DeNovoSync: Efficient Support for Arbitrary Synchronization without Writer-Initiated Invalidations [*]

Hyojin Sung and Sarita V. Adve

Department of Computer Science
University of Illinois at Urbana-Champaign
denovo@cs.illinois.edu

Abstract

Current shared-memory hardware is complex and inefficient. Prior work on the DeNovo coherence protocol showed that disciplined shared-memory programming models can enable more complexity-, performance-, and energy-efficient hardware than the state-of-the-art MESI protocol. DeNovo, however, severely restricted the synchronization constructs an application can support. This paper proposes DeNovoSync, a technique to support arbitrary synchronization in DeNovo. The key challenge is that DeNovo exploits race-freedom to use reader-initiated local self-invalidations (instead of conventional writer-initiated remote cache invalidations) to ensure coherence. Synchronization accesses are inherently racy and not directly amenable to self-invalidations. DeNovoSync addresses this challenge using a novel combination of registration of all synchronization reads with a judicious hardware backoff to limit unnecessary registrations. For a wide variety of synchronization constructs and applications, compared to MESI, DeNovoSync shows comparable or up to 22% lower execution time and up to 58% lower network traffic, enabling DeNovo's advantages for a much broader class of software than previously possible.

1. Introduction

Both software and hardware in today's multicore systems face a major challenge in efficiently exploiting parallelism. Shared-memory remains a popular programming model due to its global address space, but it is unfortunately plagued with complexities arising from undisciplined programming practices; e.g., data races, unstructured parallelism with implicit side-effects, and unstructured non-determinism. Such "wild" shared memory behavior undermines the efficiencies that can be achieved through parallelism. It makes it difficult to test and maintain software as well as complicates hardware, preventing energy-efficient scaling.

Recently, there has been active research in exploiting information about memory accesses to improve the complexity and performance of hardware coherence and consistency [10, 11, 32, 35]. Specifically, the DeNovo project [10, 21, 35, 36] showed that hardware coherence can be simpler and more efficient if shared-memory programs are more "disciplined;" i.e., if parallelism is explicitly requested through structured parallel constructs with memory side-effects statically or dynamically identified. DeNovo eliminates directory storage for sharer lists, writer-initiated invalidation traffic, false sharing, and protocol transient states. Compared to the state-of-the-art MESI protocol, DeNovo offers a more complexity-, performance-, and energy-efficient solution.

DeNovo has primarily focused on supporting data accesses, assuming restricted forms of synchronization (global barriers for determinism in the original DeNovo [10] and disciplined locks for safe non-determinism in the later DeNovoND [35]) supported with special hardware support. For DeNovo to be widely adopted, however, it must relax its software requirements and support programs with arbitrary synchronization without expensive hardware support. This paper proposes DeNovoSync, an extension to DeNovo that supports programs with arbitrary synchronization without sacrificing DeNovo's benefits.

The primary insight behind DeNovo's efficiency is that for data-race-free programs with restricted synchronization, it is possible to determine statically (for deterministic programs) or dynamically (for disciplined non-deterministic programs) which memory locations are protected by the allowed forms of synchronization. DeNovo uses software-initiated self-invalidation instructions at the reader's cache to invalidate stale cached data in lieu of writer-initiated invalidations in traditional coherence protocols (e.g., MESI). This enables DeNovo to eliminate the overhead of sharers lists in directories, useless invalidation and ack messages, and protocol complexities due to myriad transient states caused from protocol races.

[*] This work was supported in part by Intel Corp. through the Illinois/Intel Parallelism Center at Illinois, by the National Science Foundation under grants CCF-1018796 and CCF-1302641, and by the Center for Future Architectures Research (C-FAR), one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey.
Copyright © 2015 ACM 978-1-4503-2835-7/15/03…$15.00.
http://dx.doi.org/10.1145/2694344.2694356

The challenge with synchronization accesses is that they are *inherently "racy"* and depend on writer-initiated invalidation for a reader to see a new value. DeNovo's use of reader-initiated self-invalidations worked well for race-free data, where appropriate self-invalidations (to known writeable regions) at synchronization points ensured that a data read would see any newly written value (which would be written before the synchronization). Racy synchronization accesses, however, do not have any a priori knowledge of when a value may be updated. DeNovoND solved this problem for well-structured locks by building special hardware for such locks [35]; however, it is unclear how such a custom approach could be used for arbitrary synchronization. To our knowledge, no previous systems without writer-initiated invalidations allow caching of synchronization variables for arbitrary synchronization constructs We could revert back to writer-initiated self-invalidations with directories for sharers lists only for synchronization variables, but this would eliminate the simplicity advantage of DeNovo.

This paper proposes DeNovoSync, a simple and efficient mechanism to support racy synchronization accesses on DeNovo, without impacting any of DeNovo's advantages. Specifically, we do not require adding any new states to the protocol and retain the properties of no writer-initiated invalidations and no sharers lists.

Our implementation is derived from a careful study of various common synchronization patterns and non-blocking data structures. We use sequential consistency as the correctness semantics for synchronization. From the conditions required for such semantics and the behavior of synchronization, we propose a single-reader constraint for such accesses, requiring synchronization reads to the same location to be serialized by registering for ownership. DeNovo already provides a single writer constraint requiring write serialization. Adding read serialization ensures that synchronization reads always get up-to-date values without writer-initiated invalidations. This already provides reasonable performance, but in some cases, incurs excessive read traffic under high read-sharing contention. Inspired by software backoff based algorithms, we judiciously integrate a hardware-backoff to reduce contention. We call the resulting protocols DeNovoSync0 (no backoff) and DeNovoSync.

Although our focus is on synchronization, we must provide a way to ensure data consistency as well. If there is no further information available from the program, we can simply self-invalidate all cached data (or shared writeable data) at a synchronization acquire, similar to the method used in [32]. With more information, we can perform selective self-invalidations as in DeNovo (using purely software information) or DeNovoND (using hardware signatures). Since our focus is on synchronization, we assume that we have software information similar to DeNovo to determine which data to self-invalidate at a synchronization acquire.

We compare DeNovoSync0 and DeNovoSync with a state-of-the-art MESI implementation for 24 synchronization kernels on 16 and 64 cores and for 13 applications on 64 cores (2 on 16 cores). For the kernels, for all but four out of the 48 cases studied, DeNovoSync shows comparable or lower execution time (22% lower on average) and lower network traffic (58% lower on average), which can be translated into energy savings. For larger applications where data accesses prevail, for all but one application, DeNovoSync provides comparable execution time (4% lower on average) and lower traffic (24% lower on average).

To our knowledge, this is the first work to show an efficient implementation of such a variety of synchronization patterns for a coherence protocol that does not rely on writer-initiated invalidations. The paper takes a large step in broadening the software that can enjoy the advantages of DeNovo and presenting DeNovo hardware as a complete and comprehensive software-aware hardware coherence solution.

2. Background

This section describes how disciplined programming languages drove the design of the previous DeNovo systems [10, 35]. We focus on the original DeNovo system for deterministic programs [10] since it covers most of the relevant concepts. We emphasize that DeNovoSync does not impose the software constraints in this section.

2.1 Software Assumptions for Original DeNovo [10]

Most disciplined programming models use some form of metadata about program structures and memory access patterns to prove certain desirable properties of programs. DeNovo used information in the Deterministic Parallel Java (DPJ) language [7, 8], as an example driver of the approach. (DeNovoSync does not assume DPJ.)

DPJ is an extension to Java that enforces deterministic-by-default semantics via compile-time type checking. DPJ provides (1) structured parallel constructs of *foreach* and *cobegin* supporting nested fork-join parallelism, and (2) a type and effect system for programmers to safely express common access patterns in object-oriented programs. Programmers assign a "region" to memory locations and annotate a method with "effects" summarizing the regions read and written by the method. The compiler uses this information to (1) type-check operations in the region type system and (2) ensure that no two parallel tasks conflict. With the above constraints, DPJ provides strong safety guarantees such as data-race-freedom and determinism.

2.2 Previous DeNovo Systems

DeNovo [10] divided the coherence problem for "data accesses" 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.

For (1), DeNovo uses DPJ-provided information on regions that could be potentially written in a parallel phase (each DPJ parallel construct forms a phase, with an implicit barrier at the join). DeNovo issues compiler-inserted self-invalidations for all writeable regions in the previous phase at phase boundaries, so that reads in subsequent phases cannot see stale data. (This requires the cache to store region information as described in [10].)

