2011 International Conference on Parallel Architectures and Compilation Techniques

DeNovo: Rethinking the Memory Hierarchy for Disciplined Parallelism*

Byn Choi,† Rakesh Komuravelli,† Hyojin Sung,† Robert Smolinski,† Nima Honarmand,†
Sarita V. Adve,† Vikram S. Adve,† Nicholas P. Carter,‡ and Ching-Tsun Chou‡

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

‡Intel
Santa Clara, CA
{nicholas.p.carter, ching-tsun.chou}@intel.com

***Abstract*—For parallelism to become tractable for mass programmers, shared-memory languages and environments must evolve to enforce disciplined practices that ban "wild shared-memory behaviors;" e.g., unstructured parallelism, arbitrary data races, and ubiquitous non-determinism. This software evolution is a rare opportunity for hardware designers to rethink hardware from the ground up to exploit opportunities exposed by such disciplined software models. Such a co-designed effort is more likely to achieve many-core scalability than a software-oblivious hardware evolution.**

This paper presents DeNovo, a hardware architecture motivated by these observations. We show how a disciplined parallel programming model greatly simplifies cache coherence and consistency, while enabling a more efficient communication and cache architecture. The DeNovo coherence protocol is simple because it eliminates transient states – verification using model checking shows 15X fewer reachable states than a state-of-the-art implementation of the conventional MESI protocol. The DeNovo protocol is also more extensible. Adding two sophisticated optimizations, flexible communication granularity and direct cache-to-cache transfers, did not introduce additional protocol states (unlike MESI). Finally, DeNovo shows better cache hit rates and network traffic, translating to better performance and energy. Overall, a disciplined shared-memory programming model allows DeNovo to seamlessly integrate message passing-like interactions within a global address space for improved design complexity, performance, and efficiency.

I. Introduction

Achieving the promise of Moore's law will require harnessing increasing amounts of parallelism using multicores, with predictions of hundreds of cores per chip. Shared-memory is arguably the most widely used general-purpose multicore parallel programming model. While shared-memory provides the advantage of a global address space, it is known to be difficult to program, debug, and maintain [52]. Specifically, unstructured parallel control, data races, and ubiquitous non-determinism make programs difficult to understand, and sacrifice safety, modularity, and composability. At the same time, designing performance-, power-, and complexity-scalable hardware for such a software model remains a major challenge; e.g., directory-based cache coherence protocols are notoriously complex [3] and hard to scale and an active area of research [72, 37, 54, 63, 57]. More fundamentally, a satisfactory definition of memory consistency semantics (i.e., specification of what value a shared-memory read should return) for such a model has proven elusive, and a recent paper makes the case for rethinking programming languages and hardware to enable usable memory consistency semantics [4].

*This work is supported in part by Intel and Microsoft through the Universal Parallel Computing Research Center (UPCRC) at Illinois and by the National Science Foundation under grant number CCF-1018796. We thank Craig Zilles for discussions in the initial phase of this project and for the project name.

The above problems have led some researchers to promote abandoning shared-memory altogether (e.g., [52]). Some projects do away with coherent caches, most notably the 48 core Intel Single-Chip Cloud Computer [43], pushing significant complexity into the programming model. An alternative view is that the above problems are not inherent to a global address space paradigm, but instead occur due to undisciplined programming models that allow arbitrary reads and writes for implicit and unstructured communication and synchronization. This results in "wild shared-memory" behaviors with unintended data races and non-determinism and implicit side effects. The same phenomena result in complex hardware that must assume that any memory access may trigger communication, and performance- and power-inefficient hardware that is unable to exploit communication patterns known to the programmer but obfuscated by the programming model.

There is much recent software work on more *disciplined* shared-memory programming models to address the above problems (Section I-B). This paper concerns the first step of a hardware project, DeNovo, that asks the question: *if software becomes more disciplined, can we build more performance-, power-, and complexity-scalable hardware?* Specifically, this paper focuses on the impact of disciplined software on the cache coherence protocol.

A. Hardware Coherence Scaling Issues

Shared-memory systems typically implement coherence with snooping or directory-based protocols. Although current directory-based protocols are more scalable than snooping protocols, they suffer from several limitations:

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

Verification complexity and extensibility: They are notoriously complex and difficult to verify since they require dealing with subtle races and many transient states [60, 34]. Furthermore, their fragility often discourages implementors from adding optimizations to previously verified protocols – additions usually require re-verification due to even more states and races.

State overhead: Directory protocols incur high directory storage overhead to track sharer lists. Several optimized directory organizations have been proposed, but also require considerable overhead and/or excessive network traffic and/or complexity. These protocols also require several coherence state bits due to the large number of protocol states (e.g., ten

1089-795X/11 $26.00 © 2011 IEEE
DOI 10.1109/PACT.2011.21

bits in [63]). This state overhead is amortized by tracking coherence at the granularity of cache lines. This can result in performance/power anomalies and inefficiencies when the granularity of sharing is different from a contiguous cache line (e.g., false sharing).

Researchers continue to propose new hardware directory organizations and protocol optimizations to address one or more of the above limitations [72, 37, 54, 50, 1, 62, 67]; however, all of these approaches incur one or more of complexity, performance, power, or storage overhead. Recently, Kaxiras and Keramidas [45] exploited the data-race-free property of current memory consistency models to address the performance and power costs of the directory-based MESI protocol. DeNovo is a hardware-software co-designed approach that exploits emerging disciplined software properties in addition to data-race-freedom to target all the above mentioned limitations of directory protocols for large core counts.

