asserts that the sharer list is empty when $L2$ transitions to $Modified$ state (i.e., $L2$ cache entry is modified and not present in any of the L1s). When $L2$ is in $Modified$ state, there is no copy of the line in any of the $L1$s and the sharer list should be empty.

Only one modifiable cache copy. There can not be two modifiable L1 cache copies in the system at the same time. This is a clear violation of cache coherence and this invariant checks whether there are two L1 caches in $Modified$ state for the same line.

Data values consistency. One of the requirements of the MESI protocol is that whenever a cache copy is read only (i.e., either in $Valid$ or $Exclusive$) state, then its value is the same as at memory. This invariant checks if this requirement holds at all points in the protocol.

4.2.2 DeNovo invariants

We modeled five invariants for the DeNovo protocol. As there is no sharer list maintained in the DeNovo protocol, we do not check for the first two invariants of the MESI protocol. The first two invariants of the DeNovo protocol are similar to the last two invariants of the MESI protocol. The last three invariants of the DeNovo protocol are checks on $touched$ bit functionality even though two of these invariants are basic assumptions of DeNovo guaranteed by data-race-freedom from software.

Only one modifiable cache copy. There cannot be two modifiable L1 cache copies in the system at the same time. This is a clear violation of cache coherence and this invariant checks whether there are two L1 caches in $Registered$ state for the same line.

Consistent valid copies. This invariant checks that all valid copies are consistent; i.e., if there are two cache copies in $Valid$ state, then their data values should be same.

No previous access before a write. On a write, this invariant checks that no other cache has the touched bit set to true indicating that there is no previous read or write to this line in another core. This verifies that the touched bit is implemented correctly even if data-race freedom guarantees conflict free accesses.

No previous writes before a read. Similar to the above, on a read, this invariant checks that there is no previous write to the same line by some other core by asserting that the only cache lines that can have the touched bit set to true (for cores other than the requestor) are the ones in $Valid$ state.

Unsetting touched bits. Finally, this invariant checks that all the touched bits are set to false at the end of the phase.

4.3 Extending to Line-based Protocol

The model for MESI remains the same for both word-based and line-based protocols whereas the word-based model of DeNovo requires slight extension. The only additions required are maintaining multiple state bits per line, merging of valid data into the cache whenever a data response is received and an additional counter at the directory to keep track of outstanding L1 writebacks in case of L2 initiated replacement. None of these additions require making changes to the core protocol model and we leave this extension to future work.

4.4 Results

Through model checking, we found three bugs in DeNovo and six bugs including two deadlock scenarios in MESI. Note that DeNovo is much less mature than the GEMS MESI protocol which has been used by many researchers. All the three bugs found in DeNovo were simple to fix and showed mistakes in translating our internal high level specification into the implementation (i.e., their solutions were already present in our internal high level description of the protocol).

In MESI, all the bugs except one of the deadlocks are caused by protocol races between L1 writebacks and other cache events. This other deadlock is caused due to incorrect handling of clean replacement at the L2. Most of these bugs are due to interactions between L1 writebacks and remote reads and writes which do not happen in DeNovo due to the data-race-freedom guarantee. Most of these bugs found in MESI involved subtle data races and took several days to track, debug, and fix.

Each of the bugs found in DeNovo and MESI is described in detail next. In all the descriptions below, we consider a single address, $L1_{P1}$, $L1_{P2}$ and $L2$ indicate the cache lines corresponding to the above address in core P1, core P2, and L2 respectively. We assume an in-cache directory at L2 and hence we use the words $directory$ and $L2$ interchangeably.

4.4.1 DeNovo bugs

Bug 1. The first of the three bugs found in the DeNovo protocol was caused due to not unsetting the dirty bit on replacement of a dirty L2 cache line. To begin with, let us assume that $L2$ is initially in $Valid$ state and the $dirty$ bit is set to true. Then on $L2$ replacement, it transitions to $Invalid$ state and writes back data to memory. But the $dirty$ bit is mistakenly not unset. This bug was found when Murphi tried to replace the line while still in $Invalid$ state as the $dirty$ bit was set to true. On next allocation to the same line, the protocol implementation resets the entire cache data structure where the $dirty$ bit is set to false. But this happens only when the tag does not match, i.e., only when a different address is allocated the same line. For the case where the tag does match, this reset doesn't happen and the $dirty$ bit is incorrectly set to true. This results in unnecessary writebacks to memory on future $L2$ replacements which might well be silent replacements. This is a performance related bug and turned out to be a rare case to hit in our simulation runs.