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 (LLC), the data bank of the cache doubles as the registry, storing either the data or a pointer to it (the core ID that last registered to write the data).

Thus, the DeNovo protocol has three states – *Registered*, *Valid*, and *Invalid*. These are analogous to the M, S, and I states in an MSI protocol. The key difference is that a real MSI implementation has numerous additional transient states to deal with races while DeNovo implementations have exactly three states (because there are no data races).

The key aspects of the protocol's operation are as follows (assuming a two level cache hierarchy without loss of generality) [10]. 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. 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 transient states) and generates a registration request. 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.

DeNovoND [35, 36] extended DeNovo to support disciplined non-deterministic data accesses using (i) dynamic access signatures to determine what data to self-invalidate at a synchronization point and (ii) a queue-based hardware implementation for locks. In this paper, we seek simple implementations for arbitrary synchronization patterns and leverage the original DeNovo style static self-invalidations for data consistency.

DeNovo [10] provides the following advantages: (1) There are no transient states and so it is much easier to verify and extend (incorporating optimizations did not introduce any protocol state changes) than MESI [10, 21]. (2) DeNovo does not rely on writer-initiated 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, performs comparably or better than MESI, and is more energy-efficient (since it reduces cache misses and network traffic). However, so far its efficacy has only been demonstrated for deterministic codes and codes with disciplined lock-based non-determinism.

3. Software Assumptions for DeNovoSync

As we extend DeNovo to support broader classes of applications, we relax our assumptions about disciplined software. We assume the following software properties, which are already enforced by most modern programming languages: (1) Software obeys the standard data-race-free memory model adopted by C++, Java, and other languages [9, 26] that define sequentially consistent semantics for data-race-free programs. (2) Software distinguishes between synchronization and data accesses (also part of the requirement of data-race-free). Mainstream languages all require this; e.g., programmers are required to use volatile (Java) or atomic (C++) declarations for synchronization variables [9, 26]. We assume this distinction is conveyed to the hardware as well.

DeNovoSync can run any applications that obey the above requirements. However, data consistency performance is improved with more information. Without further information, DeNovoSync will work correctly by invalidating all (shared, writable) data that is not registered in a core's cache at an acquire synchronization. However, with more information at acquires, these invalidations can be selective, possibly yielding better performance. This information may be in the form of compiler-specified shared writeable regions that are synchronized by the acquire (as in DeNovo [10]) or it may be generated dynamically through hardware signatures (as in DeNovoND [35]). In this paper, we assume the program provides static regions that need to be invalidated at an acquire. Although determining this information may be difficult in the general case,[1] for the programs we examined, it was generally easy to identify because most of them were written in a disciplined manner using high-level parallel constructs that made clear which data was being protected by which synchronization. We leave the question of how to (semi-)automatically (statically or dynamically) deduce such information in the general case to future work.

4. Protocols for Synchronization Accesses

Our focus here is on correctly implementing synchronization accesses; i.e., accesses involved in a race [1]. We assume the correctness criteria for synchronization is sequential consistency, the strongest hardware-level guarantee possible. Translating this to high-level safety properties when such synchronization constructs are used in the context of otherwise disciplined code is outside the scope of this work. Without loss of generality, we assume below that there is a two level cache hierarchy with private L1 caches and a shared L2 cache.

[1] To determine which region needs to be invalidated at an acquire more formally, we use the standard happens-before definition [1]. If X is the last conflicting write ordered before a read Y by happens-before, then either (1) X must be program ordered before Y, or (2) there must be an acquire A program ordered before Y such that X happens-before A and there is a self-invalidation for the region of X (which is the same as the region of Y) between A and Y. This ensures that Y never sees a "stale" value. (Note that a self-invalidation only affects non-registered data in the cache; registered data stays in the cache across synchronization boundaries.)

4.1 The DeNovoSync0 Protocol

The following conditions are sufficient for sequential consistency of synchronization accesses [1]:

- *Write propagation:* A write is eventually visible to all cores.
- *Write atomicity:* A write is not made visible to a read until it is visible to all cores.
- *Write serialization:* Writes to the same synchronization location are serialized (i.e., seen by all cores in the same order).
- *Program order:* A synchronization access must not be issued until the previous (by program order) synchronization access completes (i.e., a write is visible to all cores and a read returns its value).

We next describe how we modify the DeNovo protocol [10] to satisfy the above conditions.

Write propagation: Since we do not have writer-initiated invalidations, a synchronization read to a word in valid state will never see a new write. To ensure the write propagation condition, we need to perform periodic self-invalidation for such reads so that they miss and go to the last level cache and see the values of any newly registered writes. DeNovoSync0 always performs such a self-invalidation for synchronization reads to valid state; i.e., unless the word is in registered state, a synchronization read always incurs a miss.

Write atomicity: For write atomicity, DeNovoSync0 simply uses a single-reader protocol for synchronization (at the word granularity, which is the coherence granularity for DeNovo). Thus, a synchronization read is always required to register itself at the LLC and only one read can be registered at a time. Read registration produces read-read "races" in the protocol, which we discuss further below. Since all synchronization reads seek registration, once a synchronization write is visible to a synchronization read, no later read can see an older value. This ensures write atomicity.

Write serialization: DeNovo already provides write serialization by requiring writes to be registered and allowing only a single registered write to a given location. However, with synchronization, we can have write-write races, which the baseline DeNovo protocol did not have to handle. This is discussed below.

Program order: The program order requirement is easily met by not issuing a synchronization access until the previous synchronization access (by program order) is complete (i.e., registered).

Handling races: The baseline DeNovo protocol is built on the assumption of race-freedom; i.e., there are no concurrent conflicting accesses to the same location at any time. Specifically, when a registration request reaches the LLC in registered state, the LLC immediately updates the current registrant in its registry and forwards the new request to the previous registrant – the previous registrant invalidates itself and sends an ack to the new registrant without any need to inform the directory. In contrast, many MESI protocols (including the one we study) implement a blocking transaction where the directory serves as the intermediary for transferring the ownership from the previous to the new owner – until the full transaction is completed (with all invalidation acks collected), the directory does not service any new requests to that line to avoid even more complex transient states.

```
void queue.enqueue(value v):
   node *pw := new node(v, null)
   ptr pt, pn
   loop
(1)    pt := tail
(2)    pn := pt->next
(3)    if pt == tail
(4)       if pn == null
(5)          if (CAS(&pt->next, pn, pw))
                break;
(6)       else CAS(&tail, pt, pn)
(7)CAS(&tail, pt, pw)
```

(a) enqueue().

```
value queue.dequeue():
   ptr ph, pt, pn
   loop
   ph := head
   pt := tail
   pn := ph->next
   if ph == head
      if ph.p == pt.p
         if pn.p == null return <failure>
         CAS(&tail, pt, pn)
      else
         rtn := pn->val
         if CAS(&head, ph, pn) break
   free for reuse(ph)
   return rtn
```

(b) dequeue().

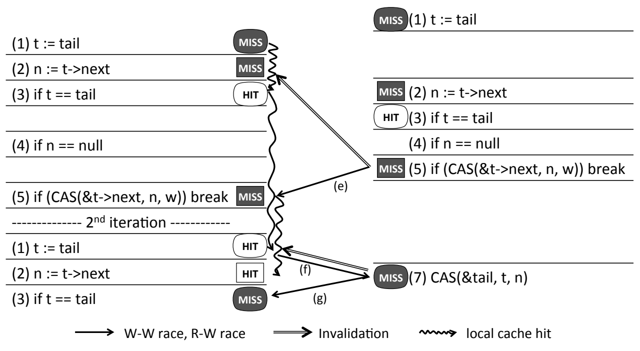
Figure 1: Pseudo code for Michael-Scott Queue.

