DeNovoND: Efficient Hardware Support for Disciplined Non-Determinism

Hyojin Sung, Rakesh Komuravelli, and Sarita V. Adve

Department of Computer Science
University of Illinois at Urbana-Champaign
{sung12, komurav1, sadve}@illinois.edu

Abstract

Recent work has shown that disciplined shared-memory programming models that provide deterministic-by-default semantics can simplify both parallel software and hardware. Specifically, the DeNovo hardware system has shown that the software guarantees of such models (e.g., data-race-freedom and explicit side-effects) can enable simpler, higher performance, and more energy-efficient hardware than the current state-of-the-art *for deterministic programs.* Many applications, however, contain non-deterministic parts; e.g., using lock synchronization. For commercial hardware to exploit the benefits of DeNovo, it is therefore necessary to extend DeNovo to support non-deterministic applications.

This paper proposes DeNovoND, a system that supports lock-based, disciplined non-determinism, with the simplicity, performance, and energy benefits of DeNovo. We use a combination of distributed queue-based locks and access signatures to implement simple memory consistency semantics for safe non-determinism, with a coherence protocol that does not require transient states, invalidation traffic, or directories, and does not incur false sharing. The resulting system is simpler, shows comparable or better execution time, and has 33% less network traffic on average (translating directly into energy savings) relative to a state-of-the-art invalidation-based protocol for 8 applications designed for lock synchronization.

Categories and Subject Descriptors B.3.2 [*Hardware*]: Memory Structures – Cache memories; Shared memory; C.1.2 [*Processor Architectures*]: Multiple Data Stream Architectures (Multiprocessors) – Parallel processors

Keywords shared memory, cache coherence, disciplined parallelism, memory consistency, non-determinism

1. Introduction

Shared-memory remains a popular programming model among multicore programmers and is the de facto model provided by multicore hardware. It is, however, increasingly evident that unbridled "wild" shared-memory programming environments that allow data races, ubiquitous non-determinism, unstructured parallelism, and complex memory consistency models make programming, debugging, testing, and maintaining software difficult [1, 32]. Recent software research has therefore proposed more *disciplined* shared-memory programming models that retain the advantage of a global address space, but make it easier to write safe parallel programs that are easier to debug, test, and maintain [4–6, 10–12, 14, 18, 19, 30, 39].

At the same time, providing hardware cache coherence and consistency that can scale in a power-efficient manner to hundreds of cores is also a significant challenge. There has recently been a surge in research by academics (see Section 7) and hardware companies [23, 26] to address this challenge in unconventional ways. In particular, the DeNovo hardware project observes that disciplined shared-memory programming models such as mentioned above can drive a holistic rethinking of the multicore memory hierarchy, providing more complexity-, performance-, and power-efficient hardware than the state-of-the-art *for deterministic programs* [17]. This paper shows that the benefits of disciplined programming and DeNovo can be extended to non-deterministic programs as well.

DeNovo has used Deterministic Parallel Java (DPJ) as an example disciplined programming model [11] to drive its design. DPJ provides the programmer with a novel region-based type and effects system to convey the read and write side-effects on shared-memory for every method. A type-checked DPJ program is guaranteed deterministic-by-default semantics. That is, unless non-determinism is explicitly requested, DPJ programs appear deterministic and with sequential semantics (the programmer can debug and test such a program as if it were sequential). Even when non-determinism is explicitly requested, DPJ provides strong safety guarantees; e.g., data-race-freedom, strong isolation, and sequential composition of deterministic code sections [12]. The DPJ compiler enforces these guarantees by checking that conflicting accesses from two concurrent tasks – the root cause of non-determinism – are always identified (as atomic) and always occur within explicitly marked atomic sections.

DeNovo has so far focused on deterministic programs, and shown that DPJ's information and guarantees can be exploited to provide a simpler and more efficient cache coherence protocol than the state-of-the-art MESI for such programs [17]. Specifically, DeNovo's protocol has the following advantages: (1) The implementation has no transient states and so is much easier to verify (verification is an order of magnitude faster) and much easier to extend (incorporating optimizations did not introduce any protocol state changes). (2) DeNovo does not rely on writer-induced invalidations; it therefore eliminates invalidation message traffic and does not require storage overhead for sharer lists in directories, removing a key source of unscalability. (3) DeNovo keeps coherence state at the granularity at which data is shared and so does not suffer from false sharing (the added state overhead is much less than the reduced directory state). Overall, compared to MESI, DeNovo is much simpler,

ASPLOS '13, March 16–20, 2013, Houston, Texas, USA.
Copyright © 2013 ACM 978-1-4503-1870-9/13/03...$15.00

performs comparably or better than MESI, and is more energy-efficient (since it reduces cache misses and network traffic) for a range of deterministic codes.

Although determinism is considered desirable for many application classes, there are many common codes that are non-deterministic or contain parts that are non-deterministic, most commonly through lock synchronization. For example, 21 out of 25 of the PARSEC and SPLASH-2 benchmarks contain locks in some parts. DeNovo currently cannot run such codes.[1] For commercial hardware to be able to exploit the benefits of DeNovo, it is imperative that we develop techniques to support non-deterministic codes with at least as much performance as conventional systems, without losing the benefits of DeNovo.

This paper explores exploiting disciplined programming models to develop simpler and more efficient hardware even for programs that contain non-determinism. We use DPJ's safe non-determinism model (with atomic sections replaced with locks), and show that simple additions to the DeNovo coherence protocol can support such non-determinism without giving up on DeNovo's previous advantages. We call the resulting system DeNovoND.

For deterministic programs, DeNovo achieves its benefits primarily by recognizing that DPJ explicitly provides the regions that could be potentially written in a parallel phase (e.g., DPJ's foreach or cobegin constructs) through its explicit effects. At the start of a new phase, DeNovo's cores execute compiler-inserted self-invalidations to all regions that could have write effects in the previous phase. Their caches therefore now have only valid data. If any of this data is updated in the next phase, DPJ's data-race-freedom guarantee ensures that only the writing core will read that data, ensuring up-to-date values for all reads. These observations eliminate the need for writer-induced invalidations, directories, and false sharing due to cache line driven protocols.

Unlike DeNovo, DeNovoND cannot assume that a parallel phase will have no conflicting accesses among concurrent tasks any more, but it knows that such accesses will be protected by the same lock (this lock may change in a different parallel phase). Further, such accesses are explicitly identified as atomic accesses in DPJ programs. Within a critical section, DeNovoND therefore tracks atomic writes through a signature which is conveyed to the next acquirer of the lock. The acquirer uses the signature to determine which data to invalidate in its cache. The strong guarantees given by DPJ enable an efficient implementation, while still providing freedom to express a variety of non-deterministic algorithms. Underlying the above is an implementation for a lock that does not require directories and a full-fledged MESI protocol – we use a distributed queue based implementation modeled after the Queue-on-Sync-Bit (QOSB) lock [20].

Overall, our system retains the advantages of DeNovo while significantly expanding the class of programs it supports without compromising performance. Specifically, for lock accesses, although DeNovo's coherence protocol state machine is extended to handle the distributed queue, it reuses the state bits from DeNovo's data accesses. For data accesses, again, no new externally visible states are added; the only support needed is a signature per core, the ability to transfer it to the next acquirer, and to use it for self-invalidation at subsequent reads. A bit per word at the L1 cache is used as an optimization. We continue to not have any directories, not have invalidations, and not incur false sharing.

We compared DeNovoND with a state-of-the-art MESI protocol for 11 benchmarks with lock synchronization. 3 of these spent more than 70% of their time in lock acquires, clearly requiring alternate synchronization techniques for reasonable parallel efficiency that are out of the scope of this work. We therefore focus on the remaining 8 benchmarks here, although we report results for the above 3 as well for completeness. We found that DeNovoND performs comparably or slightly better than MESI in terms of execution time. DeNovoND also shows 33% lower network traffic than MESI on average, which directly translates into energy savings. Performance optimizations previously proposed for DeNovo (for cache to cache and flexible granularity data transfer) [17] are applicable to DeNovoND as well without any additional changes, but are orthogonal to this work and not reported here. Thus, DeNovoND allows us to extend the benefits of DeNovo to include lock-based (safe) non-deterministic applications.

Our system shares commonalities with previous software distributed shared memory consistency models such as lazy release consistency [27], entry consistency [7], and scope consistency [22] as well as recent hardware shared-memory work that exploits data-race-freedom such as SARC [25]. However, none of those systems distinguish between deterministic and non-deterministic accesses in a way that is possible with our hardware/software co-designed approach, and so those systems cannot exploit the corresponding optimizations. Section 7 discusses the relationship of our work to prior work in more detail.

While DeNovoND takes a major step in exploiting software discipline in hardware for a larger class of programs, there is still much left to future work and outside the scope of one paper. Section 8 discusses future work to explore how to incorporate other key constructs (e.g., pipelined parallelism), and support more complex codes such as legacy codes and operating systems within this vision.

2. Background

2.1 Deterministic Parallel Java (DPJ)

DPJ is an extension to Java that enforces deterministic-by-default semantics via compile-time type checking [11, 12]. We first discuss DPJ without non-deterministic constructs [11]. DPJ provides parallel constructs of foreach and cobegin to express parallelism in a structured way as in many current languages (we refer to an iteration of a foreach loop or a parallel statement of a cobegin as a task). DPJ provides a new type and effect system for expressing common patterns of imperative, object-oriented programs. The DPJ programmer assigns every object field or array element to a named "region" and annotates every method with read and write "effects" summarizing the regions read and written by that method (a region can be non-contiguous in memory). The compiler uses this information to (i) type-check program operations in the region type system and (ii) ensure that no two parallel tasks interfere (conflict).