Bug 2. The second bug is caused because $L2$ initiated writeback and future requests to the same cache line are not serialized. Initially, $L1_{P1}$ is in $Registered$ state and $L2$ registered that $P1$ is the registrant. On

replacing the line, $L2$ sends a writeback request to $L1$. $L1$ replies to this writeback request by sending the data to $L2$ and transitions to $Valid$ state. Then on receiving the writeback from $L1$, $L2$ sends an acknowledgement to $L1$ and in parallel writebacks to memory and waits for an acknowledgement. Meanwhile, let us assume that $L1$ issued a registration request (on receiving a store request) and successfully registers itself with $L2$. At this point, the trace that led to the discovery of the bug consists of yet another $L2$ replacement finally leading to multiple writebacks to memory in flight, which is incorrect because they can be serviced out of order. The real source of this bug is allowing $L1$ registration to proceed while a writeback to memory is pending. This can be easily solved by serializing the requests at $L2$ (i.e., holding the $L1$ registration until the first writeback is completed), which is already present in our high level specification but was missed out in the actual protocol implementation.

Bug 3. The last bug is due to a protocol race where both the L1s and the L2 replace the line. This bug involves both cores and cross phase interactions. At the beginning of the phase, let us assume that $L1_{P1}$ is in $Invalid$ state and $L1_{P2}$ is in $Registered$ state (from the previous phase). $L1_{P2}$ replaces the line and issues a writeback to $L2$. While this writeback is in flight, $L1_{P1}$ successfully registers itself with $L2$ ($L2$ redirects the request to $L1_{P2}$ as it is the current registrant). This is followed by a replacement, thus triggering another writeback to $L2$. $L2$ first receives the writeback from $L1_{P1}$ and responds by sending an acknowledgement and transitioning to $Valid$ state while setting the $dirty$ bit to true. Now, $L2$ also replaces the line transitioning to $Invalid$ state and writebacks to memory. But the writeback from $L1_{P2}$ is still in flight. This writeback now reaches $L2$ while in $Invalid$ state and the current implementation doesn't handle this case and results in an assertion failure. This bug can be simply fixed by adding an extra transition to send an acknowledgement to the requesting $L1$ without triggering any actions at $L2$.

4.4.2 MESI bugs

Bug 1. The first bug is caused due to unhandled protocol race between $L2$ and $L1$ replacements. To begin with, $L1_{P1}$ is in $Exclusive$ state and $L2$ records that $P1$ is the exclusive owner. Then both $L2$ and $L1$ replace the lines simultaneously, triggering invalidation and writeback messages respectively. $L1_{P1}$ on receiving the invalidation message, transitions to $Invalid$ state and sends data to $L2$. On receiving this data, $L2$ completes the rest of the steps for the replacement. In the end, both $L1$ and $L2$ have transitioned to $Invalid$ states, but the initial writeback message from $L1$ is still in the network and this is incorrect. This

bug can be solved by not sending the data when $L1$ receives invalidation message and treat the invalidation message itself as acknowledgement for its earlier writeback message. Also, treat $L1$ writeback message as the data response for invalidation message at $L2$.

Bug 2. The second bug results in a deadlock scenario due to incorrect transitioning by $L2$ on a clean replacement. Let us assume that $L1_{P1}$ is in $Exclusive$ state (with $dirty$ bit set to false) and $L2$ records $P1$ is the exclusive owner. $L1_{P1}$ on replacement writes back to $L2$ and this writeback triggers a transition to $Modified$ state in $L2$ (with $dirty$ bit set to false). On a future $L2$ replacement to this line, this should result in a clean silent replacement as the dirty bit is not set. But the current implementation incorrectly transitions to an intermediate state which waits for an acknowledgement from memory even though this transition doesn't trigger any writeback to memory. Hence, this results in a deadlock situation. The fix is simple and it requires transitioning to $Invalid$ state instead.