We continue to maintain a non-blocking registry; i.e., the registry (LLC) forwards a new registration request to the previous registrant and continues to service other requests to the same word (including forwarding new registrations). This can result in an L1 cache receiving a forwarded registration request before it receives an ack for its own registration request by a remote L1. This is indicated by the word being in invalid state – the forwarded registration is simply stored in the MSHR entry of the pending registration. When the pending registration's ack arrives, the cache services the stored request in its MSHR, forwarding the registration ack (or data for a read) to the requesting core (keeping itself invalid). Thus, instead of serializing registrations at the LLC, we build a queue distributed among the L1 caches with pending registrations (similar to [12, 13, 34]). [2]

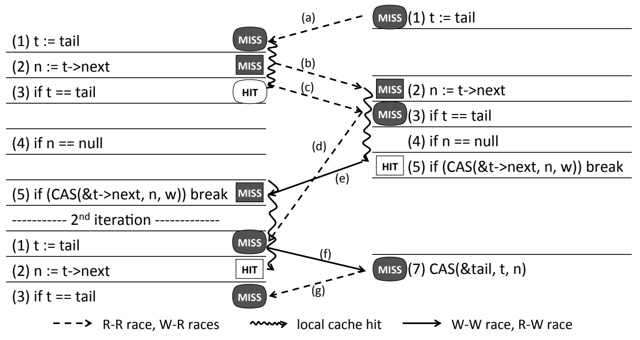
Summary and Example: In summary, the DeNovoSync0 protocol effectively treats a synchronization read like a read-modify-write (RMW), requiring registration. It does not add any new states to the protocol and requires only a few small changes to the actions on state transitions to accommodate registration request races.

To illustrate the working of the protocol, we apply it to an example. Figure 1 (adapted from [33]) shows pseudo-code for the enqueue and dequeue functions for the Michael-Scott queue [28]. Figure 2 shows two concurrent threads executing the enqueue function with MESI (part (a)), DeNovoSync0 (part (b)), and DeNovoSync (part (c), to be discussed later) protocols. We focus on the two synchronization variables, `tail` and `tail->next` here. All solid arrows represent legitimate invalidation of stale copy or ownership transfer by R-W and W-W races respectively on MESI and DeNovo, while dashed arrows indicate invalidation caused by read registration on false races (R-R and W-R races) for DeNovo. Rippled lines connect a read hit and a read/write that brought in the value. Rounded and rectangular boxes indi-

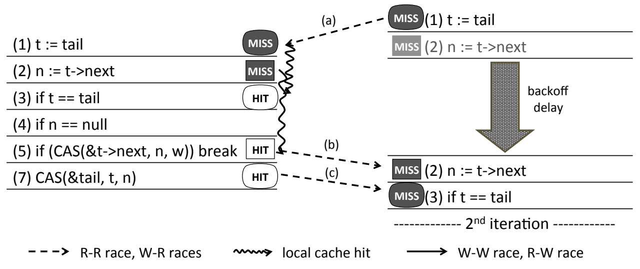
[2] We did not implement a non-blocking directory for MESI because of the much higher complexity. Exploring its potential impact on performance is part of future work.



(a) MESI.



(b) DeNovoSync0.



(c) DeNovoSync.

Figure 2: Example interleavings for enqueue() of Michael-Scott queue on MESI, DeNovoSync0, and DeNovoSync.

cate whether memory accesses to `tail` and `tail->next` in each line respectively result in a cache hit or miss.

Figures 2a and 2b show the following key differences between MESI and DeNovoSync0:

- False R-R and W-R races (edges (a), (b), (c), (d), and (g) in Figure 2b) trigger registration transfer and unnecessary invalidation of unchanged values for DeNovoSync0, while MESI does not incur any inter-thread communication for read accesses once valid copies are cached. These conservative invalidations on DeNovoSync0 cause the additional read misses in line 1 of the second iteration on thread 1 and line 3 on thread 2 relative to MESI.
- Rippled lines in Figure 2b show synchronization read/write hits enabled by earlier writes and read registrations by DeNovoSync0. Without an intervening read/write, thread 1 can safely re-read a valid copy of `tail` in line (3) and (2) of the second iteration in Registered state (similar to MESI). In addition, on thread 2, the rippled line from line (2) to (5) enables a write hit for DeNovoSync0 as `tail->next` is brought in Registered state in line (2). MESI, on the other hand, incurs a write miss on line (5).

Overall, compared to MESI, DeNovoSync0's read registrations suffer from two extra read misses but save a future write miss. Section 6 qualitatively analyzes the net impact.

4.2 The DeNovoSync Protocol

DeNovoSync0 is a reasonable solution for certain scenarios; e.g., with a single producer and consumer. However, in many scenarios, requiring all synchronization reads to register results in unnecessary misses. Specifically, with multiple waiting consumers, the synchronization data will ping-pong between the readers unnecessarily even while there is no intervening write.

The DeNovoSync protocol continues to require registration on synchronization reads, but attempts to delay such registration requests (in hardware) based on the perceived contention in the system. It is inspired by the idea of software backoff in conventional synchronization mechanisms [2]. Hardware backoff shares with software backoff the main goal of reducing contention in the system by delaying synchronization attempts. However, hardware backoff implements more fine-grained contention control than software backoff. Efficient, finer-granularity backoff is critical for many scenarios. For example, for non-blocking synchronization (e.g., the Michael-Scott queue described earlier), there are often several synchronization reads in succession and it is not reasonable to insert a software backoff before each of them. We therefore enhance the use and scope of conventional software backoff with a hardware backoff mechanism that adaptively delays synchronization reads to a location in non-registered state as follows.

4.2.1 Backoff Counter

DeNovoSync uses one hardware backoff counter per core to delay the core's synchronization read misses (i.e., reads to non-registered state). The size of the counter determines the maximum backoff cycles and should be determined by the system configuration; e.g., the number of cores, average miss latency, and network characteristics. The backoff counter logic performs the following operations.

Update counter: The backoff cycles value stored in the counter should adapt to the contention in the system. Our design uses *remote synchronization read requests* coming into a core as a symptom of system contention. On receiving a remote synchronization read request, a core downgrades from Registered to Valid state, sends a response to the remote requester, and increments the backoff counter. The increment is determined by a separate counter described in Section 4.2.2. The next time the core issues a synchronization read request to a value in Valid state, it stalls for the number of cycles indicated in the updated counter, delaying the issue of the synchronization read miss request. A synchronization variable in Valid state is not considered a usable valid copy. We simply reuse the state to differentiate from initial reads to Invalid state and trigger backoff delay. A synchronization read request to a value in Registered state will be a read hit and will not trigger backoff. When the counter overflows, it wraps around to zero.

Reset counter: The backoff counter is reset on a synchronization read to registered state (i.e., a read or RMW hit). Such a hit means that no other core intervened between the core's last access and the current access to the accessed location. We translate this as a low-contention status and do not delay subsequent requests.

Wakeup from backoff: When the backoff delay expires, it unstalls the core and issues the delayed synchronization read request. Since backoff delay is triggered only when the accessed location is in non-registered state, the core ignores any cached copy and issues a miss.

4.2.2 Increment Counter

The backoff counter efficiently adapts to changing contention levels in the system by increasing its value on a remote synchronization read registration request. However, a fixed increment value may not be flexible enough to handle both extremely low and extremely high contention. If the increment is fixed too low, the backoff counter may not climb fast enough to reduce unnecessary read registration requests under high contention. On the other hand, the increment of a large value may cause unnecessarily long stalls under low contention. Therefore, we introduce another level of adaptivity for the increment counter.

Update Increment: Again, we use the number of incoming remote synchronization read registration requests to indicate the contention level and update the increment counter accordingly. The size of the increment and the frequency of update should be determined by considering the system configuration. We found that the number of cores is a good indicator. For example, the increment counter is increased by the default increment value on receiving every 16th remote synchronization read registration request for the 16-core system evaluated in the paper (64th request for the 64-core system). The updated increment counter is then used to increase the backoff counter for the next remote read registration request.

Reset Increment: The increment counter is reset to a default value on a release. A release indicates the successful completion of a synchronization construct and the reset prepares the core to adapt for the next synchronization.

4.2.3 Example

Figure 2c shows how the Michael-Scott enqueue() function results in a different number of misses for the DeNovoSync0 and DeNovoSync systems. On the R-R race (a), the Registered copy of `tail` at thread 2 is downgraded to Valid state and the backoff counter is updated. When the thread tries to read `tail->next` in line (2), it recognizes the backoff counter is non-zero and starts the delay (assuming `tail->next` is in Valid state).

While the read of `tail->next` is being delayed on thread 2, thread 1 can proceed with its read and write of `tail` and `tail->next`. As a result, thread 1 can escape the loop in one iteration. Depending on when thread 2 wakes up, it may or may not fail the test in line (3). Compared to DeNovoSync0 in Figure 2b, DeNovoSync can save three misses in thread 1 – line (5) in the first iteration and (1) and (3) in the second iteration, allowing thread 1 to finish early. As will be seen in Section 7, DeNovoSync quickly adapts to different levels of contention in the system. Especially under high contention, contending cores can save on numerous read and write misses that are doomed to fail and repeat.