DPJ also provides parallel constructs that are potentially non-deterministic; i.e., foreach_nd and cobegin_nd [12]. These constructs allow conflicting accesses between their tasks, but require that such accesses be enclosed within atomic sections, their read and write effect declarations also include the atomic keyword, and their region types be declared as atomic. Note that there continue to be no conflicts allowed between a task from a deterministic parallel construct and any other concurrent (non-deterministic or deterministic) task. The compiler checks that all of the above constraints are satisfied by any type-checked program, again using a simple, modular type checking algorithm.

With the above constraints, DPJ is able to provide the following guarantees: (1) Data-race freedom. (2) Strong isolation of accesses in atomic section constructs and all deterministic parallel constructs; i.e., these constructs appear to execute atomically. (3) Sequential composition for deterministic constructs; i.e., tasks of a deterministic construct appear to occur in the sequential order

[1] The DeNovo work reports results for some of these benchmarks, but the parts with locks were either run sequentially or rewritten or not simulated [17].

implied by the program (even if they contain or are contained within non-deterministic constructs). (4) Determinism-by-default; i.e., any parallel construct that does not contain an explicit non-deterministic construct provides deterministic heap output for a given heap input. The above guarantees are strong – they not only ensure sequential consistency but also allow programmers to reason with very high-level strongly isolated and composable components such as complete foreach constructs and all atomic sections.

Although DPJ supports atomic sections, this paper assumes we can convert them to locks. This is possible because by default we can associate each atomic region with its own lock. For each atomic section, we can acquire locks for each atomic region that it accesses in a predefined order. This can be optimized in several ways; e.g., by coarsening the locks. An implementation of this algorithm is outside the scope of this paper. We therefore use hand inserted locks – for the applications we used, these locks were as provided in the original application.

2.2 DeNovo for Deterministic Codes

DeNovo divides the coherence problem into two parts:

(1) *No stale data:* A read should never see stale data in its private cache(s).

(2) *Locatable up-to-date data:* When a read misses in its private cache(s), it should know where to get an up-to-date copy of the data.

Above, *stale* and *up-to-date* are defined by the memory consistency model (sequential semantics, in our case). For (1), DeNovo recognizes that DPJ explicitly provides the regions that could be potentially written in a parallel phase (each DPJ parallel construct such as cobegin and foreach forms a phase, with an implicit barrier at the join). Before starting a new phase, a core issues compiler-inserted self-invalidations for all regions that could have write effects in the previous phase, eliminating all stale data from its private cache(s).[2] For data updated in the current phase, DPJ's data-race-freedom guarantee ensures that only the writing core will read that data, ensuring up-to-date values for all (private) cache hits. For (2), DeNovo uses a structure called the *registry* to keep track of one up-to-date copy of each cache line. This is analogous to a conventional directory, but unlike the latter, it does not track all sharers of a cache line (eliminating a source of unscalability). With systems with a shared last level cache, the data bank of the cache doubles as the registry storing the data or a pointer to it.

The DeNovo protocol has three states, *Registered*, *Valid*, and *Invalid*. These states are analogous to those in a conventional MSI directory protocol; *Registered* is similar to *M* with the line modified in a private cache and *Valid* is similar to *S*. The DeNovo protocol state transition diagram also resembles typical textbook pictures for MSI. A key difference, however, is that real implementations of MSI have tens of transient states to handle protocol races, introducing significant complexity and making verification difficult. In contrast, DeNovo has no transient states since it assumes race-free software, which eliminates virtually all races from the protocol hardware.

Next we describe the key aspects of the protocol's operation and refer to [17] for more details. For easier exposition, we assume a two level cache hierarchy with a shared L2 without loss of generality, and a line size of one word (this is relaxed below). A read hits in the L1 if the line is *Valid* or *Registered*. A read miss request goes to the registry (the shared L2) and either finds the data there or a pointer to the L1 that contains the data in *Registered* state. In the latter case, the request is routed to the registered data for service.

[2] This requires the cache to store region information as described in [17].

A write to data in *Registered* state at the L1 updates the data. A write to data in *Valid* or *Invalid* state at the L1 immediately transitions the data to *Registered* and updates it (no transients) and generates a registration request (and a writeback if needed). If the data is not registered elsewhere, the L2 immediately registers it and sends an acknowledgment. Otherwise, the L2 records the new registration and forwards the request to the previously registered core to relinquish its registration. Due to the data-race-free guarantee, registration transfer occurs only once in a phase (assuming no task migration, which can also be easily handled [17]), without any danger of protocol races.

Additionally, as an optimization, L1 contains *touched* bits that are set when the corresponding data is read. Due to data-race-freedom, it is guaranteed that no other core will write such data in that phase. Thus, "touched" data is up-to-date and does not need to be invalidated for the next phase. All self-invalidations occur at the end of the phase – regions with write effects in that phase are invalidated unless the data is registered or touched. Touched bits are reset after the invalidation, in preparation for the next phase.

The baseline word-based DeNovo protocol assumes equal address/tag allocation, communication, and coherence granularity, which is the granularity at which data-race-freedom is ensured. This granularity is a word for the applications evaluated. (Details about supporting sub-word (byte) granularity can be found in [17].) DeNovo further observes that any data that is marked *touched* or *Registered* is always up-to-date and can be freely copied from one cache to another without informing anyone (there is no directory tracking sharer lists). Thus, the word-based DeNovo protocol is easily enhanced to operate on larger communication and address/tag allocation granularities, while still maintaining coherence state at the word granularity.

A natural granularity for communication and allocation is a conventional cache line (e.g., 64 bytes), and the corresponding DeNovo protocol is referred to as the line based protocol. Here, a responding cache for a demand request sends a cache line worth of data (potentially with some words marked as invalid) and the valid words in the response message are merged with the local copy of the cache line of the requestor. These words are marked as *Valid*, but not *touched* (the *touched* bit is set when those words are actually read). DeNovoND is designed on top of this line-based protocol.

DeNovo has also explored more flexible communication granularities (more or less than one cache line) and direct L1 to L1 data transfers. These optimizations are simple with DeNovo and do not require any new states, but are difficult to incorporate in conventional protocols because they introduce even more transient states. The same optimizations can be directly applied to DeNovoND as well, again with no new states for DeNovoND. We do not study them here since they are orthogonal to the goal of this paper.

The DeNovo protocol we study additionally implements the optimization of write combining where multiple registration requests to words in a given cache line are combined into one request. This optimization was mentioned, but not implemented, in [17] to reduce write traffic. This optimization is not meaningful for conventional protocols since conventional store requests always operate on a full line while DeNovo registrations are for a word.

3. DeNovoND Design Overview

3.1 Basic Assumptions and Definitions

We assume all synchronization occurs through DPJ's parallel constructs (foreach, cobegin, and their nd versions) and through locks. We assume a barrier at the implicit join associated with the parallel constructs. We say all concurrent tasks of a given parallel construct – loop iterations in a foreach and parallel statements in a cobegin – form a *phase*.

For locks, we assume that an atomic section does not call a parallel construct, as is the case with all our applications. Thus, all operations of an atomic section occur within a single task and are enclosed within a lock acquire and release to the same lock variable (there may be nested locks to different lock variables). We refer to memory operations within such a lock acquire/release pair as occurring in a critical section protected by that lock variable.

For data accesses, we assume the ISA provides a mechanism by which loads and stores can be tagged as accessing atomic regions with atomic effects (e.g., with a bit in the op-code). The DPJ compiler has this information and can generate code with the bit set for such accesses. We refer to such accesses below as *atomic* accesses and to others as *non-atomic* accesses. Note that the former are regular data accesses from atomic sections and are not to be confused with atomic read-modify-writes or the C++ atomic keyword used for synchronization races.

Without loss of generality, we assume a two level cache hierarchy. We also assume a shared L2 cache. DeNovoND can be extended to deeper hierarchies and private last level caches in a straightforward way (similar to DeNovo [17]).

3.2 Memory Consistency Model

For a correct design, we must first understand the constraints imposed by the memory consistency model which specifies what value a read must return.

Informal model: DPJ provides a very strong consistency model. It guarantees sequential consistency and hence a total order over all memory operations (that is consistent with program order). A read must return the value of the last write to its location as defined by this total order. DPJ also enforces additional rules that further constrain this last write for data operations, simplifying reasoning for software and implementation for hardware as follows.

Non-atomic accesses: DPJ ensures that for a non-atomic access, there cannot be a conflicting access by another concurrent task in the same phase. Thus, for a non-atomic read, the last conflicting write is either from its own task or from a task in a previous phase. This is identical to DeNovo and we can use the identical implementation.

Atomic accesses: For atomic accesses as defined above, DPJ allows conflicting accesses among concurrent tasks, but ensures that all such accesses to a given location are in critical sections protected with the same lock. These critical sections must execute atomically, imposing a total order on all conflicting atomic accesses within a phase. A read therefore must return the value from the (unique) last conflicting write from a critical section in the current phase; if such a write does not exist, then the read must return the (unique) last conflicting write from the previous phase.

Formal model: We now state the model more formally. Note that this model is motivated as a specification for hardware and is therefore at a low level, in terms of individual reads and writes. DPJ programmers work at a higher level in terms of composition and serialization of higher level constructs (cobegin, atomic section, etc.) as described in Section 2.1. Our model can be stated in two parts for synchronization and data accesses respectively:

(1) Synchronization accesses are sequentially consistent. This implies a total order between phases and between critical sections to a given lock variable within a phase; this total order is consistent with program order.

(2) For conflicting data accesses, X and Y, we define a *happens-before* relation, denoted $\rightarrow_{hb}$ such that $X \rightarrow_{hb} Y$ iff

Type 1 edge: X's phase precedes Y's phase (by the total order in (1)), or

Type 2 edge: X and Y are in the same task, and X is before Y by program order, or

Type 3 edge: X and Y are atomic accesses in critical sections protected by the same lock variable, and X's critical section precedes Y's critical section (by the total order in (1)).