B. Software Scope

There has been much recent research on disciplined shared-memory programming models with explicit and structured communication and synchronization for both deterministic and non-deterministic algorithms [7]; e.g., Ct [33], CnC [22], Cilk++ [18], Galois [49], SharC [10], Kendo [61], Prometheus [9], Grace [14], Axum [35], and Deterministic Parallel Java (DPJ) [21, 20].

Although the targeted disciplined programming models are still under active research, many of them guarantee determinism. We focus this paper on deterministic codes for three reasons. (1) There is a growing view that deterministic algorithms will be common, at least for client-side computing [7]. (2) Focusing on these codes allows us to investigate the "best case;" i.e., the potential gain from exploiting strong discipline. (3) These investigations will form a basis on which we develop the extensions needed for other classes of codes in the future; in particular, disciplined non-determinism, legacy software, and programming models using "wild shared-memory." Synchronization mechanisms involve races and are used in all classes of codes; here, we assume special techniques to implement them (e.g., hardware barriers, queue based locks, etc.) and postpone their detailed handling to future work. We also postpone OS redesign work, and hope to leverage recent work on "disciplined" OS; e.g., [13, 42] and [25].

We use Deterministic Parallel Java (DPJ) [21] as an exemplar of the emerging class of deterministic-by-default languages (Section II), and use it to explore how hardware can take advantage of strong disciplined programming features. Specifically, we use three features of DPJ that are also common to several other projects: (1) structured parallel control; (2) data-race-freedom, and guaranteed deterministic semantics unless the programmer explicitly requests non-determinism (called determinism-by-default); and (3) explicit specification of the side effects of parallel sections; e.g., which (possibly non-contiguous) regions of shared-memory will be read or written in a parallel section.

Most of the disciplined models projects cited above also enforce a requirement of structured parallel control (e.g., a nested fork join model, pipelining, etc.), which is much easier to reason about than arbitrary (unstructured) thread synchronization. Most of these, including all but one of the commercial systems, guarantee the absence of data races for programs that type-check. Coupled with structured parallel control, the data-race-freedom property guarantees determinism for several of these systems. We also note that data races are prohibited (although not checked) by existing popular languages as well; the emerging C++ and C memory models do not provide *any* semantics with any data race (benign or otherwise) and Java provides extremely complex and weak semantics for data races only for the purposes of ensuring safety. The information about side effects of concurrent tasks is also available in other disciplined languages, but in widely varying (and sometimes indirect) ways. Once we understand the types of information that is most valuable, our future work includes exploring how the information can be extracted from programs in other languages.

C. Contributions

DeNovo starts from language-level annotations designed for concurrency safety, and shows that they can be efficiently represented and used in hardware for better complexity and scalability. Two key insights underlie our design. First, structured parallel control and knowing which memory regions will be read or written enable a cache to take responsibility for invalidating its own stale data. Such self-invalidations remove the need for a hardware directory to track sharer lists and to send invalidations and acknowledgements on writes. Second, data-race-freedom eliminates concurrent conflicting accesses and corresponding transient states in coherence protocols, eliminating a major source of complexity. Our specific results are as follows, applied to a range of array and complex pointer-based applications.

Simplicity: To provide quantitative evidence of the simplicity of the DeNovo protocol, we compared it with a conventional MESI protocol by implementing both in the Murphi model checking tool [29]. For MESI, we used the implementation in the Wisconsin GEMS simulation suite [56] as an example of a (publicly available) state-of-the-art, mature implementation. We found several bugs in MESI that involved subtle data races and took several days to debug and fix. The debugged MESI showed 15X more reachable states compared to DeNovo, with a verification time difference of 173 seconds vs 8.66 seconds.

Extensibility: To demonstrate the extensibility of the DeNovo protocol, we implemented two optimizations: (1) Direct cache-to-cache transfer: Data in a remote cache may directly be sent to another cache without indirection to the shared lower level cache (or directory). (2) Flexible communication granularity: Instead of always sending a fixed cache line in response to a demand read, we send a programmer directed set of data associated with the region information of the demand read. Neither optimization required adding any new protocol states to DeNovo; since there are no sharer lists, valid data can be freely transferred from one cache to another.

Storage overhead: Our protocol incurs no storage overhead for directory information. On the other hand, we need to maintain information about regions and coherence state bits at the granularity at which we guarantee data-race freedom, which can be less than a cache line. For low core counts, this overhead is higher than with conventional directory schemes, but it pays off after a few tens of cores and is scalable (constant per cache line). A positive side effect is that it is easy to eliminate the requirement of inclusivity in a shared last level cache (since we no longer track sharer lists). Thus,

DeNovo allows more effective use of shared cache space.

Performance and power: In our evaluations, the base DeNovo protocol showed about the same or better memory behavior than the MESI protocol. With the optimizations described, DeNovo saw a reduction in memory stall time of up to 81% compared to MESI. In most cases, these stall time reductions came from commensurate reductions in miss rate and were accompanied with significant reductions in network traffic, thereby benefiting not only execution time but also power.