# of cores	16 cores	64 cores
Core frequency	2 GHz	
L1 data cache (private)	32KB, 64 bytes line	
L2 (shared, NUCA)	4MB, 16 banks	8 MB, 64 banks
	64 bytes line	
Memory	4GB, 4 on-chip controllers	
L1 hit latency	1 cycle	
L2 hit latency	28 to 68 cycles	28 to 140 cycles
Remote L1 hit latency	37 to 97 cycles	37 to 205 cycles
Memory hit latency	197 to 277 cycles	197 to 421 cycles
Network parameters	2D mesh, 16 bit flits	

Table 1: Simulated system parameters.

4.2.4 Protocol States

We still have only three protocol states for DeNovoSync/0 (meaning "both DeNovoSync0 and DeNovoSync" from now on): *Invalid, Valid* and *Registered*. When the word is Invalid or not present in the private cache, both reads and writes request registration and bring the address in Registered state. When the word is Registered, both reads and writes hit in the cache. While DeNovoSync0 makes the same transitions for Invalid and Valid state for all synchronization requests, DeNovoSync uses Valid state to trigger backoff on a synchronization read request. To this end, DeNovoSync distinguishes synchronization read and synchronization write requests and handles them differently in the protocol – synchronization write requests will be immediately issued in Invalid/Valid states while synchronization read requests may be delayed based on the backoff counter value.

5. Evaluation Methodology

5.1 Simulation Environment

For our evaluation, we use the Wind River Simics [25] full-system functional simulator to drive the Wisconsin GEMS detailed memory timing simulator [27] 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. We assume 1 CPI for all instructions executed inside the OS. We model 16 and 64 core systems. Table 1 shows the key parameters of our simulated systems.

5.2 Simulated Systems

We evaluated the following systems:

- **MESI:** We used the GEMS implementation of the MESI protocol [27], modified to support non-blocking writes for a fair comparison with DeNovo (where writes are non-blocking by default).
- **DeNovoSync0:** DeNovoSync0 implements the protocol as described in Section 4.1.

- **DeNovoSync:** DeNovoSync enhances DeNovoSync0 with the hardware backoff mechanism from Section 4.2, using the following parameters: 9-bit backoff counter with 1-cycle default increment for 16 cores and 12-bit backoff counter with 64-cycle default increment for 64 cores.

5.3 Workloads

We study several synchronization kernels as well as applications from standard benchmark suites. The kernels allow analyzing the new protocols in detail and showing that they can efficiently support a large variety of synchronization patterns. The applications are dominated by data accesses, but show that DeNovo can fully support common workloads.

5.3.1 Synchronization Kernels

We evaluated 24 synchronization kernels covering several lock-based concurrent data structures, non-blocking data structures, and barriers.

For lock-based structures, we adapted 5 kernels from [29]: single-lock queue, double-lock queue, stack, heap, and counter. The original kernels in [29] used Test-and-Test-and-Set (TATAS) locks. We also evaluated them using the more efficient array/list based queuing locks [4]. The critical sections of most of the above kernels have a small number of data accesses for one or two shared variables (e.g., pointers to queue head/tail or stack top) which allows us to isolate protocol behaviors for synchronization accesses. To cover a larger space, we also wrote a small kernel, *large CS*, with larger critical sections of fixed length, again both with TATAS and array locks. We did not use software backoff for any of the above kernels because our preliminary experiments with TATAS based kernels showed increased benefit of DeNovo over MESI for software backoff (further discussed in Section 7.1.1).

For non-blocking data structures, we adapted 6 kernels from [29]: Michael-Scott queue, PLJ queue, Treiber stack, Herlihy stack, Herlihy heap, and FAI counter. Each kernel has a software exponential delay in the range of [128, 2048) cycles to backoff after a failed attempt.

For barriers, we evaluated a static binary tree barrier, a static tree barrier with non-binary fan-in of 4 and fan-out of 2, and a centralized sense-reversing barrier derived from pseudo codes in [33]. To examine how the variance in the amount of parallel work between barriers can affect barrier performance, we evaluated a load-balanced and an unbalanced version for each barrier.

As mentioned, we insert region-based static self-invalidation instructions for data consistency. We ran 100 iterations of each kernel (1,000 for *FAI counter* due to its very small size) with dummy computations in between to spread out iterations. In each iteration, a queue, stack, or a heap kernel executes one insertion and one retrieval (or deletion) function call for one entry to/from the shared data structure, a counter kernel performs a single increment, and a barrier kernel executes two barrier instances around dummy computation. The length of the dummy computations is randomly chosen in the range of [1400, 1800) cycles for 16 cores and [6200, 6600) cycles for 64 cores, except that the unbalanced versions of barriers use [400, 2800) cycles for 16 cores and [1600, 11,200) cycles for 64 cores.

SPLASH-2	Input	**PARSEC**	Input
FFT	m16	fluidanimate	sim_small
LU	n256	blackscholes	sim_medium
barnes	8192	swaptions	sim_small
radix	524288	canneal	sim_small
water	512	ferret	sim_small
ocean	258	x264	sim_medium
		bodytrack	sim_medium

Table 2: Benchmark inputs.

5.3.2 Benchmarks

We evaluated 13 benchmarks from SPLASH-2 [37] and PARSEC 3.1 benchmark suites [6]. Table 2 shows the benchmarks and their input parameters. The benchmarks are chosen to represent programs with different synchronization patterns including locks, barriers, producer-consumer synchronization (pipeline parallelism), and aggressive lock-free synchronization. We show results with 64 cores for all except two benchmarks. Results for *ferret* and *x264* are for 16 cores because the simulation inputs provided do not fully utilize 64 cores concurrently. In all cases, we inserted region-based static self-invalidation instructions, both in the application code and in the POSIX thread library synchronization routines that were used.

6. Qualitative Analysis

This section qualitatively analyzes the performance of the different synchronization mechanisms on MESI and DeNovoSync/0, using the notion of *linearization points* [14]. Commonly used synchronization objects are *linearizable*; i.e., the operations on the object appear to occur in some total order that is consistent with the program order in each thread and also with any ordering observable by threads [33]. With linearizable synchronization objects, we can typically identify a linearization point within each method whose ordering determines the ordering of the method itself. To repeat on a synchronization failure, the linearization point for acquire synchronization is usually enclosed in a loop. This loop often includes synchronization accesses that are not linearization points; i.e., accesses that do not determine if an attempt is successful. For example, the first Test part of Test-and-Test-and-Set (TATAS) or equality checks before the final CAS instruction in non-blocking algorithms do access the shared synchronization object in the same loop, but their purpose is only to filter unsuccessful attempts early before reaching the linearization point.

To systematically understand protocol overheads for each synchronization mechanism, we divide the overheads into two parts: (1) *Linearization cost* is the cost to execute the instruction at the linearization point, including all coherence activities involved. (2) *Pre-linearization cost* is the cost of executing all the other synchronization accesses in the synchronization loop. Note that the coherence overhead for data accesses protected by these synchronizations may vary between the protocols, further affecting synchronization cost;

e.g., by changing contention. Further, DeNovo may inherently see lower traffic because it has word-based coherence state and does not transfer locations in a cache line known to be invalid. Since these overheads are orthogonal to the synchronization focus of this paper, we do not discuss them here, but they do affect our results in Section 7.

6.1 Locks

Atomicity synchronization ensures that a sequence of instructions executes as a single, indivisible unit [33], and its most common form is a lock. Here we explore the commonly used single variable based Test-and-Test-and-Set (TATAS) lock as well as the distributed array (or list) lock.

6.1.1 TATAS Locks

For individual TATAS lock acquire and release, we identify their linearization points at a successful Test-and-Set instruction for acquire, and the release write of a lock for release.
Linearization cost: When there are no or only two competing cores, the linearization cost for MESI and DeNovo is simply a two-hop or three-hop write miss acquiring the Modified (Registered) state from the shared LLC or the other core's cache respectively. As the number of contending cores increases, MESI is expected to have increasingly high linearization cost, since invalidating all cached copies increases the release write latency for the lock. While the release write may be overlapped with subsequent instructions at the releasing core, its latency is on the critical path of the next acquirer; the release write will be made visible to the next acquirer only after invalidations are complete for all cores (to ensure the write atomicity requirement for sequential consistency). In terms of network traffic, MESI incurs increasingly high network traffic for invalidation and acknowledgment messages. On the other hand, DeNovoSync/0 has no invalidation message overheads and only needs point-to-point registration transfer. Thus, DeNovo's linearization cost is expected to be independent of the number of competing cores and generally lower than MESI's.
Pre-linearization cost: Pre-linearization cost occurs only for acquires and consists of reads that perform preliminary checks to determine if it is worth proceeding to the linearization point. For MESI, waiting cores efficiently spin on a cached copy, incurring read misses only after the lock is released. For DeNovo, every synchronization read incurs a miss, unless the lock is Registered. Since registration can be revoked by remote synchronization reads, read misses by such false R-R races can be a source of performance inefficiencies for DeNovo. However, unless critical sections are very long, the percentage of false races relative to the total registration transfers will not be high since writes (CAS acquire and write release) by a winning core will frequently intervene between pre-linearization reads (which will trigger legitimate registration transfer). If critical sections are long and there are multiple waiting cores, DeNovoSync0 will suffer excessive ping-ponging of registrations between these cores, but DeNovoSync is expected to reduce this inefficiency through its hardware backoff.