Bug 3. The third bug too results in a deadlock situation due to incorrectly handled protocol race between $Exclusive\_unblock$ (response sent to unblock L2 on receiving an exclusive access) and $L1$ writeback. Here we explain the trace that leads to the deadlock situation. In the current implementation, while $L2$ is waiting for an $Exclusive\_unblock$, an incoming $L1$ writeback from the same core is consumed without sending any reply to the requesting $L1$. Surprisingly, this situation is not handled correctly. When the incoming $L1$ writeback reaches $L2$, it checks whether this writeback is coming from the current owner or from a previous owner. It so happens that the owner information is updated on $Exclusive\_unblock$ and as $L1$ writeback and $Exclusive\_unblock$ race, $L2$ incorrectly discards the $L1$ writeback assuming it is coming from a previous owner. This bug can be fixed by holding the $L1$ writeback to be serviced until $Exclusive\_unblock$ is received by $L2$.

Bug 4. The fourth bug is similar to the third but instead its caused because of a protocol race between $Unblock$ (response sent while transitioning to $Shared$ or $Invalid$ states) and $L1$ writeback.

Bug 5. The fifth bug is caused by a race between $L1$ writeback and a write request by some other $L1$. This scenario does not arise in DeNovo as the software guarantees data race freedom. Let us assume that to begin with $L1_{P1}$ is in $Modified$ state, $L1_{P2}$ is in $Invalid$ state and $L2$ records that the cache entry is modified in $L1_{P1}$. Then, $L1_{P1}$ issues a replacement triggering a writeback ($PUTX$) and transitions to an transient state waiting for an acknowledgement to this writeback request. In the meanwhile, $L1_{P2}$ issues a write request triggering $GETX$ to $L2$. $L2$ first receives $GETX$ from $L1_{P2}$. It forwards the request to $L1_{P1}$ and waits

for an acknowledgement from $L1_{P2}$. $L1_{P1}$ on receiving the $GETX$ request, forwards the data to $L1_{P2}$ and transitions to $Invalid$ state. Then, $L1_{P2}$ on receiving the data from $L1_{P1}$ transitions to $Modified$ state and unblocks the directory which in turn records that cache entry is now modified in $L1_{P2}$. But the writeback ($PUTX$) sent by $L1_{P1}$ is still in the network and it can reach the directory at any time as we have an unordered network. In the particular trace which led to the error, $L1_{P1}$ later services a write request invalidating $L1_{P2}$ and the directory is appropriately updated. $L1_{P1}$'s writeback($PUTX$) then reaches the directory which is clearly an error. We solved this problem by not transitioning $L1_{P1}$ to $Invalid$ state on receiving $L1_{P2}$'s $GETX$ request and adding a transition to send a writeback acknowledgement when the requester is not the owner in the directory's record. With this, there is no longer a dangling $PUTX$ in the network and the problem is solved.

Bug 6. The last bug is similar to that of the fifth bug but instead its caused by a race between $L1$ writeback and a read request by some other $L1$.

Comparing states explored: After fixing all the bugs, the model for MESI explores 1,257,500 states in 173 seconds whereas the model for DeNovo explores 85,012 states in 8.66 seconds. Our experience clearly indicates the simplicity and reduced verification overhead for DeNovo compared to MESI.

CHAPTER 5

PERFORMANCE EVALUATION

5.1 Simulation Environment

We use a full-system simulation environment, which consists of the Wisconsin GEMS memory timing simulator [42] driven by the Intel/Virtutech Simics[55] full-system functional simulator. The 5-stage, one-issue, in-order core model provided as part of Simics is used to drive the DeNovo protocol implemented in the memory timing simulator. We also use the Princeton Garnet [7] interconnection network simulator to accurately model network traffic. We chose not to employ a detailed core timing model due to an already excessive simulation time ($>$ 16 hours for some cases). The use of a simple core as opposed to a more aggressive out-of-order superscalar processor model implies that our memory stall times (and improvements thereof) are under-reported as a fraction of total execution time.

Processor Parameters		**Memory Hierarchy Parameters**	
Frequency	2Ghz	L1 (Data cache)	128KB
Number of cores	64	L2 (Shared, 16 banks, NUCA)	32MB
		Memory	256MB, 4 on-chip controllers
		L1 hit latency	1 cycle
		L2 hit latency	29 $\sim$ 61 cycles
		Remote L1 hit latency	35 $\sim$ 83 cycles
		Memory latency	197 $\sim$ 261 cycles

Table 5.1 Parameters of the simulated processor.