Then DPJ's guarantees ensure that $\rightarrow_{hb}$ orders all conflicting accesses, and hardware should ensure that a data read returns the value of the last conflicting write in $\rightarrow_{hb}$ order. For a non-atomic read, the last write is always ordered before it by a *type 1* or *type 2* $\rightarrow_{hb}$ edge. For an atomic read, the last write may be ordered before it by a *type 2* or *type 3* edge if such a write exists; otherwise, it is ordered by a *type 1* edge.

3.3 Data Coherence Mechanism

The coherence mechanism must simply ensure that a read returns the value from the write as defined by the consistency model. As with DeNovo, we divide the coherence mechanism into two components:

(1) *No stale data:* A read should never see *non-last* (stale) data in its L1 cache(s).

(2) *Locatable up-to-date data:* When a read misses in its L1 cache(s), it should know where to get the *last* (up-to-date) copy of the data.

Above, *last* is precisely defined by the happens-before order. For non-atomic accesses, both components above remain identical to DeNovo since the consistency model requirements are identical. For atomic accesses, the requirements are met as follows.

No stale data: For the first requirement of no stale data, we use self-invalidations as with DeNovo, thereby precluding the need for adding invalidation messages and directories with sharer lists. Additional self-invalidations are needed with DeNovoND only if there are conflicting atomic accesses among concurrent tasks in a phase (otherwise, DeNovo's self-invalidations at the start of a phase suffice). In the case of conflicting atomic accesses among concurrent tasks, we use the happens-before relation to determine *when* and *what* to self-invalidate as follows.

To determine when to self-invalidate, we note that a concurrent conflicting read must be in a critical section itself and must return the value of the last write also in a critical section protected by the same lock in the same phase (type 2 or 3 edge). Thus, it is sufficient to self-invalidate any time between the start of a critical section and an atomic read in that section.

To determine what to self-invalidate, we have several choices. We could invalidate the entire cache (which seems excessive) or only the atomic regions (for which we would need to keep extra state to identify in the cache). An alternative is for each core to update a signature that records all writes to atomic regions, and then to transfer this signature when the lock is acquired by another core. On a first atomic read to a location, the acquiring core needs to check the signature and self-invalidate the location if it is present in the signature. The acquiring core must forward the union of its signature and the signatures it has received to the next acquirer.

Locatable up-to-date data: For the second requirement of finding the value of the last write on a miss, we use ideas similar to DeNovo. On a write to valid or invalid data, the L1 cache sends a registration request to the L2. The registrations are required to complete before the lock release so that conflicting writes from critical sections are serialized in the right order (it is possible to postpone the registration completion until the next lock acquire). A read that misses in the cache simply goes to the registry (L2) to find the up-to-date value.

Thus we continue with only three states in the protocol as before: *Valid*, *Invalid*, and *Registered*. The extra work over DeNovo is to update the signature on atomic writes, send the signature on a lock transfer, and invalidate appropriately on atomic reads. Section 4.1 discusses each of these steps in more detail.

3.4 Distributed Queue-based Locks

Our distributed queue-based lock design is modeled after QOSB [20, 24], where the identities of the cores waiting for a lock are maintained in a queue of pointers distributed across the waiting cores' L1 caches and the L2 cache. All requests to a given lock are serialized at the corresponding shared L2 cache bank. The data portion of the L2 cache entry for a contended lock tracks the last requestor (i.e., the tail of the queue of waiters), referred to as *tailPtr*. When the L2 receives the next request for the lock, it forwards it to the current tail's L1. On receiving such a forwarded request, the L1 checks a bit in its copy of the lock word, called the *Locked* bit, to determine if the lock is still held or was unlocked. In the former case, the L1 stores the requestor's ID in another field of the lock word, referred to as *nextPtr*. In the latter case, the L1 responds to the requestor with its signature and transfers the lock, marking its own lock word *Invalid*. When a core releases a lock, its L1 checks its *nextPtr* – if not null, it transfers the lock (with the signature) to the *nextPtr* core; otherwise, it unsets its *Locked* bit. We allow eviction of lock words from the L1 and L2 caches by reusing the data portion of the lock words in the next level of the memory hierarchy to store lock queue information. This approach relies on using L2 data banks to store (non-data) metadata, which is similar to DeNovo's tracking of registration information for the *Registered* state. Section 4.2 discusses our implementation in more detail.

4. Implementation

This section discusses in detail how DeNovoND implements the memory consistency model and the coherence mechanism described in Section 3 using *access signatures* and the distributed queue-based lock mechanism. We also qualitatively discuss the hardware and performance overheads of the implementation.

4.1 Access Signatures for Coherence of Atomic Accesses

DeNovoND's memory consistency model requires that a read return the value of the last write preceding it, as ordered by the three types of *happens-before* edges described in Section 3. DeNovo already guarantees that a write ordered by a type 1 or type 2 edge is seen at a read (the former through self-invalidations at the start of a new phase and the latter through single core semantics). For a non-atomic read, a write is ordered only through the above two edge types; therefore, DeNovo already provides consistency for such reads. For atomic reads where a previous (atomic) write is ordered by a type 3 edge, however, DeNovoND must provide a new mechanism – it needs to track which data in atomic regions has been modified in a critical section in the current phase, as well as a mechanism to efficiently represent and transfer this information on a successful lock acquire.

We use an "access signature" for the purpose of tracking atomic writes. A signature is a compact representation of a set at the expense of precision. Its main functionality includes element insertion, membership query, and flash clear functions. DeNovoND implements the access signature as a small Bloom filter in hardware [9]. Due to its storage efficiency, simplicity, and low access latency, a hardware Bloom filter has been a popular solution for many areas including networking and transactional memory [13, 16].

For our Bloom filters, the keys are addresses accessed (i.e., atomic regions that have atomic effects in this phase), since we are interested only in modifications made to those addresses. The key domain dynamically changes between cores and phases, as a new set of atomic accesses occurs. To keep the false positive rate of Bloom filter reasonably low, the size of each Bloom filter should be determined based on the average size of the key domain. This turns out to be quite small in our case (256 bits suffice) since we only track atomic accesses in a given phase (later sections discuss

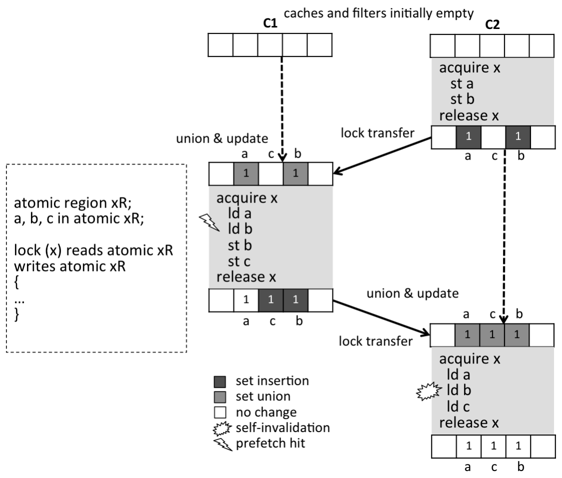


Figure 1: An example of propagating atomic writes using access signatures. Assume *a* and *b* are in the same cache line.

the size in more detail). We conservatively keep one filter per core to track all modifications across different critical sections (with different locks) on the same core. Thus, for a system with *n* cores, we have a total of *n* Bloom filters in the system.

The following uses Figure 1 as a running example to show how DeNovoND uses the Bloom filters. On the left, the figure shows DPJ style code depicting three variables, *a*, *b*, and *c* in atomic region *xR*. It then shows a critical section protected by lock *x* with atomic read and write effects on region *xR*. The right side of the figure shows an execution with two cores, *C1* and *C2*. *C2* acquires the lock for the critical section first, followed by *C1* and then *C2* again. The figure also shows the signatures at each core, assuming a perfect hash function.

On atomic writes: An atomic write (as determined by the op-code of the store instruction as discussed in Section 3) invokes the same cache protocol operations as in DeNovo. That is, if the word is not in *Registered* state at the L1, a registration request is sent to the L2. Additionally, the word is updated right away and any required writeback is sent to the L2 as well.

For DeNovoND, an atomic write additionally inserts the accessed address into its core's Bloom filter. To avoid repeating insertion of the same address to the Bloom filter, we can add an additional bit, called the "dirty bit," to mark a memory location already updated in a given phase. The "dirty bit" is set on the first atomic store request to a word in a phase, and all dirty bits get unset at the end of a phase. If a store finds the dirty bit already set, it means the word is already inserted into the core's Bloom filter and does not need to be inserted again. Since this is purely an optimization, we can piggyback the functionality of a dirty bit on other state bits described below (e.g., the *touched-atomic* bit) – this may result in some extraneous resets, but does not affect correctness and reduces extra state.

Thus, at the end of a critical section, all addresses modified in the section are recorded in the core's filter; i.e., their entries are non-zero. From Figure 1, every store request to *a*, *b*, and *c* in the lightly shaded critical sections updates the Bloom filter on *C1* and *C2*. The second critical section phase on *C2* does not update the Bloom filter since it does not have atomic writes.

On acquire/release: On an acquire, all modifications preceding the release associated with the acquire are made visible to the acquirer by transferring the access signature at the releaser. The releaser compresses and sends the Bloom filter at its core to the acquirer, when transferring the lock. The acquirer, on receiving the Bloom filter, updates its own Bloom filter by making a *union* of its local Bloom filter and the releaser's Bloom filter. Figure 1 shows the resulting Bloom filters at the beginning of each critical section, of which the lightly shaded entries come from the union operation. Note that we only send the signature, not the actual data. On acquire and release points, we also reset the "touched-atomic" and "prefetch" bits (as will be explained in detail below).

On atomic reads: Atomic reads need to conceptually consult the signatures obtained from remote releasers to determine if cached data is valid or stale. If the read is to a word in *Registered* state in the L1, then regardless of the signature state, the word is up-to-date in the cache and the read is a cache hit. If the word is *Invalid* in L1, then a normal read request is sent to L2. If the word is in *Valid* state, then it is also up-to-date if its address does not appear in the access signature. If the word is in *Valid* state and its address hits in the access signature, then it may or may not be up-to-date depending on whether it has been previously read in this critical section.