II. Background: Deterministic Parallel Java

DPJ is an extension to Java that enforces *deterministic-by-default* semantics via compile-time type checking [21, 20]. DPJ provides a new type and effect system for expressing important patterns of deterministic and non-deterministic parallelism in imperative, object-oriented programs. Non-deterministic behavior can only be obtained via certain explicit constructs. For a program that does not use such constructs, DPJ guarantees that if the program is well-typed, any two parallel tasks are *non-interfering*, i.e., do not have conflicting accesses.

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

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

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

DPJ has been evaluated on a wide range of deterministic parallel programs. The results show that DPJ can express a wide range of realistic parallel algorithms, and that well-tuned DPJ programs exhibit good performance [21].

III. DeNovo Coherence and Consistency

A shared-memory design must first and foremost ensure that a read returns the correct value, where the definition of "correct" comes from the memory consistency model. Modern systems divide this responsibility between two parts: (i) cache coherence, and (ii) various memory ordering constraints. These are arguably among the most complex and hard to scale aspects of shared-memory hierarchy design. Disciplined models enable mechanisms that are potentially simpler and more efficient to achieve this function.

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

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

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

A. DeNovo with Equal Address/Communication and Coherence Granularity

DeNovo eliminates the drawbacks of conventional directory protocols as follows.

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

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

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

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

No transient states: The DeNovo protocol has three states in the L1 and L2 – `registered`, `valid`, and `invalid` – with obvious meaning. (The touched bit mentioned above is local to its cache and irrelevant to external coherence transactions.) Although textbook descriptions of conventional directory protocols also describe 3 to 5 states (e.g., MSI) [40], it is well-known that they contain many hidden transient states due to races, making them notoriously complex and difficult to verify [3, 65, 70]. For example, considering a simple MSI protocol, a cache may request ownership, the directory may forward the request to the current owner, and another cache may request ownership while all of these messages are still outstanding. Proper handling of such a race requires introduction of transient states into the cache and/or directory transition tables.

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

The full protocol: Table I shows the L1 and L2 state transitions and events for the full protocol. Note the lack of transient states in the caches.

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

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

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

The L2 transitions are also straightforward except for writebacks which require some care. A read or registration request to data that is invalid or valid at the L2 invokes the obvious response. For a request for data that is registered by an L1, the L2 forwards the request to that L1 and updates its registration id if needed. For a forwarded registration request, the L1 always acknowledges the requestor and invalidates its own copy. If the copy is already invalid due to a concurrent writeback by the L1, the L1 simply acknowledges the original requestor and the L2 ensures that the writeback is not accepted (by noting that it is not from the current registrant). For a forwarded read request, the L1 supplies the data if it has it. If it no longer has the data (because it issued a concurrent writeback), then it sends a negative acknowledgement (nack) to the original requestor, which simply resends the request to the L2. Because of race-freedom, there cannot be another concurrent write, and so no other concurrent writeback, to the line. Thus, the nack eventually finds the line in the L2, without danger of any deadlock or livelock. The only somewhat less straightforward interaction is when both the L1 and L2 caches want to writeback the same line concurrently, but this race also occurs in uniprocessors.

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

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

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

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

(b) **L2 cache**

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

objects as one region.

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

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

During final code generation, the memory instructions generated can convey the region name of the address being accessed to the hardware; since DPJ regions are parameterizable, the instruction needs to point to a hardware register that is set at runtime (through the compiler) with the actual region number. When the memory instruction is executed, it conveys the region number to the core's cache. A straightforward approach is to store the region number with the accessed data line in the cache. Now a self-invalidate instruction invalidates all data in the cache with the specified regions that is not `touched` or `registered`.

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

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

registered to the appropriate cores. (The direct communication is explained in the next section.)

B. DeNovo with Address/Communication Granularity > Coherence Granularity

To decouple the address/communication and coherence granularity, our key insight is that any data marked `touched` or `registered` can be copied over to any other cache in `valid` state (but not as `touched`). Additionally, for even further optimization (Section III-D1), we make the observation that this transfer can happen without going through the registry/L2 at all (because the registry does not track sharers). Thus, no serialization at a directory is required. When (if) this copy of data is accessed through a demand read, it can be immediately marked `touched`. The above copy does not incur false sharing (nobody loses ownership) and, if the source is the non-home node, it does not require extra hops to a directory.

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

Note that if the L2 has a line `valid` in the cache, then an element of that line can be either `valid` (and hence sent to the requestor) or `registered` (and hence not sent). Thus, for the L2, it suffices to keep just one coherence state bit at the finer (e.g., word) granularity with a line-wide valid bit at the line granularity.[1] As before, the id of the registered core is stored in the data array of the registered location.

This is analogous to sector caches – cache space allocation (i.e., address tags) is at the granularity of a line but there may be some data within the line that is not valid. This combination effectively allows exploiting spatial locality without any false sharing, similar to multiple writer protocols of software distributed shared memory systems [46].

C. Flexible Coherence Granularity

Although the applications we studied did not have any data races at word granularity, this is not necessarily true of all applications. Data may be shared at byte granularity, and two cores may incur conflicting concurrent accesses to the same word, but for different bytes. A straightforward implementation would require coherence state at the granularity of a byte, which would be significant storage overhead. [2] Although previous work has suggested using byte based granularity for state bits in other contexts [53], we would like to minimize the overhead.