Table 5.1 summarizes the key common parameters of our simulated systems, consisting of 64 cores. Each core has a 128KB private L1 Dcache (we do not model an Icache). L2 cache is shared and banked (512KB per core). The latencies in Table 5.1 are chosen to be similar to those of Nehalem [28], and then adjusted to take some properties of the simulated processor (in-order core, two-level cache) into account.

5.2 Simulated Protocols

We compare the following four different systems:

MESI word (Mword): MESI protocol with single-word (4 byte) cache lines. (All our benchmarks are data-race-free at the word granularity.)

DeNovo word (Dword): DeNovo protocol with single-word (4 byte) cache lines (Section 3.1).

MESI line (Mline): The MESI protocol with 64 byte cache lines.

DeNovo line (Dline): DeNovo protocol with 64 byte cache lines. As explained in Section 3.2, this is similar to Mline, but retains word level coherence granularity. Also, L2 is inclusive of only the words that are modified in an L1 cache (similar to Dword). We optimistically do not charge any additional cycles for gathering/scattering valid-only packets. We charge network bandwidth for only the valid part of the cache line plus the valid-word bit vector.

5.3 Conveying Regions for Self-invalidation

In a real system, the compiler would convey the region of a data through memory instructions (Chapter 3). We did not have a compiler implementation for this study, and so, created an API, *setRegion(beginAddr, size, R)*, to manually instrument the program to convey this information. At every allocation of a heap object, we make this call into the simulator to indicate the region for the allocated addresses. This call triggers a hap in the simulator, which updates a table in the simulator which maintains a mapping from the virtual address, *[beginAddr, beginAddr+size]* to the region number, *R*. At every load or store, the simulator indexes into the table through the address to find the region number for that address (which is then stored with the data in the L1 cache). This implementation basically emulates the behavior of memory instruction itself conveying the region number to the hardware.

At the end of a parallel phase, we insert *flush(R)* calls to self-invalidate regions. self-invalidate regions. This call invalidates all the data in the cache associated with the region *R* that is not *touched* or *registered*. For the applications studied in this thesis (see below), the total number of regions ranged from 2 to about 20. These could be coalesced by the compiler, but we did not explore that here.

5.4 Workloads

We use four benchmarks for evaluation. FFT and LU are from the SPLASH-2 benchmark suite [56]. kdTree [21] is a program for construction of k-D trees which are well studied acceleration data structures for ray tracing in the increasingly important area of graphics and visualization. We use two versions of kdTree: one with false-sharing and another without. The following paragraphs provide a brief description of each workload as used in this paper.

Each application is composed of a set of phases and the computation in the parallel phases is divided among the processors using fork/join parallelism. For each application, the parallel phases are data-race-free. Memory system timing simulation was performed only during the task computations, and not during synchronization (although synchronization time was measured assuming a CPI of one).

FFT. FFT is a complex 1-D Fast Fourier Transform kernel. The data set consists of the complex data points to be transformed, and another set of complex data points referred to as the *roots of unity*. With input size *n* and *p* processors, both sets of data are organized as $\sqrt{n} \times \sqrt{n}$ matrices partitioned so that every processor is assigned a contiguous set of rows which are allocated in its local memory. There is one more matrix of the same size as the ones above that acts as the scratch memory. Communication occurs in three matrix transpose steps, which require all-to-all interprocessor communication. Every processor transposes a contiguous $\sqrt{n}/p \times \sqrt{n}/p$ submatrix from every other processor, and transposes one submatrix locally. In the application code, each matrix is put in a separate regions. For this workload, we use $n = 2^{16}$ and $p = 16$. The program has one warmup phase in which each processor touches the rows assigned to it, followed by six more measured phases that actually do the transformation.