Specifically, if the word has already been read in this critical section, the previous read brought up-to-date data that is still valid (since no other core can write to the word during the same critical section). We identify this situation by using a *touched-atomic* bit that is set on the first read of the word in a critical section and reset at the release – more precisely, it needs to be reset only when the lock is handed off for another core's acquire (lock hand-off). Thus, a read to a word in *Valid* state with *touched-atomic* bit set is a cache hit.

Another case where a valid word may be up-to-date is when it is obtained as part of a cache line transfer for a demand access to another word in that line. We would like to take advantage of such a prefetch as with conventional cache lines and with DeNovo. If the word comes directly from the L2 or from memory, then it is definitely valid. If it comes from a remote cache, then it is valid if that word was marked as *touched-atomic* or *Registered* in the remote cache. In this case, we can conceptually add another bit called the "prefetch bit" which can be set for prefetched words with the above properties. These bits must be reset on the next lock hand-off or the next acquire, whichever happens first. A read that accesses a valid word with *prefetch* bit set is considered a cache hit. Although the *touched-atomic* and *prefetch* bits are separately motivated, both functions can be achieved by a single bit that we collectively refer to as the *touched-atomic* bit.

In summary, the *touched-atomic* bit of a word is set on the first read of the word in a critical section or for a word prefetched from L2/memory or from a remote L1 in *touched-atomic* or *Registered* state. The bit is reset on an acquire or a lock hand-off, including the end of the phase. A read to *Valid* data with *touched-atomic* bit set or with an address that misses in the access signature is considered a hit. Otherwise, the *Valid* data is no longer up-to-date and must be marked invalid and a read miss request is issued.

In Figure 1, assume that variables *a* and *b* are in the same cache line. Then *C1*'s `load b` will be a hit since *C1*'s `load a` will bring in *b* as well and set its *touched-atomic* bit. On the other hand, `load b` in *C2*'s second critical section is a miss. This is because the preceding `load a` will read *a* in its own cache in *Registered* state and so will not prefetch *b* which is registered at *C1*.

Finally, we note that using a single, plain Bloom filter at each core to determine what to invalidate is inherently conservative. For example, it is possible that an address may have been updated before it had been last seen by a core but not updated again since then; our system will still invalidate the address on a read (in the same phase) from that core. In addition, false positives in a finite Bloom filter cause valid addresses to be invalidated if the filter entry is updated by another address mapped to the same entry. Another source of imprecision occurs when the signature is transferred well after the lock release occurs. Such a signature may include addresses to accesses after the release and before the subsequent acquire – these do not precede the acquire by happens-before and may lead to false positives and unnecessary invalidations. Our evaluation, however, showed that such cases did not occur often for applications with reasonable lock synchronization; nevertheless, we later discuss some approaches to mitigate such effects (Section 6).

End of phase actions: At the end of a phase, as with DeNovo, we insert self-invalidation instructions for all regions with writable effects in that phase. This includes atomic and non-atomic regions. Analogous to DeNovo, all data in such regions is invalidated unless it is registered or its *touched* bit (for non-atomic regions) is set or its *touched-atomic* bit (for atomic regions) is set. All *touched* and *touched-atomic* bits are reset at the end of the phase and all Bloom filters are cleared.

4.2 Lock Implementation

Tables 1a and 1b describe the state transitions for the L1 and L2 caches respectively for lock words, building on top of the DeNovo line protocol (as with DeNovo, the coherence states are at word granularity). We next discuss these in detail.

L1 transitions: There are two states at L1 for a lock word: *LockQ* and *Invalid*. The lock word transitions to *LockQ* on receiving a lock request from its core, and stays there until it transfers the lock (along with the access signature) to *nextPtr* or until the line is evicted. While in *LockQ* state, a bit in the data portion of the lock entry, called *Locked*, indicates whether the lock is held or released. Figure 2 shows the lock word layout at the L1 with a lock queue.

On a lock request by a core, its L1 sets the *Locked* bit for the corresponding word. If the word was already in *LockQ* state, the L1 informs the core of a successful lock acquire. If the previous state was *Invalid*, a lock request is sent to the L2 and the core is stalled (the cache does not service any further requests from the core) until the response is received.

On an unlock request to *LockQ* state, if *nextPtr* is not null, the L1 transfers the lock to the *nextPtr* core and transitions to *Invalid*. Otherwise, it unsets *Locked*. An unlock request to *Invalid* state generates a request to the L2. This request is simply a notification and does not bring back the cache line (the state stays *Invalid*).

An L1 in *LockQ* state may receive a remote lock request forwarded by the L2. If the *Locked* bit is set, the request is queued in *nextPtr*; otherwise, it is serviced immediately by transferring the lock and changing the state to *Invalid*. The L1 may also receive a remote lock request in *Invalid* state due to a previous writeback. If this request is only for the signature, it transfers the signature (along with an implicit lock transfer) to the remote requestor. If the request is for the lock as well, then it signifies a race between the L1's writeback and the remote request at the L2. In this case, L1 returns a Nack to the L2 – we discuss how the L2 responds to the Nack in detail below.

Eviction of lines with lock words at the L1 is similar to DeNovo's L1 evictions (not shown in Table 1a). The main difference is that the writeback message needs to indicate which words are in *LockQ* state so that the L2 can perform appropriate action as discussed below. Table 1a does not show any action for writeback requests generated by L2 for L1. This is because the L2 does not need to maintain inclusion with the L1 for lock words (similar to *Valid* data in DeNovo). The distributed lock queue constructed in the L1s stays valid and does not need to be rebuilt on an L2 writeback.

L2 transitions without L1 writebacks: The L2 has two states – *Invalid* and *Valid*. The main source of complexity at the L2 comes from L1 writebacks of *LockQ* words; we therefore first discuss L2 transitions without L1 writebacks, indicated by *WB*=0 in Table 1.

On a lock request in *Valid* state, the L2 forwards the request to its *tailPtr* core and updates the *tailPtr* with the requesting core's ID. A lock request in *Invalid* state allocates the line for the lock word, triggers a fetch from memory, and keeps the L2 in *Invalid* state. When the response returns, the L2 transitions to *Valid* and applies the actions for the *Valid* state to the lock request (i.e., forwards the

	Lock request from core *i*	Unlock request from core *i*	Response for lock request from core *i*	Remote lock request from core *k*
LockQ	set *Locked*	**if** *nextPtr* != null send response to *nextPtr*; go to *Invalid* **else** unset *Locked*	unstall core *i*; merge received signature	**if** *Locked* is set *nextPtr* := *k* **else** send response to core *k*; go to *Invalid*
Invalid	stall core *i*; update tag; go to *LockQ*; set *Locked*; send lock request to L2 (writeback if needed)	send unlock request to L2	X	**if** sig-only request send response to core *k* **else** send *Nack* to L2

(a) L1 cache for core *i*

	Lock request from core *i*	Unlock request from core *i*	Lock/Unlock/WB/Nack response from memory for core *i*	Lock writeback from core *i*	Nack from core *i* for core *k*
Valid	**if** *WB* == 0 fwd req to *tailPtr*; **else** // *WB* == 1 **if** *Locked* is not set send sig-only req to *lastAcquirer* for *i*; *WB* := 0 **else** // *Locked* is set **if** *firstWaiter* != null fwd req to *tailPtr* **else** *firstWaiter* := *i*; *tailPtr* := *i*	**if** *firstWaiter* != null send sig-only req to *i* for *firstWaiter*; *WB* := 0 **else** unset *Locked*	X	**if** *firstWaiter* == null copy *Locked* from WB message; *lastAcquirer* := *i*; *firstWaiter* := *nextPtr*; *WB* := 1; **else** // race **if** *Locked* is not set send sig-only req to *i* for *firstWaiter*	**if** *WB* == 0 *firstWaiter* := *k* **else** **if** *Locked* is not set send sig-only req to *lastAcquirer* for *k*; *lastAcquirer* := null **else** *firstWaiter* := *k*
Invalid	update tag; send data req to memory; (writeback if needed)	update tag; send data req to memory; (writeback if needed)	**if not** tag match allocate line; update tag; (writeback if needed) go to *Valid*; apply actions for Lock/Unlock/WB/Nack as specified in *Valid*	update tag; send data req to memory; (writeback if needed)	update tag; send data req to memory; (writeback if needed)

(b) L2 cache

Table 1: State transitions for a lock word. *X* indicates unreachable states.

request to *tailPtr*). If the line was deallocated between the request and the response due to eviction, another line is allocated and the above action taken.

An unlock request in *Valid* state can only occur if the unlocking L1 previously performed a writeback on the lock (i.e., *WB=1*), and so is discussed below.

Writebacks generated by the L2 to memory are similar to DeNovo. As we see below, all the lock queue related information needed at the L2 is maintained as part of the lock word in the L2 – on an L2 writeback, this information is simply preserved at memory and made available to the L2 for later use.

Handling L1 lock writeback at the L2: When the L2 receives a writeback from an L1, it must ensure that it stores all information needed to construct the lock queue that was stored at the L1. This information is stored in the data portion of the L2 along with the *tailPtr*. An L1 writeback containing a lock word can originate only from the head of the lock queue in *LockQ* state because other cores are either stalled on their lock request or invalidated after transferring the lock. The L2, therefore, stores the following information in its data portion on an L1 writeback from core *i* (Figure 2 illustrates the L2 data layout with example values before and after the writeback):[3]

WB: The *WB* bit is set to 1 to indicate that the lock has been evicted from the L1 of the head of the lock queue.

Locked: The *Locked* bit from the writeback message is copied into the L2 to indicate whether the lock was released (*Locked*=0) at the time of the writeback.

lastAcquirer: L2 sets *lastAcquirer* as *i*. This is used to forward the next lock requestor to core *i* to obtain the access signature.