Overall, for most cases, we expect that the linearization cost will dominate since it is on the critical path of the lock handoff; therefore, DeNovoSync/0 will perform better than MESI for execution time and network traffic. An exception is the case of long critical sections with multiple waiters, where DeNovoSync0's higher pre-linearization costs may or may not be offset by its lower linearization cost.

6.1.2 Array Locks

To reduce synchronization overheads for highly contended locks, array locks distribute synchronization points in space so that cores wait on different locations (array entries) in order. The linearization points are (1) a synchronization read in the acquire loop indicating the lock for the given entry is available, and (2) the release write of the next entry's lock.
Linearization cost: Array locks allocate a unique memory location for each acquiring core. Linearization involves setting of this location by the releaser and reading of the new value by the acquirer. For DeNovoSync/0, this is a simple 2 or 3 hop registration transfer for both the release and the acquire. For MESI, the critical path is similar (there are no excessive invalidations as for TATAS), but there are two subtle effects. First, the transactions are slightly more complex; e.g., the acquire involves a potential writeback to the directory and accompanying unblock messages for a blocking directory. Second, the successful acquire read is immediately followed by a write to reset the same lock so it can be reused in the next invocation. For DeNovoSync/0, this write is a hit since the lock is already registered by the acquire, but MESI needs to separately request ownership. This write latency for MESI can be on the critical path if the critical section is too small and the ensuing data accesses cannot overlap the latency. In that case, the write latency will be visible as an additional linearization cost for MESI. A read for ownership transaction in MESI would eliminate that cost, but our system does not implement such a transaction.
Pre-linearization cost: Pre-linearization includes spinning on the unique lock entry for each waiter. Since there is only one waiter per entry, both DeNovo and MESI spin on a cached copy, with minimal overhead.

Overall, for array locks, we expect comparable performance except for very small critical sections (where MESI may be worse) and higher traffic for MESI due to its more complex transactions and additional ownership request.

6.2 Non-Blocking Algorithms

It is common that non-blocking algorithms (1) linearize at their final CAS instruction, and (2) perform relatively many reads for equality checks to guarantee fast forward progress until the linearization point (e.g., Figure 1).
Linearization cost: Non-blocking algorithms provide atomicity of concurrent operations by linearizing cores that successfully execute the final CAS instruction. Similar to TATAS locks, MESI adds significant invalidation overheads for linearization cost with many competing cores, while DeNovo requires only a single point-to-point communication.
Pre-linearization cost: Non-blocking algorithms tend to have many repeated reads for equality checks on multiple

synchronization variables before the linearization point. In this case where synchronization read-to-write ratio is high, DeNovo with pessimistic reader-initiated self-invalidation is likely to suffer from spurious read misses from R-R and W-R registration transfer. While MESI can perform such reads without any cost unless there is a true race, we expect DeNovo to have high pre-linearization cost proportional to the number of synchronization read accesses.

Overall, under relatively low contention, DeNovoSync/0 may offset its high pre-linearization cost with its lower linearization cost compared to MESI. Under high contention, DeNovoSync0 will likely see the negative impact of pre-linearization cost while DeNovoSync can mitigate the impact with hardware backoff.

6.3 Barriers

For barriers, separate linearization and pre-linearization points can be defined for arrival and departure phases. Signaling arrival (e.g., incrementing the arrived core counter for centralized barriers or reversing and propagating parent's flag for tree barriers) and departure (e.g., reversing the sense(s) and propagating the updated sense to waiting cores) belongs to linearization, while pre-linearization includes unsuccessful checks (synchronization reads) of sense/flags (in a loop).

Linearization cost: For a centralized barrier, for arrival, both MESI and DeNovo incur high linearization cost by serializing cores incrementing the arrived core counter. MESI with blocking directory may suffer from extra queuing and messaging delays than DeNovo when the counter is severely contended. On departure, the release of a centralized barrier allows many readers to proceed. For MESI, this adds invalidation related overheads to linearization cost similar to TATAS locks. For DeNovo, we expect a potentially higher cost because all the waiting readers incur serialized registrations, adding to the critical path.

The linearization cost for both MESI and DeNovo is lowered with tree barriers since these reduce the number of threads synchronizing on a given location. Trees in tree barriers can be configured with different width and depth, depending on the number of child threads synchronized for the same parent thread. Regardless of the tree structure, there is only one reader and one writer for each node in the trees: an owner thread for a given tree node and its parent thread. For arrival, the owner thread updates its node (flag) and its parent thread reads it to see if the thread has arrived and vice versa for departure. Thus, the case of a distributed tree barrier is similar to that of an array lock, and we expect similar latency for both DeNovo and MESI.

Pre-linearization cost: Pre-linearization cost for barriers comes from synchronization reads by cores waiting for their flag to be reversed. Again, MESI efficiently spins from the local cache. DeNovoSync/0 pays similar costs for distributed tree barriers with only one reader per node. For scenarios with a large number of readers (i.e., centralized barrier), DeNovoSync0 sees a negative effect on network traffic through ping-ponging registrations, while DeNovoSync partly mitigates this effect through the hardware backoff.

Overall, MESI and DeNovo perform comparably for distributed tree barriers, behaving similar to array-based locks. DeNovo, however, can potentially suffer from higher pre-linearization and linearization costs with worse performance and traffic for centralized barriers. We note, however, that centralized barriers with many readers and one writer contending for a single variable is not a scalable synchronization pattern regardless of the coherence protocol. For best absolute performance, tree barriers with limited read sharing per synchronization variable are preferred. DeNovoSync is expected to be comparable to MESI in both execution time and traffic for such scalable barriers.

7. Results

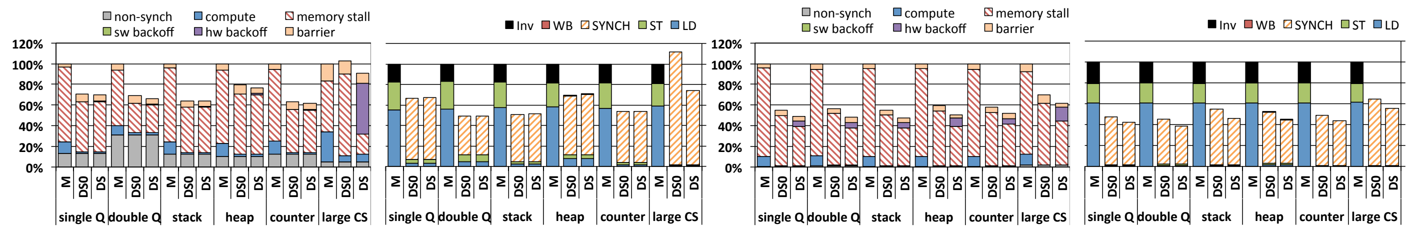
We next validate our qualitative analysis by evaluating a variety of synchronization kernels (Section 7.1) and applications (Section 7.2).

7.1 Results for Synchronization Kernels

Figure 3, 4, 5, and 6 show results for synchronization kernels using TATAS locks, array locks, non-blocking algorithms, and barriers respectively. In each figure, parts (a) and (b) respectively show the execution time and network traffic for MESI (denoted M), DeNovoSync0 (denoted DS0), and DeNovoSync (denoted DS) protocols, all normalized to MESI, for a 16 core system. Parts (c) and (d) show the analogous data for a 64 core system. We measure time in cycles and network traffic in terms of flit crossings across all network links (i.e., a flit going over one network link constitutes one unit of network traffic).