LU. The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The dense $n \times n$ matrix A is divided into an $N \times N$ array of $B \times B$ blocks ($n = NB$) to exploit temporal locality on submatrix elements. To reduce communication, block ownership is assigned using a 2-D scatter decomposition, with blocks being updated by the processors that own them. Elements within a block are allocated contiguously to improve spatial locality benefits, and blocks are allocated locally to the processors that own them. In the application code, the whole matrix A is put in one region. For this workload, we use $n = 512$ and $B = 16$. The program has one warmup phase in which each processor touches its local data. This phase is followed by the measured phases doing the actual iterative LU factorization.

kdTree. This benchmark implements the k-D tree construction algorithm described in [21].The input

to the algorithm is a mesh of triangles. The output of the algorithm is a tree of nodes, in which, each node represents a subspace that contains all the triangles that intersect with that subspace. Each parent node in the tree has exactly two child nodes which are obtained by dividing the subspace of the parent along one of the X, Y or Z dimensions. In the initialization phase of the algorithm, two auxiliary arrays of structs (AoS) are constructed. The rest of the algorithm is a loop, in which, each iteration creates a new level in the tree. In each itertion of this loop, these two data structures are divided between processors, and each processor scans its portions of the arrays in a streaming fashion to find the best division planes for nodes in the last level of the tree. After that, each node is broken into two new nodes, creating a new level in the tree. In our implementation each field of each of the auxiliary structs is put in a different region. For this workload, we use the well known bunny input. The first iteration of the main loop is used for warmup and is followed by one measured iteration. We use two versions of kdTree : *kdTree-false* which consisted of false sharing in an auxiliary data structure and *kdTree-padded* which included padding to eliminate this false sharing. We use these two versions to analyze the effect of application-level false sharing on the DeNovo protocols.

5.5 Results

Figure 5.1 shows four charts for each application. The first chart for each application shows the execution times for MESI word (MW), DeNovo word (DW), MESI line (ML), and DeNovo line (DL) as described in Section 5.2. The bars are normalized to ML (the state-of-the-art). The execution time is divided into compute time, memory stall time, and synchronization stall time. The second and third charts show the detailed memory stall time breakdown and read miss count breakdown respectively, which are further divided into L1 misses that hit in L2, a remote L1, or main memory. And finally, the fourth chart for each application shows the number of flits injected into the on-chip network, normalized to that of ML.

All of the applications were run with 64 cores and considerably large synchronization times were measured in all the simulation runs. This should be attributed to the load imbalance inherently found in those applications. Using larger input sizes would alleviate this problem but prohibitively long simulation times made that impractical for this paper. Also, for kdTree applications, we observed that synchronization time varies significantly across different protocols. We believe the reason is that these applications are parallelized using the Intel Threading Building Blocks (TBB) library. The work-stealing algorithm used in TBB causes the number of under-utilized and over-utilized processors to vary across protocols due to different