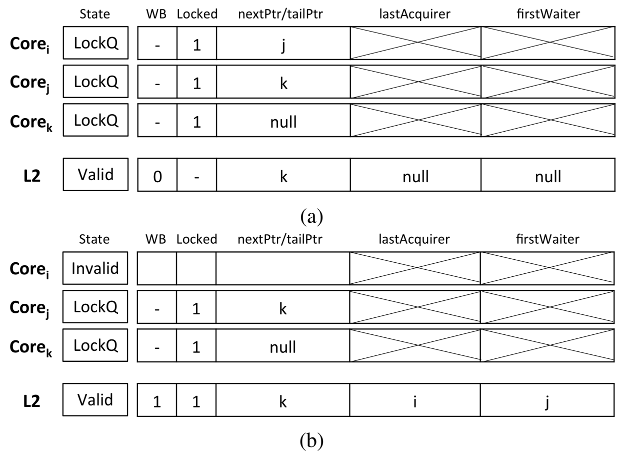
firstWaiter: L2 copies *nextPtr* from the writeback message into its *firstWaiter* field to indicate the first element in the queue after the head. On a subsequent unlock, the lock must be transferred to the *firstWaiter* core if it is not null.

Next we revisit the transitions for various messages at the L2 when the *Valid* state has *WB*=1. On a lock request, if *Locked* is not set (writeback occurred after lock release), L2 forwards the request to the *lastAcquirer* core. This request is for the access signature only since we already know that the lock has been released. If *Locked* is set (writeback before release), then L2 checks if *firstWaiter* is null. If it is not null, then L2 queues the request by forwarding it to *tailPtr*. Otherwise, it sets *firstWaiter* to *i* since there is no other waiter in the queue.

Similarly for unlock requests, if *firstWaiter* is not null, L2 forwards the request of *firstWaiter* to *lastAcquirer* for the signature (and implicit lock transfer). Otherwise, the queue is empty. L2 resets *Locked*, indicating that the evicted head is unlocked now and is ready to transfer the lock.

Handling races: There can be a race between an L1 lock writeback from core *i* and a request for the same lock from another core *k*. Thus, before getting the writeback, the L2 can forward core *k*'s request to L1. In this case, L1 nacks the request back to L2, which takes the following actions depending on whether it has already received the writeback (last column of Table 1b):

[3] Storing these fields in the data bank of the L2 does not limit the number of cores that can be supported as we can increase the data size of a lock variable as needed.



	State	WB	Locked	nextPtr/tailPtr	lastAcquirer	firstWaiter
Core$_i$	LockQ	-	1	j	☒	☒
Core$_j$	LockQ	-	1	k	☒	☒
Core$_k$	LockQ	-	1	null	☒	☒
L2	Valid	0	-	k	null	null

(a)

	State	WB	Locked	nextPtr/tailPtr	lastAcquirer	firstWaiter
Core$_i$	Invalid				☒	☒
Core$_j$	LockQ	-	1	k	☒	☒
Core$_k$	LockQ	-	1	null	☒	☒
L2	Valid	1	1	k	i	j

(b)

Figure 2: Example showing L1 and L2 data layout for the distributed queue-based lock (a) before writeback and (b) after writeback.

The Nack arrives before the writeback (WB=0): L2 simply sets *firstWaiter* to core *k*. When the writeback arrives, L2 finds its *firstWaiter* is not null and its request must be handled. If the *Locked* bit in the writeback is unset, L2 knows the lock was released and so can forward *firstWaiter*'s request to core *i* for signature transfer. If the *Locked* bit is set, then nothing needs to be done; the lock transfer to core *k* will occur when the *Unlock* arrives.
The Nack arrives after the writeback (WB=1): L2 services core *k*'s request using the information stored in the writeback; if *Locked* is not set, the request is forwarded to *lastAcquirer*. Otherwise, *k* is stored as the *firstWaiter*.

The above race is the only one that occurs in the lock protocol. It involves at most two cores and results in exactly one possible Nack message that the L2 immediately handles, with no deadlock or livelock causing actions.

4.3 Overheads

DeNovoND incurs the following overheads over DeNovo.
Hardware Bloom filter: There is one Bloom filter per core. A conservative upper bound for its size is the virtual memory size. In practice, an effective size can be empirically determined by measuring the number of atomic writes to distinct addresses in various applications. The size must also be large enough to have tolerable false positive rates. In our system, a relatively small size Bloom filter of only 256 bits worked well and provided performance similar to an infinite size Bloom filter for most cases. This is because the size of the key domain is restricted only to the addresses in atomic regions, and the filter is flash cleared at the end of a phase.

The quality of the hash function also impacts the efficiency of Bloom filters [42]. We experimented with two hash functions, multi-bit selection (similar to the one used in [16]) and H_3 (universal hash function that provides uniformly distributed hash values [15]), which showed consistent performance across applications. For our evaluation, we used H_3 which worked better with applications with high false positive rates. Finally, [16] has shown that Bloom filter operations of element insertion, membership query, and flash clear can be implemented very efficiently in hardware.
Storage overhead: Our distributed queue-based lock protocol reuses the L1 and L2 cache data banks to store the waiter queue information, incurring zero storage overhead for that purpose. It requires one additional state *LockQ* at L1 to distinguish between lock and data words. This does not result in any added storage overhead for L1 state as DeNovo already requires two bits per word for storing three states (*Invalid, Valid, and Registered*). With an additional *LockQ* state, we now have four states stored in two bits. The two L2 states for lock words can reuse the L2 per-word state bit of the baseline DeNovo protocol – lock words simply add new transitions to the existing L2 states, triggered by lock related messages. Thus, the lock protocol does not incur any additional storage overhead. The externally visible protocol states for data accesses also stay the same as for DeNovo. For efficient tracking of atomic writes, however, we added a *touched-atomic* bit per word in the L1 as an additional state bit (used only by the local core).
Communication and computation overhead: On acquire/release, the Bloom filter of the releaser is piggybacked on the lock transfer message. In order to minimize impact on network traffic, we can compress the Bloom filter using run-length encoding as in [16] or a Bloom-filter specific compression technique [38]. In our evaluations, we conservatively do not model such compression and charge the full 256 bits (32 bytes) of network traffic for the Bloom filter at a lock transfer. When a core receives a lock transfer message along with the signature, it needs to merge the received Bloom filter with its own before executing memory instructions in the critical section. The time for merging can be partially hidden by not blocking the execution until the first write/read instruction to an atomic region is issued.

For the distributed queue-based lock, there is an additional overhead for writeback messages which need to include an additional bit per word to indicate if the word is in *LockQ* state so that the L2 can perform appropriate lock related actions for this word. This overhead, however, can be compensated by observing that the writeback message does not have to contain full lock words, but only the *Locked* and *nextPtr* parts. The queue-based lock protocol also requires new state transitions in response to lock related messages; however, these do not introduce any new transient states or interact with the data protocol and can be separately verified.

5. Evaluation Methodology

5.1 Simulation Environment

For our evaluations, we use the Wind River Simics [34] full-system functional simulator to drive the Wisconsin GEMS detailed memory timing simulator [35] that we modified to implement our protocols. We also use the Princeton Garnet [3] interconnection network simulator to model network communication. To keep simulation times reasonable, as is common practice, we employ a simple, single-issue, in-order core model with blocking loads and 1 CPI for all non-memory instructions. (Note that DeNovoND does not require simple cores, but detailed timing simulation of a complex core would take an inordinate amount of time and we believe would not qualitatively affect our results.) We also assume 1 CPI for instructions executed inside the OS.

Table 2 shows the key parameters of our simulated systems. We simulate a multicore with 16 cores, a 64KB private L1 data cache per core (we do not model an Icache), a 16MB shared, NUCA L2 cache, and 4 memory controllers, all connected by a 2D mesh network. We configured the miss latencies to approximate those of the Nehalem processors [21]; e.g., a last-level shared cache miss (memory hit) costs 190 to 309 cycles on Nehalem (several of the latencies specify a range, depending on which L2 bank, remote L1 cache, or memory controller is accessed). We use the Bloom filter implementation shipped with GEMS [35] with the H_3 hashing function and 256 single-bit entries. We also simulated configurations with infinite Bloom filter entries for reference.

5.2 Simulated Systems

Our distributed queue-based lock is specifically designed for DeNovoND, reusing the coherence states of DeNovo, with no added transient states and limited race interactions. Implementing it on a conventional MESI-like protocol is possible, but will involve far more complexity to deal with interactions with the already existing numerous transient states and race conditions. On the other hand, comparing DeNovoND with distributed queue-based locks and MESI with conventional locking may not be fair to MESI. We therefore implemented simplified (idealized) queue-based locks that work for both MESI and DeNovo to isolate the effectiveness of access signatures. This idealized implementation maintains a "lock table" which is keyed by a lock variable address and maintains the waiter queue for each lock. Accesses to this table – creating an entry and grabbing the lock, adding a core to the waiter queue, waking up the first waiter in the queue, etc. – do not incur extra cycles. We also do not charge traffic overhead for lock and signature transfer for the idealized lock. Once a core is ready to release the idealized lock, lock transfer is instant and the next requestor wakes up immediately. Hence we evaluated the following systems:

MESI: We simulated MESI using idealized queue-based locks (MIL) and the `POSIX pthreads` mutex library (MPL). We modified the original implementation of MESI in GEMS [35] to support non-blocking writes for a fair comparison with DeNovoND where writes are non-blocking by default. Atomic instructions used in `pthreads` mutex codes are simulated using blocking store fences for correct execution.

DeNovoND: We simulated DeNovoND with idealized queue-based locks (DIL) and with distributed queue-based locks (DQL), both with a 256 bit Bloom filter (DIL-256 and DQL-256)) and, for reference, an infinite size Bloom filter (DIL-inf and DQL-inf). For DQL, operations on the lock incur latency consistent with table 2. For the signature transfer, we add a 256 bit (32 byte) payload to the lock transfer message and simulate network traffic and latency accordingly. This is conservative for DQL-256 since the signature could be compressed. It is aggressive but reasonable for DQL-inf since DQL-inf is intended to be a best case reference model.