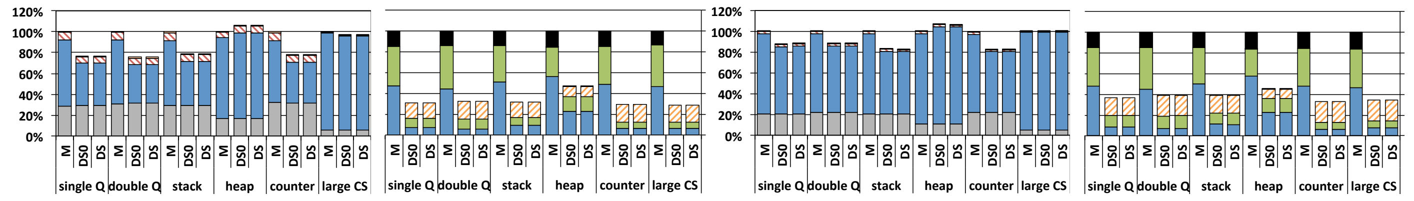
Parts (a) and (c) in each figure divide the execution time into multiple components. The gray (non-synch) component is the computation time spent between iterations of the studied synchronization kernels. These non-synchronization cycles are not affected by our techniques, but changing their length may increase/decrease the contention level. The rest of the components all represent the time spent strictly within invocations of the synchronization kernels for computation and memory accesses to data and synchronization variables (as described in Section 5, most kernels are dominated by synchronization accesses). These components consist of compute time (1 cycle per instruction, not including software backoff cycles), memory stall time for both synchronization and data accesses inside the kernel, software backoff time, hardware backoff time (only for DeNovoSync), and barrier stall time. Barrier stall time measures the time spent in the barrier at the end of the kernels for non-barrier kernels, which can indicate possible load imbalance caused by contention in synchronization methods. Note that a large part of compute time is from spinning synchronization read accesses (cache hits), so it can vary across protocols.

Parts (b) and (d) in each figure divide the network traffic based on the message type. For MESI, we show load, store, writeback, and invalidation (including ack) messages. For



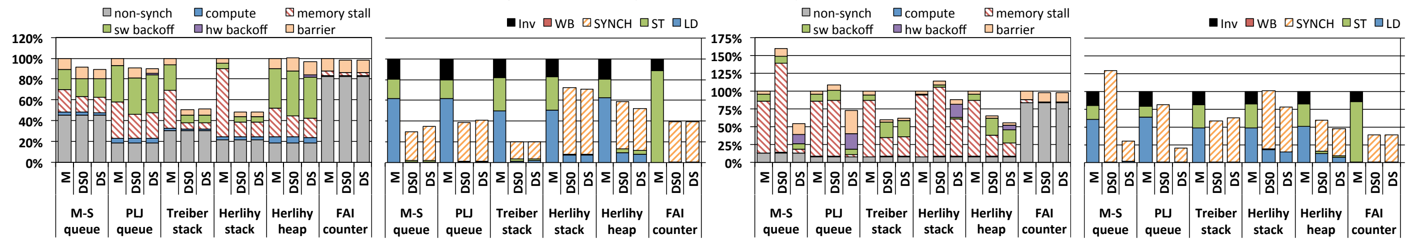
(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 3: Test-and-Test-and-Set (TATAS) locks based synchronization.



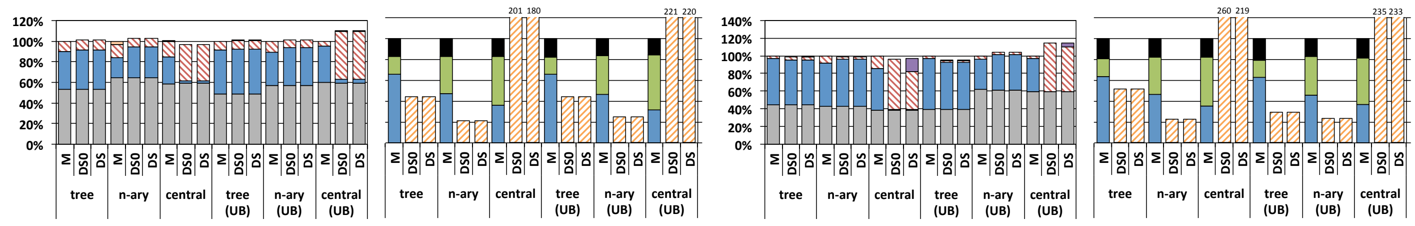
(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 4: Array locks based synchronization.



(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 5: Non-blocking algorithms.



(a) Execution time (16 cores) (b) Network traffic (16 cores) (c) Execution time (64 cores) (d) Network traffic (64 cores)

Figure 6: Barrier synchronization (UB = unbalanced computations).

DeNovo, we show data load, data store, synchronization (load, store, and RMW), and writeback messages.[3]

7.1.1 Test-and-Test-and-Set (TATAS) Locks

Figure 3 shows that across all synchronization kernels using TATAS locks, DeNovoSync outperforms MESI in terms of execution time (31% on average) and network traffic (42% on average) for 16 and 64 cores.

MESI vs. DeNovoSync0: DeNovoSync0 outperforms MESI on both systems except for *large CS* on 16 cores. The benefit in execution time ranges from 20% to 36% (average 25%) on 16 cores and an even higher 40% to 45% (average 41%) on 64 cores. Based on the analysis in Section 6, this benefit occurs because MESI incurs higher linearization cost in the form of invalidation/ack latencies on the critical path from lock release to the next successful acquire. For these kernels, which have small critical sections, the linearization cost dominates pre-linearization cost as explained in Section 6. Further, this cost increases in larger systems when more cores need to be invalidated, explaining the higher benefit for 64 core systems. For the *large CS* kernel, on the other hand, higher pre-linearization cost from false registration transfers makes DeNovoSync0 perform slightly (3%) worse than MESI on 16 cores. On 64 cores, however, this effect is mitigated by MESI's higher linearization cost and DeNovoSync0 shows 29% reduction in execution time.

DeNovoSync0 reduces network traffic by 35% on average over MESI on 16 and 64 cores, except for *large CS* where it suffers from high read miss rates as discussed above. The main reasons for reduced traffic on DeNovo are: (1) lack of invalidation/ack traffic (which contributes about 20% of

[3] For MESI, our system does not explicitly identify loads and stores as synchronization or data. We use fences to enforce ordering at the acquire and release synchronization points. We therefore do not separate LD/ST traffic into data and synchronization for MESI. For DeNovo, on the other hand, such information was readily available and we report it here.

network traffic for MESI), and (2) per-word coherence granularity for DeNovo which allows sending only valid data. The latter significantly reduces response traffic for synchronization requests on DeNovo since most software pads lock variables to avoid false sharing.

DeNovoSync0 vs. DeNovoSync: DeNovoSync is comparable or better than DeNovoSync0 for all TATAS-based kernels on 16 and 64 cores (average 5% and 13% lower execution time and 14% and 16% lower network traffic). The benefit comes from delaying unnecessary synchronization read registrations, improving both network traffic and execution time through reduced contention.

Impact of lock padding: We ran all TATAS kernels without lock padding and found that most showed worse performance for MESI than the original versions due to false sharing. However, performance gaps between MESI and DeNovoSync/0 are also reduced by removing the padding because DeNovo has to issue more read/write requests separately for locks and data in the same cache line.

Impact of software backoff: We conducted a sensitivity study to see the impact of software backoff on TATAS-based kernels. We ran the kernels with exponential software backoff in the range of [128,2048) cycles. We found that the performance gap between DeNovoSync0 and MESI increased significantly (up to 70% on 64 cores), relative to kernels without software backoff. Similar to hardware backoff, software backoff also separates failed synchronization read accesses. As a result, it reduces read misses from false races for DeNovoSync/0. On the other hand, its does not affect invalidation latency for MESI, which is the largest source of memory stall time in these kernels.

7.1.2 Array Locks

Figure 4 shows that for array lock based kernels, DeNovoSync0 and DeNovoSync show similar performance and traffic. As analyzed in Section 6, the single-reader design of array locks does not generate spurious registrations and does not benefit from DeNovoSync's backoff.

Compared to MESI, DeNovoSync/0 provides comparable or (up to 24%) better performance except for *heap* and reduces network traffic by 64% on average. As discussed in Section 6, for the most part, synchronization related pre-linearization and linearization costs for MESI and DeNovo are largely similar except for subtle effects; e.g., DeNovo saves one write miss whose latency may or may not be overlapped in MESI, depending on the critical section length. Based on detailed analysis of our experiments, we find that the overall performance of array lock kernels is therefore sensitive to the nature of the critical section computation.