[1]This requires that if a registration request misses in the L2, then the L2 obtain the full line from main memory.

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

We focus on the overhead in the L2 cache since it is typically much larger (e.g., 4X to 8X times larger) than the L1. We observe that byte granularity coherence state is needed only if two cores incur conflicting accesses to different bytes in the same word in the same phase. Our approach is to make this an infrequent case, and then handle the case correctly albeit at potentially lower performance.

In disciplined languages, the compiler/runtime can use the region information to allocate tasks to cores so that byte granularity regions are allocated to tasks at word granularities when possible. For cases where the compiler (or programmer) cannot avoid byte granularity data races, we require the compiler to indicate such regions to the hardware. Hardware uses word granularity coherence state. For byte-shared data such as the above, it "clones" the cache line containing it in four places: place i contains the ith byte of each word in the original cache line. If we have at least four way associativity in the L2 cache (usually the case), then we can do the cloning in the same cache set. The tag values for all the clones will be the same but each clone will have a different byte from each word, and each byte will have its own coherence state bit to use (essentially the state bit of the corresponding word in that clone). This allows hardware to pay for coherence state at word granularity while still accommodating byte granularity coherence when needed, albeit with potentially poorer cache utilization in those cases.

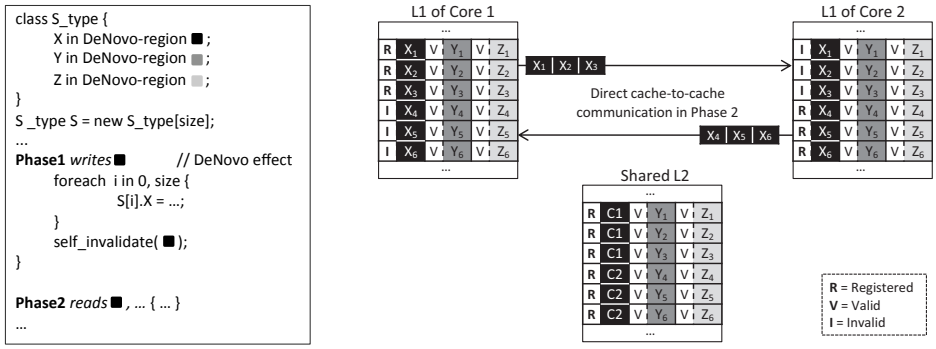
D. Protocol Optimizations

1) Eliminating indirection: Our protocol so far suffers from the fact that even L1 misses that are eventually serviced by another L1 cache (cache-to-cache transfer) must go through the registry/L2 (directory in conventional protocols), incurring an additional latency due to the indirection.

However, as observed in Section III-B, `touched/registered` data can always be transferred for reading without going through the registry/L2. optimization). Thus, a reader can send read requests directly to another cache that is predicted to have the data. If the prediction is wrong, a Nack is sent (as usual) and the request reissued as a usual request to the directory. Such a request could be a demand load or it could be a prefetch. Conversely, it could also be a producer-initiated communication or remote write [2, 48]. The prediction could be made in several ways; e.g., through the compiler or through the hardware by keeping track of who serviced the last set of reads to the same region. The key point is that there is no impact on the coherence protocol – no new states, races, or message types. The requestor simply sends the request to a different supplier. This is in sharp contrast to adding such an enhancement to MESI.

This ability essentially allows DeNovo to seamlessly integrate a message passing like interaction within its shared-memory model. Figure 1 shows such an interaction for our example code.

2) Flexible communication granularity: Cache-line based communication transfers data from a set of contiguous addresses, which is ideal for programs with perfect spatial locality and no false sharing. However, it is common for programs to access only a few data elements from each line, resulting in significant waste. This is particularly common in modern object-oriented programming styles where data



(a) (b)

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

structures are often in the form of arrays of structs (AoS) rather than structs of arrays (SoA). It is well-known that converting from AoS to SoA form often gives a significant performance boost due to better spatial locality. Unfortunately, manual conversion is tedious, error-prone, and results in code that is much harder to understand and maintain, while automatic (compiler) conversion is impractical except in limited cases because it requires complex whole-program analysis and transformations [28, 44]. We exploit information about regions to reduce such communication waste, without changing the software's view.

We have knowledge of which regions will be accessed in the current phase. Thus, when servicing a remote read request, a cache could send `touched` or `registered` data only from such regions (recall these are at field granularity within structures), potentially reducing network bandwidth and power. More generally, the compiler may associate a default prefetch granularity attribute with each region that defines the size of each contiguous region element, other regions in the object likely to be accessed along with this region (along with their offset and size), and the number of such elements to transfer at a time. This information can be kept as a table in hardware which is accessed through the region identifier and an entry provides the above information; we call the table the *communication region table*. The information for the table itself may be partly obtained directly through the programmer, deduced by the compiler, or deduced by a runtime tool. Figure 1 shows an example of the use of flexible communication granularity – the caches communicate multiple (non-contiguous) fields of region X rather than the contiguous X, Y, and Z regions that would fall in a conventional cache line. Again, in contrast to MESI, the additional support required for this enhancement in DeNovo does not entail any changes to the coherence protocol states or introduce new protocol races.