5.3 Workloads

We evaluated 11 benchmarks with lock synchronization, taken from various suites to represent a range of behavior such as lock frequency, lock granularity, contention, critical section length, and shared working-set size. We evaluated *barnes* (16K particles), *ocean* (258×258), and *water* (512 molecules) from SPLASH-2 [45]; *fluidanimate* (35K particles) and *streamcluster* (8,192 points) from PARSEC 2.1 [8]; *tsp* (17 cities) as used in [12]; and *kmeans* (8,192 points, 24 dimensions, 16 centers), *ssca2* (2^{13} nodes), *genome* (256 nucleotides), *intruder* (1,024 traffic flows), and *vacation* (16,384 records) from STAMP [37].

The benchmarks from SPLASH-2 and PARSEC represent traditional applications designed and optimized to scale well with lock synchronization. The benchmarks from STAMP and *tsp*, however, were originally designed for hardware and software transactional memory. We ported them to use locks for our simulated systems. For short transactions, we directly replaced them with critical sections (*tsp*, *kmeans*, *ssca2*, and *intruder*). For longer transactions, we used finer-grained locks (*genome*, *vacation*).

We found that 3 out of the 6 transactional applications (*genome*, *intruder*, and *vacation*) spent $> 70\%$ of their execution time on lock acquire for all studied configurations. Clearly, parallelization using lock synchronization is inappropriate for these applications, for both MESI and DeNovoND. We therefore focus our results on the other 8 applications, referring to them as "lock-efficient" applications (Section 6.1). For completeness, we separately report results for the above three lock-inefficient applications (Section 6.2). We discuss optimizations to improve the performance of DeNovoND for the lock-inefficient applications, but fundamentally, these must be parallelized using different techniques for reasonable parallel speedups. Such techniques (including possibly transactional memory) are outside the scope of this work.

Core frequency	2GHz
# of cores	16
L1 data cache	64KB, 64 bytes (16 words) line size
L2 (16 banks, NUCA)	16MB, 64 bytes line
Memory	4GB, 4 on-chip controllers
L1 hit latency	1 cycle
L2 hit latency	29 to 61 cycles (bank-dependent)
Remote L1 hit latency	35 to 83 cycles
Memory hit latency	197 to 261 cycles
Network parameters	2D mesh, 16 bit flits
Bloom filter size	256 bits (infinite for reference)
hash function	4 H_3

Table 2: Simulated system parameters.

Finally, the lock-inefficient applications showed significant nondeterminism in execution time. Although our timing simulations are deterministic, they depend on the state of the system when the application is started (the Simics checkpoint at the start of the application). For different state, the lock-inefficient applications showed varying results. We therefore ran each such application with five different checkpoints for each system and averaged the results (the same five checkpoints are used for all systems). We also report the results for the lock-efficient applications averaged across three different checkpoints, but these applications did not show much variability across their checkpoints.

6. Performance Results

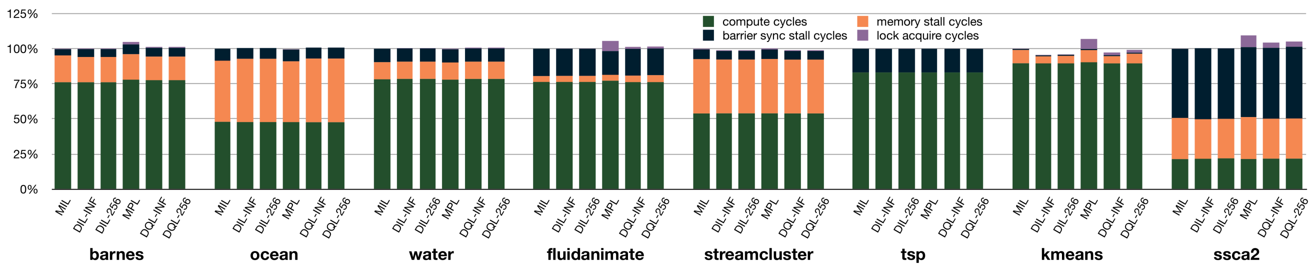
6.1 Lock-Efficient Applications

Figure 3a shows the execution time for our 8 lock-efficient applications for the 6 configurations described in Section 5.2. All bars are normalized to MIL. Each bar is divided into compute time, stall time due to data memory accesses (henceforth referred to as *memory time*), barrier time, and lock acquire time. Since we model non-blocking lock releases, lock release time is negligible. Since our focus is on the memory system, Figure 3b blows up the memory time in each bar of Figure 3a, divided into stalls for L1 misses resolved at L2, a remote L1, or main memory. Since all modeled systems implement non-blocking stores, virtually all memory stalls are due to loads. Figure 4a presents network traffic for the same applications on MPL and DQL-256 (normalized to MPL), classified by the message type: load, store, queue lock/unlock, writeback, and invalidation. The queue lock/unlock traffic exists only in DQL-256 for transferring distributed queue-based locks with signatures. For MPL, the lock traffic is aggregated with the data load and store traffic. Note that only MPL incurs invalidation traffic. We do not show network numbers with other configurations because they are idealized, but we confirmed that the network results for DQL-256 stay qualitatively similar even when compared to MIL.

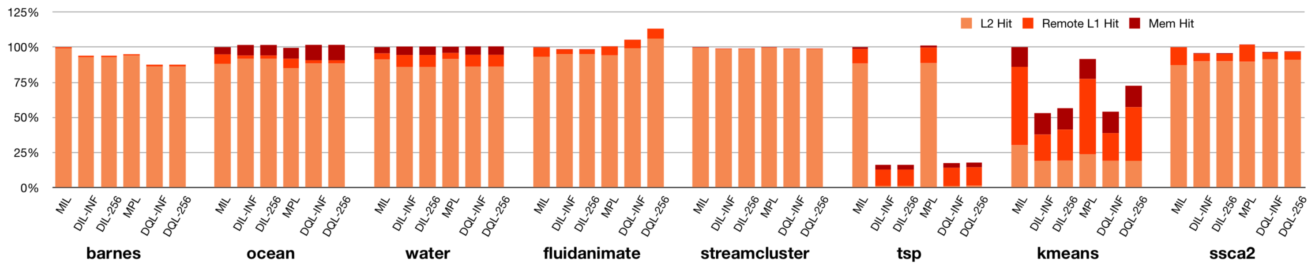
MIL vs. DIL-inf: For all 8 applications, DeNovoND shows the same or slightly better (up to 5%) execution time compared to MESI with idealized locks and infinite length Bloom filter. Focusing on memory time, again DIL-inf is either the same or better than MIL. For some applications, DIL is much better than MIL; e.g., 47% and 84% better for *kmeans* and *tsp* respectively. This is because MIL suffers from false sharing while DIL does not due to its per-word coherence state.

MPL vs. DQL-inf: Comparing the realistic lock implementations (but still with infinite Bloom filter size), we find that for all 8 applications, DQL-inf shows comparable or slightly better execution time than MPL. In fact, even compared to the idealized lock implementation in MIL, the execution time for DQL-inf is about the same or better in 7 of 8 cases and only 4% worse in the remaining case (*ssca2*). In terms of memory time, again DQL-inf is either comparable or sees large benefits due to the lack of false sharing relative to both MPL and MIL.

Impact of finite signatures: We next evaluate the impact of restricting the Bloom filter size: DIL-inf vs. DIL-256 and DQL-inf

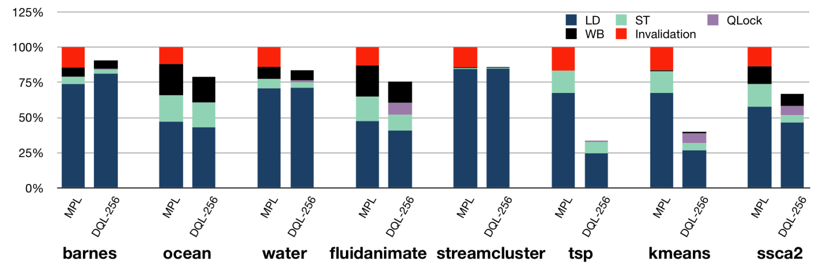


(a) Execution time.

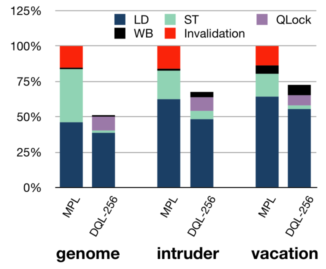


(b) Memory stall time.

Figure 3: Total execution time (a) and memory stall time (b) of lock-efficient applications on 6 configurations, normalized to MIL.



(a) Network traffic (lock-efficient).



(b) Network traffic (lock-inefficient).

Figure 4: Network traffic of all applications on MPL and DQL-256, normalized to MPL.

vs. DQL-256. The 256 bit Bloom filters show virtually the same execution times as the infinite length filters. In terms of memory time, the two Bloom filter sizes are similar for 6 of the 8 applications. For *fluidanimate* and *kmeans*, however, the 256 bit filter shows a degradation. For *kmeans*, memory time for DQL-256 continues to remain significantly better than for both MESI configurations (20% or more better), but for *fluidanimate*, it is worse by 13% (the only application where this is the case).