Four of the six kernels (all except *large CS* and *heap*) have very small critical sections; therefore, the additional write miss for MESI adversely impacts performance, giving DeNovo an advantage. *Large CS* has a large critical section, which removes this advantage, and both MESI and DeNovo see comparable performance. *Heap* also has a high number of data accesses, but their unpredictability makes DeNovo suffer from conservative region-based static self-invalidations as follows. *Heap* re-balances its tree as entries are inserted/deleted, requiring a data-dependent traversal of the tree nodes. Without a priori knowledge of which nodes have been previously updated, DeNovo conservatively self-invalidates all nodes, resulting in unnecessary data misses as well as increasing synchronization wait times for subsequent acquirers. DeNovo therefore performs 6% and 7% worse than MESI on 16 and 64 cores respectively. This can be remedied using dynamic hardware signatures to more precisely determine what data to invalidate as in [35], but is orthogonal to the synchronization focus of this work.

Finally, DeNovo's traffic savings are for the same reasons as for the TATAS locks and saved write misses.

7.1.3 Non-Blocking Algorithms

Figure 5 shows that the non-blocking data structures exhibit varying performance depending on their access patterns and system contention. On 16 cores, both DeNovoSync0 and DeNovoSync perform comparably or better than MESI (average of 14% better execution time and 60% better traffic). On 64 cores, DeNovoSync outperforms MESI (by an average of 28% for execution time and 54% for traffic). DeNovoSync0, however, often does worse than MESI on 64 cores; e.g., up to 60% worse execution time.

MESI vs. DeNovoSync0: As discussed in Section 6, DeNovoSync0 suffers from high pre-linearization cost caused by unnecessary registration transfers for many synchronization reads on multiple variables. On 16 cores, this cost is offset by MESI's higher linearization cost (due to invalidations). Thus, all kernels show comparable or better performance and network traffic for DeNovoSync0. With increased number of cores, threads experience more failed equality checks before reaching the linearization point, and DeNovoSync0 is likely to suffer from increased unnecessary read registrations and misses due to interference from remote synchronization reads. For 3 of the 6 non-blocking kernels, DeNovoSync0 therefore performs worse than MESI by 28% on average on 64 cores.

DeNovoSync0 vs. DeNovoSync: Under low contention and with software backoff delay, DeNovoSync's hardware backoff does not provide visible performance boost over DeNovoSync0 (2% better on average for 16 cores). On the other hand, as contention increases on 64 cores, DeNovoSync performs much better than DeNovoSync0 (by 30% and 41% on average for execution time and network traffic respectively). Figure 5 shows that the improved performance directly comes from replacing memory stall time with smaller hardware backoff cycles.

Software Modifications: We observed that many non-blocking algorithms are designed to perform well on systems with a writer-initiated invalidation protocol. Since a core can spin on a cached copy on such systems, most non-blocking algorithms have repeated equality checks comparing a shared variable's current value and a core's local snapshot. These checks help non-blocking algorithms make faster progress on a failed attempt. The same checks do more harm than good for reader-initiated invalidation protocols like DeNovo. If synchronization reads occur much more frequently than writes in a synchronization loop, read registration requests will end up ping-ponging registration between readers, forcing them to miss even if shared variables are not updated at all. Since many equality checks exist only for performance but not for correctness, removing/adding some checks is safe. In our experiments, we modified *Herlihy stack* and *Herlihy heap* from [29] to reduce the number of such checks (these kernels had the most equality checks among those we tested).

We saw that the modifications significantly shortened execution time for both MESI and DeNovo, but DeNovoSync saw much higher improvement than MESI, with 41% and 79% lower execution time on average on 16 and 64 cores compared to the unmodified version. Network traffic also reduced by up to 78% on 64 cores. Studying how implementation details of synchronization algorithms may affect their behaviors on different coherence protocols is an interesting future research direction.

7.1.4 Barriers

Figure 6 shows that DeNovo performs comparably or better than MESI for the barrier kernels studied on 16 and 64 cores except for centralized barrier with highly unbalanced computations (denoted UB) on 64 cores. For network traffic, the DeNovo protocols are better (67% on average) than MESI for the tree barriers, but much worse for the centralized barrier.

As discussed in Section 6, since the tree barriers exhibit a single producer and single consumer scenario for the linearization synchronization variable, all protocols behave similarly for execution time. DeNovoSync/0 sees much lower traffic for reasons similar to other kernels above.

For centralized barriers, MESI has higher linearization cost during arrival than DeNovo while DeNovo has higher linearization cost during departure, as discussed in Section 6. With load balanced computations, these effects offset each other and the execution times for all protocols are comparable. In case of higher load imbalance, however, DeNovo suffers from even more read registrations (and higher pre-linearization cost) than the load-balanced case, delaying the linearization point. As a result, DeNovoSync/0 performs worse than MESI by 9% and 14% on 16 and 64 cores respectively. The excessive registrations in the pre-linearization and linearization phases for DeNovo result in high traffic in both load-balanced and imbalanced cases.

Nevertheless, comparing absolute performance of centralized and tree barriers, we confirmed that tree barriers are better for both our 16 and 64 core systems. Our results thus show that as long as designed in a distributed and scalable fashion, DeNovo can provide reasonable performance even for barrier synchronization (and more generally, conditional synchronization) where the synchronization accesses are mostly reads.

7.2 Results for Applications

Figure 7 shows execution time and network traffic for 13 benchmarks (*ferret* and *x264* on 16 cores, others on 64 cores) on MESI (denoted M) and DeNovoSync (denoted DS), normalized to MESI. The components for each bar are as for the synchronization kernels except that there is no separation of non-synchronization time. We see that DeNovoSync provides comparable execution time to MESI (4% better on average) – it is noticeably better for *LU, water, ocean,* and *ferret* (up to 36% better), but 7% worse for *fluidanimate*. For network traffic, DeNovoSync is 24% better than MESI on average for all benchmarks. We next discuss the applications in more detail, classified by the synchronization patterns they include.

Barrier-only: FFT, LU, blackscholes, swaptions, and *radix* use only barrier synchronization (with tree barriers). For execution time, as expected, DeNovoSync is comparable to MESI. It provides a slight advantage for *LU* since *LU* exhibits data false sharing with MESI (which DeNovo avoids due to word-level coherence [10]). DeNovoSync significantly reduces network traffic because it does not incur invalidations, load responses do not contain invalid parts of the cache line, and stores involve only registration [10].

Barriers and locks: Bodytrack, barnes, water, ocean, and *fluidanimate* use both barriers and locks. Except for *fluidanimate*, DeNovoSync shows comparable or (up to 30%) better execution time than MESI for these applications. For *fluidanimate*, DeNovoSync is 7% worse because of the use of static self-invalidations for shared data protected by critical sections. Self-invalidating all data in writeable regions at every lock acquire results in unnecessary invalidation of valid data for this application. As discussed in the previous section for array-based lock heap in Section 7.1.2, we can reduce unnecessary read misses with more dynamic solutions. DeNovoSync shows substantial savings in network traffic for these applications as expected.

Non-blocking synchronization: canneal is a unique benchmark that synchronizes shared pointers in an aggressive lock-free loop with CAS instructions. Unlike other applications, synchronization forms a large fraction of the memory accesses in *canneal*. Again, DeNovoSync shows comparable execution time and reduced network traffic relative to MESI.

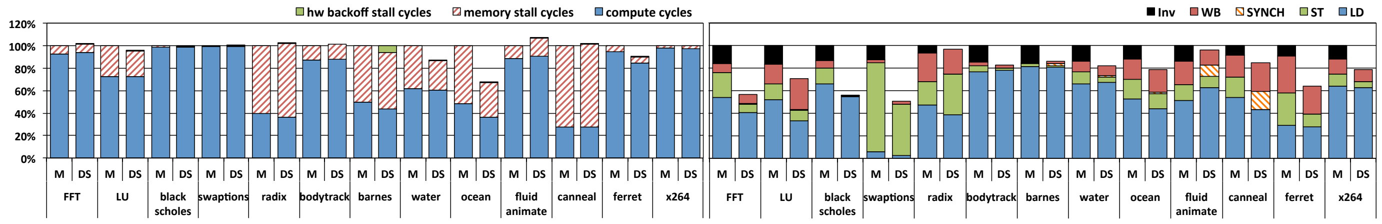
Pipeline parallelism: ferret and *x264* use pipeline parallelism, a parallel pattern that has not been previously evaluated for DeNovo. Again, relative to MESI, DeNovoSync shows comparable or better (by 5% for *ferret*) execution time, with an average network traffic reduction of 29%.