This flexible communication granularity coupled with the ability to remove indirection through the registry/L2 (directory) effectively brings the system closer to the efficiency of message passing while still retaining the advantages of a coherent global address space. It combines the benefits of various previously proposed shared-memory techniques such as bulk data transfer, prefetching, and producer-initiated communication, but in a more software-aware fashion that potentially results in a simpler and more effective system.

E. Storage Overhead

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

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

In-cache full map directory. We conservatively assume 5 bits for protocol state (assuming more than 16 stable+transient states). This gives 5 bits per 64 byte cache line at the L1. With full map directories, each L2 line needs a bit per core for the sharer list. This implies that DeNovo overhead for just the L2 is better for more than a 13 core system. If the L2 cache size is 8X that of L1, then the total L1+L2 overhead of DeNovo is better at greater than about 21 (with `keepValid`) to 30 cores.

Duplicate tag directories. L1 tags can be duplicated at the L2 to reduce directory overhead. However, this requires a very high associative lookup; e.g., 64 cores with 4 way L1 requires a 256 way associative lookup. As discussed in [72], this design is not scalable to even low tens of cores system.

Tagless directories and sparse directories. The tagless directories work uses Bloom filter based directory organization [72]. Their directory storage requirement appears to be about 3% to over 5% of L1 storage for core counts ranging from 64 to 1K cores. This does not include any coherence state overhead which we include in our calculation for DeNovo above. Further, this organization is lossy in that larger core counts require extra invalidations and protocol complexity.

Many sparse directory organizations have been proposed that can drastically cut directory overhead at the cost of sharer list precision, and so come at a significant performance cost especially at higher core counts [72].

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

TABLE II: Parameters of the simulated processor.

IV. Methodology

A. Simulation Environment

Our simulation environment consists of the Simics full-system functional simulator that drives the Wisconsin GEMS memory timing simulator [56] which implements the simulated protocols. We also use the Princeton Garnet [8] interconnection network simulator to accurately model network traffic. We chose not to employ a detailed core timing model due to an already excessive simulation time. Instead, we assume a simple, single-issue, in-order core with blocking loads and 1 CPI for all non-memory instructions. We also assume 1 CPI for all instructions executed in the OS and in synchronization constructs.

Table II summarizes the key common parameters of our simulated systems. Each core has a 128KB private L1 Dcache (we do not model an Icache). L2 cache is shared and banked (512KB per core). The latencies in Table II are chosen to be similar to those of Nehalem [36], and then adjusted to take some properties of the simulated processor (in-order core, two-level cache) into account.

B. Simulated Protocols

We compared the following 8 systems:

MESI word (MW) and line (ML): MESI with single-word (4 byte) and 64-byte cache lines, respectively. The original implementation of MESI shipped with GEMS [56] does not support non-blocking stores. Since stores are non-blocking in DeNovo, we modified the MESI implementation to support non-blocking stores for a fair comparison. Our tests show that MESI with non-blocking stores outperforms the original MESI by 28% to 50% (for different applications).

DeNovo word (DW) and line (DL): DeNovo with single-word (Section III) and 64-byte cache lines, respectively.

For DL, we do not charge any additional cycles for gathering/scattering valid-only packets. We charge network bandwidth for only the valid part of the cache line plus the valid-word bit vector.

DL with direct cache-to-cache transfer (DD): Line-based DeNovo with direct cache-to-cache transfer (Section III-D1). We use oracular knowledge to determine the cache that has the data. This provides an upper-bound on achievable performance improvement.

DL with flexible communication granularity (DF): Line-based DeNovo with flexible communication granularity (Section III-D2). Here, on a demand load, the communication region table is indexed by the region of the demand load to obtain the set of addresses that are associated with that load, referred to as the *communication space*. We fix the maximum data communicated to be 64 bytes for DF. If the communication space is smaller than 64 bytes, then we choose the rest of the words from the 64-byte cache line containing the demand load address. We optimistically do not charge any additional cycles for determining the communication space and gathering/scattering that data.

DL and DW with both direct cache-to-cache transfer and flexible communication granularity (DDF and DDFW respectively): Line-based and word-based DeNovo with the above two optimizations, direct cache-to-cache transfer and flexible communication granularity, combined in the obvious way.

We do not show word based DeNovo augmented with just direct cache-to-cache transfer or just flexible communication granularity because of lack of space, the results were as expected and did not lend new insights, and the DeNovo word based implementations have too much tag overhead compared to the line based implementations.

C. Conveying Regions and Communication Space

Regions for self-invalidation: In a real system, the compiler would convey the region of a data through memory instructions (Section III). For this study, we created an API to manually instrument the program to convey this information for every allocated object. This information is maintained in a table in the simulator. At every load or store, the table is queried to find the region for that address (which is then stored with the data in the L1 cache).

Self invalidation: This API call invalidates all the data in the cache associated with the given region, if the data is not `touched` or `registered`. For the applications studied in this paper (see below), the total number of regions ranged from 2 to about 20. These could be coalesced by the compiler, but we did not explore that here.

Communication space: To convey communication granularity information, we again use a special API call that controls the communication region table of the simulator. On a demand load, the table is accessed to determine the communication space of the requested word. In an AoS program, this set can be simply defined by specifying 1) what object fields, and 2) how many objects to include in the set. For six of our benchmarks, these API calls are manually inserted. The seventh, kdTree, is more complex, so we use an automated correlation analysis tool to determine the communication spaces. We omit the details for lack of space.