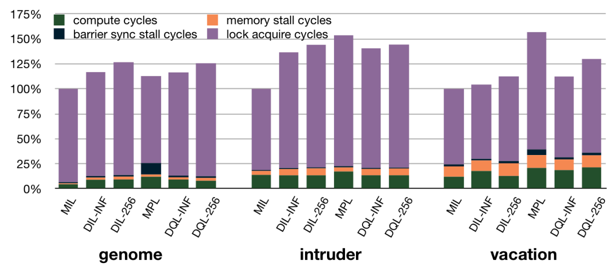
Fluidanimate and *kmeans* show the above behavior due to a confluence of a few subtle effects. First, both use critical sections where an atomic region address that is read is also written. Often an atomic region address read by a core was also last written by the same core (either in the previous phase or in a previous critical section). If this address is still in the core's cache in modified (for MESI) or registered (for DeNovoND) state, then the read will be a hit for both MESI and DeNovoND. Otherwise, if the address was written back, the read will be a miss for both MESI and DeNovoND. The difference between the protocols arises for any other atomic region addresses that come along with such a read miss as part of the same cache line. If the same core reads such an address in a subsequent critical section without an intervening write by another core, then MESI will still hit in the cache but DeNovoND will have to check against the Bloom filter. This could require a self-invalidation since the corresponding Bloom filter bit may be set, resulting in an extra miss over MESI. A smaller Bloom filter exacerbates this problem since it also results in false positives on the key domain. Further, the effect is more noticeable in DQL than in DIL because *fluidanimate* and *kmeans* have fine-grained locks – these locks pollute the cache and cause more replacements, exacerbating the above effect.

Network traffic: Figure 4a shows that for all the applications, DQL-256 has much lower traffic than MPL (33% on average, 67% maximum). This directly translates into energy reduction.

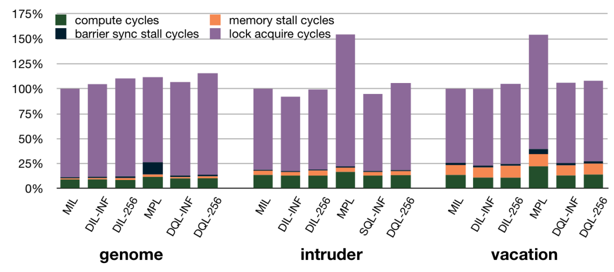
The primary sources of these savings in DeNovoND are as follows: (1) DeNovoND does not incur any traffic for invalidations, a significant effect in all applications. (2) Store traffic is reduced in some applications because store requests in DeNovoND do not bring in the cache line – they directly write into the L1 word and only send out a registration request for that word (multiple registrations for a given line are combined and sent on the network as mentioned in Section 2.2). (3) The net reduction in load misses (memory time) due to the lack of false sharing (Figure 3b) directly leads to lower load traffic in several applications. (4) Load traffic is further reduced because a load response only contains valid or registered words of a cache line. Since coherence state is preserved per word, some words may be invalid at the servicing cache.

A source for increased network traffic in DeNovoND is the 32 byte signature with all lock transfers. Figure 4a shows that this is small in all our applications. It can be further reduced through compression techniques mentioned in Section 4.3.

Summary: Overall, our results show that for these applications, the access signature mechanism allows DeNovoND to enjoy all the benefits of DeNovo even in the presence of lock-based synchronization. Further, the signature size needed is small (32 bytes).



(a) Baseline.



(b) "Write-Once" Atomic Region and Signature Clearing.

Figure 5: Total execution time of lock-inefficient applications on six configurations: (a) baseline, (b) with "write-once" atomic region optimization and signature clearing (threshold=99%) applied, normalized to the MESI with idealized locks (MIL) configuration.

6.2 Lock-Inefficient Applications

The lock-inefficient applications spend more than 70% of their time on lock acquires, but are presented here for completeness. Figure 5a shows their execution times analogous to Figure 3a. There are several ways in which these applications differ from the lock-efficient ones. First, as mentioned earlier, they are dominated by lock acquire time and so need a significantly different algorithm for parallelization and/or synchronization. These applications were originally designed to study transactional memory. Some of them use patterns for which lock-free synchronization is commonly used. Supporting such forms of parallelism and synchronization is outside the scope of this paper, but forms a key part of our future work.

Second, as discussed in Section 5.3, these applications show significant non-determinism. Although we report results averaged over five runs starting from five different Simics checkpoints (the same five checkpoints for each system), the variability makes comparing different systems difficult.

Third, we find that compute time varies across different systems for each of these applications. Although not shown here, a significant fraction of compute time comes from the OS (e.g., due to frequent memory allocations), forming the main source of the compute time variation. (The lock-efficient applications have negligible OS compute time.) Our results must therefore be understood in the context of the above caveats.

MIL vs. DIL-inf: For all three applications, DIL-inf shows observably worse performance than MIL (16% for *genome*, 36% for *intruder*, and 5% for *vacation*). A large part of the performance difference appears to come from acquire time; e.g., DIL-inf spends 40% more cycles waiting for lock acquisition than MIL with *intruder*. Though memory time is a very small portion, it affects acquire time by increasing the time spent within critical sections. Our detailed results show that DIL-inf suffers from higher memory time than MIL, especially for *genome* and *intruder*.

The higher memory time above occurs due to an access pattern where an address is written only once in a phase and then read several times. Specifically, *genome* and *intruder* use list and hash table data structures that store "data" or "key-data" pairs of each entry as a field of the entry object – in these programs, the data is initialized when a new element is inserted (within a critical section) but never modified afterwards. A core may read this data later in different critical sections – DeNovoND will self-invalidate on such reads since it does not know if there was an intervening write since the last read. MESI, on the other hand, will hit on such reads if they happen close enough to exploit temporal locality.

Section 6.2.1 discusses how we can use software information to remedy the above situation. We believe, however, that a better solution to this problem is a better synchronization construct – using locks for such reads is overkill. Such constructs in the context of DeNovo and DeNovoND are a key part of our future work.

MPL vs. DQL-inf: DQL-inf performs slightly worse than MPL with *genome* for the same reason as the comparison between MIL and DIL-inf. DQL-inf outperforms MPL with *intruder* and *vacation* – for these applications, MPL has significantly higher acquire time than MIL. MPL's pthread locks, however, are inherently inefficient with high lock contention; therefore, this is not a fair comparison for MESI. Thus, little can be deduced here except perhaps that DeNovoND performance seems to be in the same range as MESI (this inability to draw a conclusion is an inherent artifact of the problem studied).

Impact of finite signatures: With smaller Bloom filter sizes, false positives exacerbate the impact of the conservative invalidations described above; for *genome* and *intruder* – DIL-256 and DQL-256 perform worse than DIL-inf and DQL-inf by 4% to 10%.

Vacation does not suffer from the conservative invalidations of *genome* and *intruder*, but reveals a different source of inefficiency with smaller signatures. Figure 5a shows DIL-256 is 8% worse than DIL-inf, while DQL-256 is 17% worse than DQL-inf for this application. This is mainly due to its large working set of atomic data, which can increase the false positive rate if a Bloom filter is too small. In addition, *vacation* has only one phase without any barriers in between; thus the Bloom filters get filled up for a long period without clearing. This further exacerbates the false positive rate, resulting in unnecessary self-invalidations and higher memory times. Section 6.2.1 describes an optimization technique called signature clearing to deal with this issue.

Network traffic: Figure 4b shows network traffic of the lock-inefficient applications on MPL and DQL-256. DQL-256 generates less network traffic (up to 48%) than MPL for all three applications for reasons similar to that for the lock-efficient applications. In addition, with relatively high lock contention, repeated accesses to lock variables can generate increasingly higher network traffic in MPL. In contrast, distributed queue-based lock request/response traffic scales in proportion to the number of lock transfers.

6.2.1 Optimizations

Handling "write-once" atomic data: As with the case with *intruder* and *genome*, once a new entry is created and then inserted into a data structure (list, hash table, etc.), the "data" portion of the entry may remain read-only for the entire execution while other fields of the entry are modified as the structure grows or shrinks. In this case, classifying the "data" as atomic makes every self-invalidation after the very first one (the memory location may have been used and freed before) unnecessary.

DeNovoND can safely get rid of these invalidations by identifying such atomic accesses as made to a "write-once" atomic region. In addition to general information about atomic regions and effects, software can allow such "write-once" atomic data to be marked differently by using a special region ID or a special op-code for the write. Then DeNovoND can exploit it to prevent such data from being self-invalidated as follows. If the data is known to be in a "write-once" atomic region, DeNovoND does not reset its *touched-atomic* bit on lock transfer; therefore, when the data is accessed (read) again later, it is treated as if it has been already accessed in the same critical section (with *touched-atomic* bit set) and will not be self-invalidated, thereby eliminating several subsequent misses.

The write-once annotation can be considered to be a generalization of *final* variables in Java; a final variable can only be initialized once, either at the time of declaration or by the constructor of the class in which it is declared [40]. Our write-once variables must be written (at most) once per parallel phase.

Signature clearing: Depending on the atomic write-set size in a phase, the fixed-size hardware Bloom filter may get saturated (all bits set) before the phase is over. This drives the false positive rate very high, resulting in many unnecessary self-invalidations. Saturated Bloom filters can be flash-cleared by a simple hardware operation, but it also requires flushing out atomic words in the cache. Also, the fact that a signature has been cleared in the releaser should be propagated to the acquirer so that the acquirer can update its cache according to the new version of the Bloom filter. We implemented a signature clearing algorithm that carries a vector of clearing counters per core. When signature clearing is triggered on a core, its counter is incremented. The vector of clearing counters is transferred on a lock transfer along with the access signature. The acquirer compares the received vector with its own, and performs signature clearing if there exists an element in the received vector that has a larger counter than the corresponding element in its own vector. Before the lock is transferred again, the vector is updated to have up-to-date values.

Performance impact: Figure 5b presents execution times analogous to figure 5a, but with the above optimizations applied.

For *genome*, all DeNovoND protocols now perform comparable to the MESI counterpart. Our detailed results show large reductions in memory time from the write-once optimization (118% to 151%). Since this reduction mainly comes from atomic accesses within critical sections, lock contention also improved. *Intruder* shows similarly dramatic results in memory time improvement with consequently large improvements in execution time for the DeNovoND configurations; acquire time is reduced by 36 to 42%, memory time by 56 to 76%, and overall execution time by 43% on average.