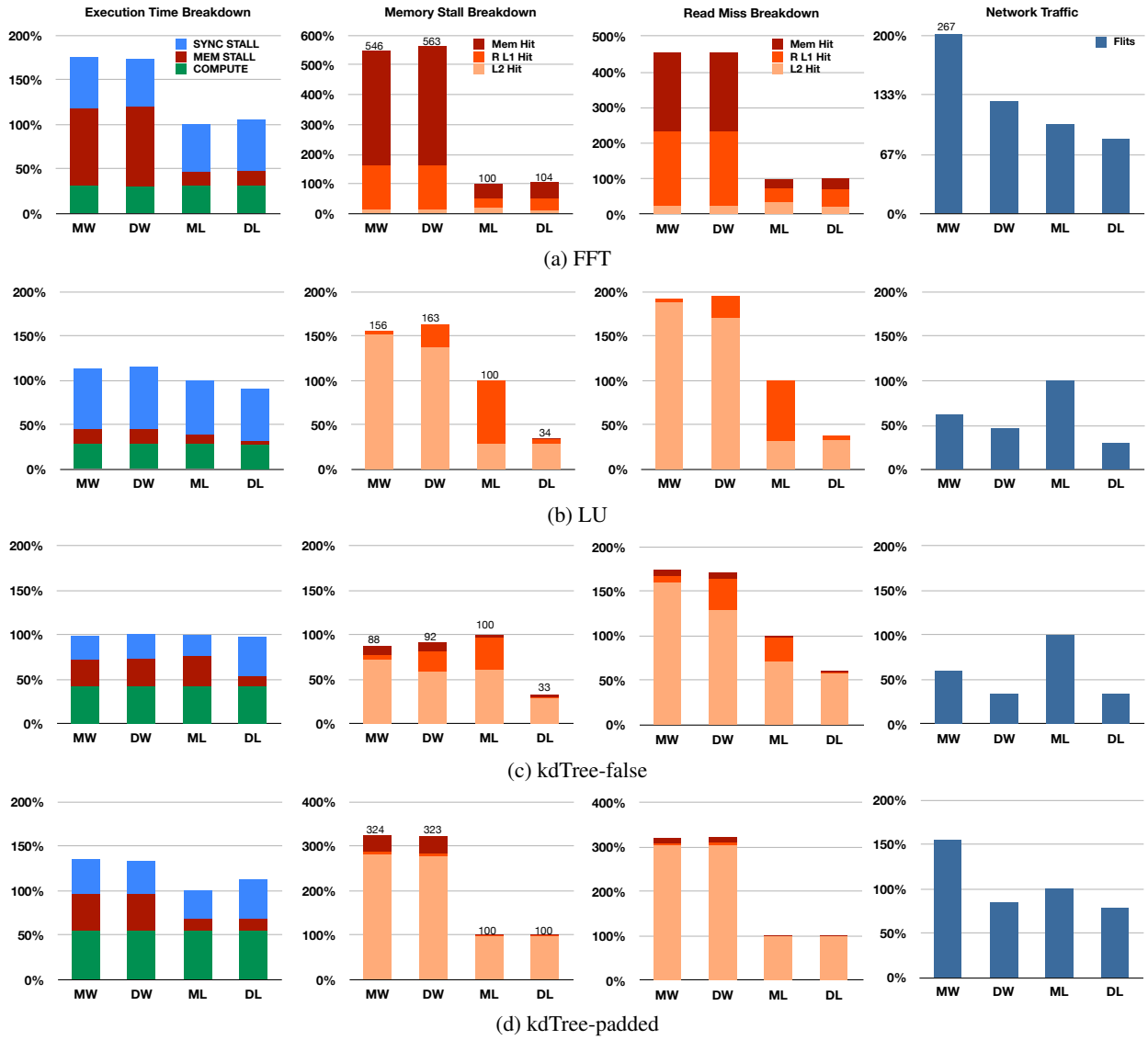

Figure 5.1 Detailed Simulation Results (All normalized to ML)

timing conditions. Again, using a larger input with enough work for all the processors could probably alleviate this problem.

MESI vs. DeNovo word protocols: In all cases, DW performs almost the same or just slightly worse than than MW. The difference mostly comes from the remote L1 hits being more in DW than MW. This is because in MW, the first reader forces the last writer to writeback to L2. Then, subsequent readers go to L2. For DW, on the other hand, all later readers go to the last writer's L1 via L2. This slightly increases the total memory stall time of DW. However, in terms of network traffic, DW always outperforms MW.

MESI vs. DeNovo line protocols: Interestingly, in LU and kdTree-false, DL outperforms ML by around

67% reduction in terms of memory stall time. Here, DL enjoys one major advantage over ML: DL incurs no false sharing due to its per-word coherence state. Both LU and kdTree-false contain some false sharing, as indicated by the significantly higher remote L1 hit component in the miss rate and stall time graphs for ML. In terms of network traffic, DL always outperforms ML with up to 70% reduction in total network traffic which completely translates to power savings.

Effectiveness of cache lines for MESI: Comparing MW with ML, we see that the amount of memory stall time reduction resulting from transferring a contiguous cache line instead of just a word is highly application dependent. Most interestingly, for kdTree (object-oriented AoS style and with false sharing), the word based MESI does better than the line based MESI by 12%. This is due to the combination of false sharing and less than perfect spatial locality. This is the motivation for an ongoing work on one of the optimizations of the DeNovo protocol which brings in only useful valid data into the cache with the help of user provided annotations.

Effectiveness of cache lines for DeNovo: Comparing DW with DL, we see again the strong application dependence of the effectiveness of cache lines. However, because false sharing is not an issue with DeNovo, both LU and kdTree-false enjoy larger benefits from cache lines than in the case of MESI (128% and 59% reduction in memory stalls). Also, DL almost always incurs considerably less network traffic (except in case of kdTree-false).