D. Protocol Verification

We used the widely used Murphi model checking tool [29] to formally compare the verification complexity of DeNovo and MESI. We model checked the word-based protocol of DeNovo and MESI. We derived the MESI model from the GEMS implementation (the SLICC files) and the DeNovo model directly from our implementation. To keep the number of explored states tractable, as is common practice, we used a single address / single region (only for DeNovo), two data values, two cores with private L1 cache and a unified L2 with in-cache directory (for MESI). We modeled an unordered full network with separate request and reply links. Both models allow only one request per L1 in the rest of the memory hierarchy. For DeNovo, we modeled the data-race-free guarantee by limiting conflicting accesses. We also introduced the notion of phase boundary to provide a realistic model to both protocols by modeling it as a sense reversing barrier. This enables cross

phase interactions in both protocols. As we modeled only one address to reduce the number of states explored, we modeled replacements as unconditional events that can be triggered at any time.

E. Workloads

We use seven benchmarks to evaluate the effectiveness of DeNovo features for a range of dense-array, array-of-struct, and irregular pointer-based applications. FFT (with input size m=16), LU (with 512x512 array and 16-byte blocks), Radix (with 4M integers and 1024 radix), and Barnes-Hut (16K particles) are from the SPLASH-2 benchmark suite [69]. kdTree [27] is a program for construction of k-D trees which are well studied acceleration data structures for ray tracing in the increasingly important area of graphics and visualization. We run it with the well known bunny input. We use two versions of kdTree: kdTree-false which has false sharing in an auxiliary data structure and kdTree-padded which uses padding to eliminate this false sharing. We use these two versions to analyze the effect of application-level false sharing on the DeNovo protocols. We also use fluidanimate (with simmedium input) and bodytrack (with simsmall input) from the PARSEC benchmark suite [16]. To fit into the fork-join programming model, fluidanimate was modified to use the ghost cell pattern instead of mutexes, and radix was modified to perform a parallel prefix with barriers instead of condition variables. For bodytrack, we use its pthread version unmodified.

V. Results

We focus our discussion on the time spent on memory stalls and on network traffic since DeNovo targets these components. Figures 2a, 2b, and 2c respectively show the memory stall time, read miss counts, and network traffic for all eight protocols described in Section IV-B for each application. Each bar (protocol) is normalized to the corresponding (state-of-the-art) MESI-line (ML) bar.

The memory stall time bars (Figure 2a) are divided into four components. The bottommost indicates time spent by a memory instruction stalled due to a blocked L1 cache related resource (e.g., the 64 entry buffer for non-blocking stores is full). The upper three indicate additional time spent stalled on an L1 miss that gets resolved at the L2, a remote L1 cache, or main memory respectively. The miss count bars (Figure 2b) are divided analogously. The network traffic bars (Figure 2c) show the number of flit crossings through on-chip network routers due to reads, writes, writebacks, and invalidations respectively.

For reference, Figure 2d shows the overall execution time for all the protocols and applications, divided into time spent in compute cycles, memory stalls, and synchronization stalls respectively.

LU and bodytrack show considerably large synchronization times. LU has inherent load imbalance. Using larger input sizes would reduce synchronization time, but prohibitively long simulation times made that impractical for this paper. Bodytrack has several sequential phases and a limited amount of parallelism for the input used (only up to 60 threads in some phases [17]). The idle cores in these phases result in the high synchronization time.

MESI vs. DeNovo word protocols (MW vs. DW): MW and DW are not practical protocols because of their excessive tag overhead. A comparison is instructive, however, to understand the efficacy of selective self-invalidation, independent of line-based effects such as false sharing. In all cases, DW's performance is competitive with MW. For the cases where it is slightly worse (LU, Barnes and Bodytrack), the cause is higher remote L1 hits in DW than in MW. This is because in MW, the first reader forces the last writer to writeback to L2. Thus, subsequent readers get their data from L2 for MW but need to go to the remote L1 (via L2) for DW, slightly increasing the memory stall time for DW. However, in terms of network traffic, DW always significantly outperforms MW.

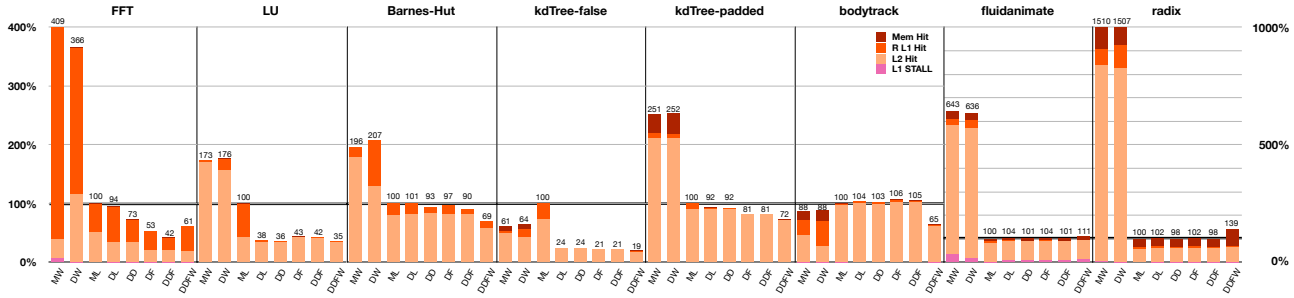
MESI vs. DeNovo line protocols (ML vs. DL): DL shows about the same or better memory stall times as ML. For LU and kdTree-false, DL shows 62% and 76% reduction in memory stall time over ML, respectively. Here, DL enjoys one major advantage over ML: DL incurs no false sharing due to its per-word coherence state. Both LU and kdTree-false contain some false sharing, as indicated by the significantly higher remote L1 hit component in the miss rate count and memory stall time graphs for ML. In terms of network traffic, DL outperforms ML except for fluidanimate and radix. Here, DL incurs more network traffic because registration (write-traffic) is still at word-granularity (shown in 2c). This can be potentially mitigated with a "write-combining" optimization that aggregates individual registration requests similar to a combining write buffer.