For *vacation*, DIL-256 and DQL-256 (protocols with finite Bloom filters) show performance benefits from signature clearing; DIL-256 and DQL-256 were 17% and 8% worse than DIL-inf and DQL-inf respectively without signature clearing. With signature clearing, with 99% filter saturation percentage as the trigger for clearing, the difference is reduced to 5% and 2%.

Overall, the optimizations are quite effective, making the DeNovoND protocols comparable or better than the corresponding MESI protocols even for the lock-inefficient applications.

7. Related Work

There has been much research on improving the performance of memory consistency models by guaranteeing consistency only at synchronization points. Our work is closest to that of lazy release consistency (LRC) [27], entry consistency (EC) [7], and scope consistency (ScC) [22]. A key focus of these models is saving invalidation network traffic by postponing propagation of modified data until an acquire. LRC maintains consistency of all shared data at every lock transfer. EC attempts to reduce traffic by requiring programmers to bind every shared object with a lock, and transferring only the bound data objects on a lock transfer. ScC attempts to relax the strict and explicit bindings between data and lock in EC; instead, it uses "consistency scope" to implicitly associate data and the acquire/release pair protecting the data. DeNovoND is similar in that it also assumes a software guarantee for data-race freedom and association of atomic regions and sections. However, a key difference between DeNovoND and the above models is that the latter are designed for software distributed shared memory, keeping coherence information at a coarse-grained page granularity and storing information about modified data in data structures in user space. DeNovoND focuses on tightly coupled multicores with different trade-offs. In particular, DeNovoND implements a much simplified yet effective scheme for tracking modified atomic locked data in hardware, while leveraging the feature of the baseline system (no invalidation traffic) for non-atomic data.

REFLEX [33] employs software distributed shared memory with release consistency to make it easier to program low-power smartphones. It uses either eager or lazy update propagation depending on the initiating core's power profile. While REFLEX concentrates on adapting release consistency for low power on heterogeneous systems, DeNovoND is a more general solution that addresses complexity, performance, and power.

The recent SARC coherence protocol [25] also exploits the data-race-free programming model, but their goal is to improve the conventional directory-based protocol [2]. SARC self-invalidates "tear-off, read-only" (TRO) copies of data to save power. However, SARC does not eliminate directory storage overhead or reduce protocol complexity like DeNovoND and its baseline system. Also, the concept of touched bit, which plays an important role in DeNovo and DeNovoND is not present in SARC.

Other efforts to improve coherence achieve one or more of our goals at the expense of other goals. [31] introduces more complexity for self-invalidations and [36] requires writes to go to a shared cache if there are potential conflicts. SWEL [41] and Atomic Coherence [44] rely on specific interconnect substrates to simplify their protocols. Rigel [28] and Cohesion [29] propose systems with accelerators using a hybrid memory model based on shared memory, and employ software-driven invalidation for coherence. However, Rigel eagerly writes back all dirty lines to the global shared cache at phase boundaries, causing potentially unnecessary and bursty network traffic. DeNovoND self-invalidates potentially stale blocks only, avoiding this unnecessary traffic. Cohesion does not address existing limitations of software and directory-based hardware coherence mechanisms. Its software coherence issues extra coherence instructions wasting cycles and network bandwidth since its coherence tracking is conservative and coarse-grained, while the hardware directory-based protocol has the same current complexity and scalability issues. In contrast, DeNovoND starts from a simple protocol and makes it easy to add various optimizations to improve performance and energy further without complicating the protocol.

We leverage much prior work on Bloom filters, which have recently been widely used for access tracking [16, 43, 46]. Typical prior such usage, however, uses filters in the range of 1K to 2K bits. DeNovoND is able to achieve competitive performance with 256 bits, with commensurately lower space and computation overheads, since its key domain is limited to atomic addresses.

8. Conclusion

This paper takes a significant step towards a vision for complexity-, performance-, and energy-efficient multicores enabled by disciplined shared-memory programming practices. Prior work on DeNovo showed how this vision could be achieved for deterministic programs. This paper develops DeNovoND, a system that additionally supports disciplined non-determinism with minimal additional overheads and complexity relative to DeNovo.

DeNovoND exploits a previously developed software-level guarantee that non-deterministic (atomic) data accesses are distinguishable and protected by a lock. The key insight is to use small and simple hardware Bloom filters to track and communicate such accesses across lock transfers, preserving DeNovo's previous advantages of no transient states, directory overhead, invalidation messages, or false sharing. Underlying the data transfer mechanism is a distributed queue-based lock mechanism that uses the cache data banks to construct a lock-waiter queue, without additional state bits or directory storage.

DeNovoND provides comparable or better performance than MESI with the lock-efficient programs studied here. Further, network traffic is significantly reduced, impacting energy. We also identified some patterns in lock-inefficient code that did not work as well with DeNovoND – we showed optimizations to mitigate those effects, but believe the correct solution lies in alternate forms of synchronization for such codes.

As future work, we plan to explore broadening the scope of our vision of hardware-software co-design rooted in disciplined programming to embrace further programming patterns such as pipelined parallelism and "lock-free" data structures, as well as support complex codes such as legacy codes and operating systems. Our ability to easily extend DeNovo to embrace lock based codes gives us further confidence in generalizing this vision.

Acknowledgments

This work was supported in part by Intel Corp. and Microsoft Corp. through the Universal Parallel Computing Research Center at Illinois, Intel Corp. through the Illinois-Intel Parallelism Center, and the National Science Foundation under grant CCF-1018796.

References

[1] S. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *CACM*, Aug. 2010.

[2] S. Adve and M. Hill. Weak Ordering - A New Definition. In *ISCA*, 1990.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha. GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator. In *ISPASS*, 2009.

[4] M. Allen, S. Sridharan, and G. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *PPoPP*, 2009.

[5] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *PLDI*, 2008.

[6] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *OOPSLA*, 2009.

[7] B. Bershad, M. Zekauskas, and W. Sawdon. The Midway Distributed Shared Memory System. In *Compcon Digest of Papers.*, 1993.

[8] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[9] B. H. Bloom. Space/Time Trade-offs in Hash Coding with Allowable Errors. *CACM*, 13:422–426, 1970.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, 1995.

[11] R. Bocchino, Jr., V. Adve, D. Dig, S. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A Type and Effect System for Deterministic Parallel Java. In *OOPSLA*, 2009.

[12] R. L. Bocchino, Jr., S. Heumann, N. Honarmand, S. V. Adve, V. S. Adve, A. Welc, and T. Shpeisman. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. In *POPL*, 2011.

[13] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh, and G. Varghese. An Improved Construction for Counting Bloom Filters. In *ESA*, 2006.

[14] Z. Budimlić, M. Burke, V. Cavé, K. Knobe, G. Lowney, R. Newton, J. Palsberg, D. Peixotto, V. Sarkar, F. Schlimbach, and S. Taşirlar. Concurrent Collections. *Sci. Program.*, 18(3-4), Aug. 2010.

[15] J. L. Carter and M. N. Wegman. Universal classes of hash functions (extended abstract). In *STOC*, 1977.

[16] L. Ceze, J. Tuck, J. Torrellas, and C. Cascaval. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *ISCA*, 2006.

[17] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou. DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism. In *PACT*, 2011.

[18] J. Devietti, B. Lucia, L. Ceze, and M. Oskin. DMP: Deterministic Shared Memory Multiprocessing. In *ASPLOS*, 2009.

[19] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct: A Flexible Parallel Programming Model for Tera-Scale Architectures, 2007.

[20] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *ASPLOS*, 1989.

[21] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *MICRO*. IEEE, 2009.

[22] L. Iftode, J. P. Singh, and K. Li. Scope Consistency: A Bridge between Release Consistency and Entry Consistency. In *SPAA*, 1996.

[23] Intel. The SCC Platform Overview, 2010.

[24] A. Kägi, D. Burger, and J. R. Goodman. Efficient Synchronization: Let Them Eat QOLB. In *ISCA*, 1997.

[25] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 2010.

[26] S. Keckler, W. Dally, B. Khailany, M. Garland, and D. Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31:7–17, 2011.

[27] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, 1992.

[28] J. H. Kelm, D. R. Johnson, M. R. Johnson, N. C. Crago, W. Tuohy, A. Mahesri, S. S. Lumetta, M. I. Frank, and S. J. Patel. Rigel: An Architecture and Scalable Programming Interface for a 1000-core Accelerator. In *ISCA*, 2009.

[29] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: A Hybrid Memory Model for Accelerators. In *ISCA*, 2010.

[30] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic Parallelism Requires Abstractions. In *PLDI*, 2007.

[31] A. Lebeck and D. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA*, 1995.

[32] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5), 2006.

[33] F. X. Lin, Z. Wang, R. LiKamWa, and L. Zhong. Reflex: Using Low-Power Processors in Smartphones without Knowing Them. In *ASPLOS*, 2012.

[34] 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. *Computer*, 35:50–58, 2002.

[35] M. M. K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset. *SIGARCH Computer Architecture News*, 2005.

[36] S. L. Min and J.-L. Baer. Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps. *TPDS*, 1992.

[37] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *IISWC*, 2008.

[38] M. Mitzenmacher. Compressed Bloom Filters. In *PODC*, 2001.

[39] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *ASPLOS*, 2009.

[40] Oracle. Java Language and Virtual Machine Specifications.

[41] S. H. Pugsley, J. B. Spjut, D. W. Nellans, and R. Balasubramonian. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. In *PACT*, 2010.

[42] D. Sanchez, L. Yen, M. D. Hill, and K. Sankaralingam. Implementing Signatures for Transactional Memory. In *MICRO*, 2007.

[43] A. Shriraman, S. Dwarkadas, and M. L. Scott. Flexible Decoupled Transactional Memory Support. In *ISCA*, 2008.

[44] D. Vantrease, M. H. Lipasti, and N. Binkert. Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In *HPCA*, 2011.

[45] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

[46] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood. LogTM-SE: Decoupling Hardware Transactional Memory from Caches. In *HPCA*, 2007.