CHAPTER 6

CONCLUSION

Disciplined programming models will be essential for software programmability and clearly specifiable hardware/software semantics. DeNovo is a hardware architecture designed from the ground up to exploit the opportunities exposed by these disciplined programming models. In this thesis, we formally verified the DeNovo word-based cache coherence protocol using a model checking tool called Murphi and found three bugs. All of these bugs were simple to fix and were mistakes in translating the high level description into the implementation. We also verified a state-of-the art implementation of the MESI protocol and surprisingly found six bugs. Most of these bugs were difficult to analyze and took several days to fix. To evaluate the simplicity of the DeNovo protocol, we compared the total number of reachable states and showed that DeNovo has 15X fewer reachable states when compared to MESI. This translates to 20X difference in the runtime of the model checking tool.

Moreover, for the applications we evaluated, we showed that this simplicity of the DeNovo protocol does not compromise performance. For some of the applications, the DeNovo protocol achieves a reduction in memory stall time up to 67% and network traffic up to 70% over the MESI protocol.

In the future, we would like to extend the verification of the word-based DeNovo protocol to the line-based protocol. We also would like to compare the number of reachable states for both DeNovo and MESI for various abstract models by varying system parameters like number of processors, addresses, regions, etc. This enables us to study how the complexity grows when moved to larger systems. We are currently working on adding some of the optimizations like direct cache-to-cache transfer and flexible communication granularity to DeNovo and we would like to formally verify the correctness of these optimizations. Finally, we would like to have more extensive evaluations of the DeNovo protocol.

REFERENCES

[1] OpenSPARC^{TM} T2 system-on-chip (soc) microarchitecture specification, May 2008.

[2] Dennis Abts et al. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *ISPD*, 2003.

[3] Dennis Abts, David J. Lilja, and Steve Scott. Toward complexityeffective verification: A case study of the cray sv2 cache coherence protocol. In *In Proceedings of the Workshop on Complexity-Effective Design held in conjunction with the 27th annual Intl Symposium on Computer Architecture (ISCA2000)*, 2000.

[4] Sarita V. Adve and Hans-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, August 2010.

[5] Sarita V. Adve et al. A Comparison of Entry Consistency and Lazy Release Consistency. In *HPCA*, pages 26–37, February 1996.

[6] Vikram S. Adve and Luis Ceze. *Workshop on Deterministic Multiprocessing and Parallel Programming, U-Washington*, 2009.

[7] Niket Agarwal et al. Garnet: A detailed interconnection network model inside a full-system simulation framework. Technical Report CE-P08-001, Princeton University, 2008.

[8] Matthew D. Allen, Srinath Sridharan, and Gurindar S. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *PPoPP*, pages 85–96, 2009.

[9] Zachary Anderson et al. SharC: Checking Data Sharing Strategies for Multithreaded C. In *PLDI*, pages 149–158, 2008.

[10] Emery D. Berger et al. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, pages 81–96, 2009.

[11] Brian N. Bershad and Matthew J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical Report TR CMU-CS-91-170, CMU, 1991.

[12] Robert D. Blumofe et al. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, pages 207–216, 1995.

[13] Matthias A. Blumrich et al. Virtual memory mapped network interface for the shrimp multicomputer. In *ISCA*, pages 142–153, 1994.

[14] Robert Bocchino et al. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *POPL*, 2011. To appear.

[15] Robert L. Bocchino, Jr. et al. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, pages 97–116, 2009.

[16] Shekhar Borkar. Major Challenges to Achieve Exascale Performance. *Salishan Conference on High-Speed Computing*, 2009.

[17] Z. Budimlic et al. Multi-core Implementations of the Concurrent Collections Programming Model. In *IWCPC*, 2009.

[18] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Verifying safety of a token coherence implementation by parametric compositional refinement. In *In Proceedings of VMCAI*, 2005.

[19] M. Castro et al. Efficient and flexible object sharing. Technical report, IST - INESC, Portugal, July 1995.

[20] Byn Choi et al. Denovo: Rethinking hardware for disciplined parallelism. In *Proceedings of the 2nd Workshop on Hot Topics in Parallelism*, 2010.

[21] Byn Choi et al. Parallel SAH k-D Tree Construction. In *High Performance Graphics (HPG)*, 2010.

[22] Byn Choi, Nima Honarmand, Rakesh Komuravelli, Robert Smolinski, Hyojin Sung, Sarita V. Adve, Vikram S. Adve, Nicholas P. Carter, and Ching-Tsun Chou. Rethinking the multicore memory hierarchy for disciplined parallelism. *Submitted for publication*.

[23] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK, 1982. Springer-Verlag.