Effectiveness of cache lines for MESI: Comparing MW and ML, we see that the memory stall time reduction resulting from transferring a contiguous cache line instead of just a word is highly application dependent. The reduction is largest for radix (a large 93%), which has dense arrays and no false sharing. Most interestingly, for kdTree-false (object-oriented AoS style with false sharing), the word based MESI does better than the line based MESI by 39%. This is due to the combination of false sharing and less than perfect spatial locality. Bodytrack is similar in that it exhibits little spatial locality due to its irregular access pattern. Consequently, ML shows higher miss counts and memory stall times than MW (due to cache pollution from the useless words in a cache line).

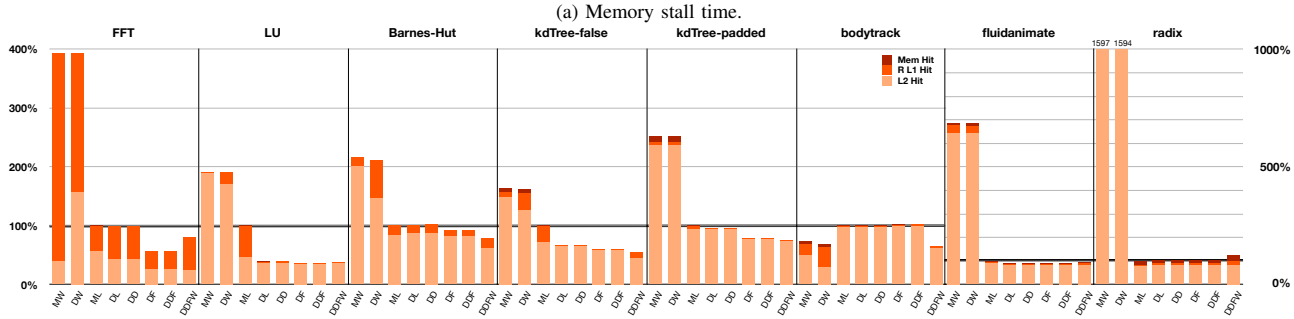
Effectiveness of cache lines for DeNovo: Comparing DW with DL, we see again the strong application dependence of the effectiveness of cache lines. However, because false sharing is not an issue with DeNovo, both LU and kdTree-false enjoy larger benefits from cache lines than in the case of MESI (78% and 63% reduction in memory stalls). Analogous to MESI, Bodytrack sees larger memory stalls with DL than with DW because of little spatial locality.

Effectiveness of direct cache-to-cache transfer with DL: FFT and barnes exhibit much opportunity for direct cache-to-cache transfer. For these applications, DD is able to significantly reduce the remote L1 hit latencies when compared to DL.

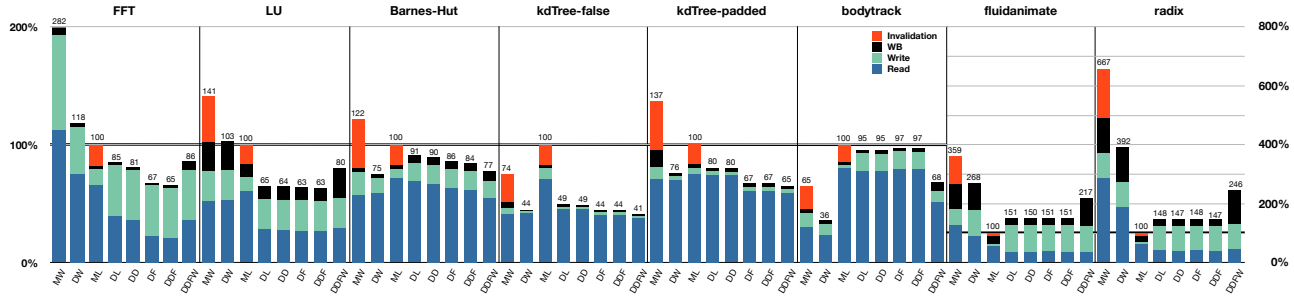
Effectiveness of flexible communication granularity with DL: DF performs about as well or better than ML and DL for all cases, except for LU. LU does not do as well because of the line granularity for cache allocation (addresses). DF can bring in data from multiple cache lines; although this data is likely to be useful, it can potentially replace a lot of allocated data. Bodytrack shows a similar phenomenon, although to a much lesser extent. As we see later, flexible communication at word