Overall, DeNovoSync efficiently supports a variety of applications, without restrictions on synchronization patterns.

8. Related Work

There has been a vast body of work on software-driven coherence protocols and various shared-memory optimizations that inspired the DeNovo system [16, 19, 20, 22, 24, 30, 31, 38]. This section focuses mainly on closely related work in efficiently supporting synchronization mechanisms.

Research in software distributed shared-memory (DSM) proposed several software techniques to provide an illusion of a consistent global address space for DSMs [5, 16, 18]. The work focused mainly on consistency for data accesses, and assumed explicit and customized software interfaces for a limited set of synchronization patterns. DeNovoSync



(a) Execution Time. (b) Network Traffic.

Figure 7: Execution time and network traffic for MESI and DeNovoSync for applications.

aims to provide integrated hardware coherence solutions for synchronization and data accesses on tightly-coupled multicores, and supports arbitrary synchronization.

VIPS-M [32], a recent version of SARC [17], proposed a protocol somewhat similar to DeNovo. It uses private/shared information about data to perform self-invalidations and eliminates the directory by performing write-through at synchronization points. The work also implemented a write-through protocol for synchronization accesses, but claimed that delaying the completion of such a write-through helps reducing spinning by other cores, which is similar to delaying of QOLB attempts in [31]. However, VIPS-M is not clear about how it efficiently deals with synchronization algorithms with multiple synchronization variables and frequent reads to them such as with non-blocking algorithms considered here. If many variables cause delayed write-through, they may interfere with each others' progress causing degraded synchronization latency. DeNovoSync, on the other hand, supports arbitrary synchronization, including non-blocking algorithms, through a novel combination of synchronization read registration and hardware backoff.

Dir$_1$SW [15] proposes a simplified directory protocol that adds little complexity to message-passing hardware by using software traps, but efficiently supports programs written within the CICO model [23]. Dir$_1$SW supports synchronization accesses separately from data accesses with non-cacheable pages, since they do not fit with the CICO model, ruining performance. DeNovoSync allows both data and synchronization accesses to be cached locally and maintained by coherence protocols.

More recently, TSO-CC [11] proposed a self-invalidation based coherence protocol for the TSO memory model. It leverages relaxed write-to-read ordering in TSO to introduce lazy self-invalidations for reads and eliminate sharers and invalidation traffic for writes. However, without distinction between data and synchronization accesses, every read miss has to self-invalidate all shared cache lines in private caches to guarantee read-to-read ordering. To avoid cache-wide flushes, TSO-CC introduces many additional hardware mechanisms such as timestamps and epoch tables and complicated logic to maintain them.

Read-for-Ownership has been explored as an optimization for reducing write misses on synchronization accesses. [31] explores how Queue-On-a-Lock-Bit (QOLB), a hardware distributed lock mechanism, can be applied to individual synchronization accesses. While they propose a range of solutions with different trade-off, Read-for-Ownership was dismissed as it is expected to generate spurious read misses. We show that for many synchronization algorithms, a judicious use of RFO can be a simple yet effective solution.

It is worthwhile to note that many of the above and other related work assume that data and non-data (synchronization) accesses are distinguished and provide different solutions for these two classes. To our knowledge, no previous writer-invalidation-free system has supported arbitrary synchronization with cacheable synchronization variables. We expect that the ideas underlying DeNovoSync for synchronization can also be adapted to other writer-invalidation-free systems to improve the efficiency and generality of their supported synchronization mechanisms.

9. Conclusion

The DeNovo system has shown that hardware coherence for data accesses can be made much simpler and more efficient by leveraging software information and properties like data-race-freedom. However, adding support for races on a protocol driven by race-free software could potentially undermine all the benefits of the existing systems. This paper showed that leveraging access patterns of synchronization operations can help design a simple yet flexible mechanism that can support the coherence and consistency requirements of synchronization accesses. While the simple "single-reader" approach proposed for DeNovoSync0 may suffer from conservative invalidations and traffic under high contention, delaying these reads with a contention-dependent backoff as in DeNovoSync can improve performance significantly. We compared our new protocols with the state-of-the-art MESI on several synchronization constructs including those for challenging non-blocking data structures. We found our technique to be quite competitive and in many cases much better. For future work, we would like to integrate more dynamic signature-based coherence support for data accesses with DeNovoSync, reaching towards a comprehensive coherence solution. Studying protocol-aware synchronization design further and exploring the potential for applying DeNovoSync to other writer-invalidation free systems are also promising future work directions.

References

[1] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[2] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, 1989.

[3] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha. Garnet: A detailed interconnection network model inside a full-system simulation framework. Technical Report CE-P08-001, Princeton University, 2008. URL `http://www.princeton.edu/~niketa/garnet`.

[4] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1), Jan. 1990.

[5] B. Bershad, M. Zekauskas, and W. Sawdon. The midway distributed shared memory system. In *Compcon Spring '93, Digest of Papers.*, Feb 1993.

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

[7] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect system for deterministic parallel java. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, 2009.

[8] 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 *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, 2011.

[9] H.-J. Boehm and S. V. Adve. Foundations of the c++ concurrency memory model. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, 2008.

[10] 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 *Proceedings of the 20th International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, 2011.

[11] M. Elver and V. Nagarajan. Tso-cc: Consistency directed cache coherence for tso. In *IEEE 20th International Symposium on High Performance Computer Architecture*, HPCA-20, Feb 2014.

[12] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, ISCA '88, 1988.

[13] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, 1989.

[14] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proceedings of the Second ACM SIGPLAN Symposium on Principles &Amp; Practice of Parallel Programming*, PPOPP '90, 1990.

[15] M. D. Hill, J. R. Larus, S. K. Reinhardt, and D. A. Wood. Cooperative shared memory: Software and hardware for scalable multiprocessor. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS V, 1992.

[16] L. Iftode, J. P. Singh, and K. Li. Scope consistency: A bridge between release consistency and entry consistency. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '96, 1996.

[17] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5), Sept.-Oct. 2010.

[18] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, ISCA '92, 1992.

[19] 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 *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[20] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: A Hybrid Memory Model for Accelerators. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, 2010.

[21] R. Komuravelli, S. V. Adve, and C.-T. Chou. Revisiting the complexity of hardware cache coherence and some implications. *ACM Trans. Archit. Code Optim.*, Dec. 2014.

[22] D. Koufaty, X. Chen, D. Poulsen, and J. Torrellas. Data forwarding in scalable shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(12), dec 1996.

[23] J. R. Larus, S. Chandra, and D. A. Wood. Cico: A practical shared-memory programming performance model. In *Workshop on Portability and Performance for Parallel Processing*, 1993.

[24] A. R. Lebeck and D. A. Wood. Dynamic self-invalidation: Reducing coherence overhead in shared-memory multiprocessors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, 1995.

[25] P. S. 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.

[26] J. Manson, W. Pugh, and S. V. Adve. The java memory model. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, 2005.

[27] 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*, 33(4):92–99, 2005.

[28] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, 1996.

[29] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1), May 1998.

[30] S. L. Min and J.-L. Baer. Design and analysis of a scalable cache coherence scheme based on clocks and timestamps. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):25–44, January 1992.

[31] R. Rajwar, A. Kagi, and J. Goodman. Improving the throughput of synchronization by insertion of delays. In *Proceedings of the Sixth International Symposium on High-Performance Computer Architecture*, HPCA-6, 2000.

[32] A. Ros and S. Kaxiras. Complexity-effective multicore coherence. In *Proceedings of the 21st international conference on Parallel architectures and compilation techniques*, PACT '12, 2012.

[33] M. Scott. *Shared Memory Synchronization*. Synthesis Lectures on Computer Architecture. Morgan & Claypool, 2013. ISBN 9781608459568. URL `http://books.google.com/books?id=N4YcnQEACAAJ`.

[34] S. Subramaniam, S. C. Steely, W. Hasenplaugh, A. Jaleel, C. Beckmann, T. Fossum, and J. Emer. Using in-flight chains to build a scalable cache coherence protocol. *ACM Trans. Archit. Code Optim.*, 10(4), Dec. 2013.

[35] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: efficient hardware support for disciplined non-determinism. In *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '13, 2013.

[36] H. Sung, R. Komuravelli, and S. V. Adve. DeNovoND: efficient hardware for disciplined nondeterminism. *IEEE Micro*, 34(3), 2014.

[37] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ISCA '95, 1995.

[38] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A tagless coherence directory. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, 2009.