[24] David L. Dill et al. Protocol verification as a hardware design aid. In *ICCD '92*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[25] Anwar Ghuloum et al. Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures. Intel White Paper, 2007.

[26] Stein Gjessing et al. Formal specification and verification of sci cache coherence: The top layers. October 1989.

[27] Niklas Gustafsson. Axum: Language Overview. Microsoft Language Specification, 2009.

[28] D. Hackenberg et al. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO*, pages 413–422. IEEE, 2009.

[29] Nikos Hardavellas et al. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *ISCA*, pages 184–195, 2009.

[30] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

[31] J. Howard et al. A 48-core IA-32 Message-Passing Processor with DVFS in 45nm CMOS. In *ISSCC*, pages 108–109, 2010.

[32] Intel. The SCC Platform Overview. http://techresearch.intel.com/spaw2/uploads/files/SCC_Platform_Overview.pdf.

[33] C.N. Ip and D.L. Dill. Efficient verification of symmetric concurrent systems. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD '93. Proceedings., 1993 IEEE International Conference on*, pages 230 –234, October 1993.

[34] Pete Keleher, Alan L. Cox, and Willy Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, pages 13–21, 1992.

[35] Pete Keleher, Sandhya Dwarkadas, Alan L. Cox, and Willy Zwaenepoel. TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems. In *Proc. Winter '94 USENIX Conf.*, pages 115–131, January 1994.

[36] John H. Kelm et al. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *ISCA*, 2009.

[37] M. Kulkarni et al. Optimistic Parallelism Requires Abstractions. In *PLDI*, pages 211–222, 2007.

[38] Akhilesh Kumar et al. Efficient and scalable cache coherence schemes for shared memory hypercube multiprocessors. In *SC*, New York, NY, USA, 1994. ACM.

[39] Edward A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, May 2006.

[40] Daniel Lenoski et al. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *ISCA*, 1990.

[41] Brandon Lucia et al. Conflict exceptions: simplifying concurrent language semantics with precise hardware exceptions for data-races. In *ISCA*, 2010.

[42] Milo M. K. Martin et al. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 33(4):92–99, 2005.

[43] Milo M.K. Martin et al. Token coherence: Decoupling performance and correctness. In *ISCA*, 2003.

[44] Michael R. Marty et al. Improving multiple-cmp systems using token coherence. In *HPCA*, pages 328–339, 2005.

[45] K. L. McMillan and Schwalbe J. Formal verification of the gigamax cache consistency protocol. In *Proceedings of the International Conference on Parallel and Distributed Computing*, pages 242–251, Tokyo, Japan, 1991. Information Processing Society.

[46] A.K. Nanda and L.N. Bhuyan. A formal specification and verification technique for cache coherence protocols. In *ICPP*, pages I22–I26, 1992.

[47] C. Norris IP and David L. Dill. Better verification through symmetry. volume 9, pages 41–75. Springer Netherlands, 1996. 10.1007/BF00625968.

[48] Marek Olszewski et al. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, pages 97–108, 2009.

[49] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for rmo (relaxed memory order). In *Proceedings of the seventh annual ACM symposium on Parallel algorithms and architectures*, SPAA '95, pages 34–41, New York, NY, USA, 1995. ACM.

[50] Fong Pong, Michael Browne, Andreas Nowatzyk, Michel Dubois, and Günes Aybay. Design verification of the s3.mp cache-coherent shared-memory system. *IEEE Trans. Comput.*, 47:135–140, January 1998.

[51] Fong Pong and Michel Dubois. Verification techniques for cache coherence protocols. *ACM Comput. Surv.*, 29:82–126, March 1997.

[52] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[53] Arun Raghavan et al. Token Tenure: PATCHing Token Counting using Directory-Based Cache Coherence. In *MICRO*, 2008.

[54] Daniel J. Sorin et al. Specifying and verifying a broadcast and a multicast snooping cache coherence protocol. *IEEE Trans. Parallel Distrib. Syst.*, 13(6):556–578, 2002.

[55] Virtutech. Simics Full System Simulator. Website, 2006. `http://www.simics.net`.

[56] Steven Cameron Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[57] David A. Wood et al. Verifying a multiprocessor cache controller using random case generation. *IEEE DToC*, 7(4), 1990.

[58] Jason Zebchuk et al. A Tagless Coherence Directory. In *MICRO*, 2009.