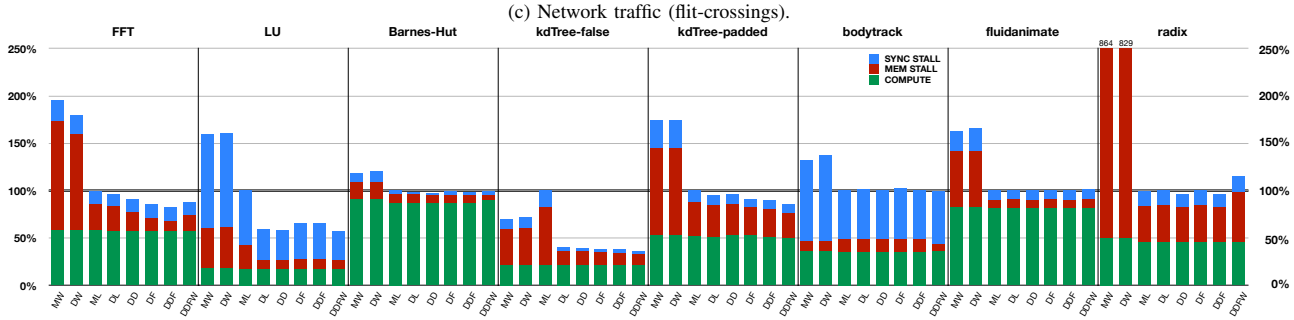
(a) Memory stall time.



(b) Read miss counts.



(c) Network traffic (flit-crossings).



(d) Execution time.

Fig. 2: Comparison of MESI vs. DeNovo protocols. All bars are normalized to the corresponding ML protocol.

address granularity does much better for LU and Bodytrack. Overall, DF shows up to 79% reduction in memory stall time over ML and up to 44% over DL. These results are pessimistic since we did not transfer more than 64 bytes of data at a time.

Effectiveness of combined optimizations with DL: DDF combines the benefits of both DD and DF to show either about the same or better performance than all the other line based protocols (except for LU for reasons described above).

Effectiveness of combined optimizations with DW: For applications like LU and bodytrack with low spatial locality, word-based protocols have the advantage over line based protocols by not bringing in potentially useless data and/or not replacing potentially useless data. We find that DW with our two optimizations (DDFW) does indeed perform better than DDF for these two applications. In fact, DDFW does better for 5 out of the 8 applications. This motivates our future work on using a more software-aware (region based) address granularity to get the best benefit of our optimizations.

Effectiveness of regions and touched bits: To evaluate the effectiveness of regions and touched bits, we ran DL without them. This resulted in all the valid words in the cache being invalidated by the self-invalidation instruction. Our

results (not shown in detail) show 0% to 25% degradation for different applications, which indicates that these techniques are beneficial for some applications.

Protocol verification results: Through model checking, we found three bugs in DeNovo and six bugs including two deadlock scenarios in MESI. Note that DeNovo is much less mature than the GEMS MESI protocol which has been used by many researchers. In DeNovo, all bugs were simple to fix and showed mistakes in translating our internal high level specification into the implementation (i.e., their solutions were already present in our internal high level description of the protocol). In MESI, all the bugs except one of the deadlocks are caused by protocol races between L1 writebacks and other cache events. These involved subtle races and took several days to track, debug and fix. After fixing all the bugs, the model for MESI explores 1,257,500 states in 173 seconds whereas the model for DeNovo explores 85,012 states in 8.66 seconds. Our experience clearly indicates the simplicity and reduced verification overhead for DeNovo compared to MESI.

VI. Related Work

There is a vast body of work on improving the shared-memory hierarchy, including coherence protocol optimizations (e.g., [51, 55, 54, 63, 66]), relaxed consistency models [30, 32], using coarse-grained (multiple *contiguous* cache lines, also referred to as regions) cache state tracking (e.g., [23, 59, 71]), smart spatial and temporal prefetching (e.g., [64, 68]), bulk transfers (e.g., [11, 26, 38, 39], producer-initiated communication [2, 48]), recent work specifically for multicore hierarchies (e.g., [12, 37, 72]), and many more. Our work is inspired by much of this literature, but our focus is on a holistic rethinking of the cache hierarchy driven by disciplined software programming models to benefit hardware complexity, performance, and power. Below we elaborate on work that is the most closely related.

The recent SARC coherence protocol [45] exploits the data-race-free programming model [6], but is based on the conventional directory-based MESI protocol. SARC introduces "tear-off, read-only" (TRO) copies of cache lines for self-invalidation and also uses direct cache-to-cache communication with writer prediction to improve power and performance. Their results, like ours, prove the usefulness of disciplined software for hardware. Unlike DeNovo, SARC does not reduce the directory storage overhead (the sharer list) or reduce protocol complexity. Also, in SARC, all the TRO copies are invalidated at synchronization points while in DeNovo, as shown in Section V, region information and touched bits provide an effective means for selective self-invalidation. Finally, SARC does not explore flexible communication granularity since it does not have the concept of regions and also it is susceptible to false sharing.

Other efforts target one or more of the cache coherence design goals at the expense of other goals. For example, the work in [51] uses self-invalidations but introduces a much more complex protocol. The work in [47] does not incur complexity but requires traffic-heavy flushing of all dirty lines to the global shared cache at the end of each phase with some assumptions about the programming model. Another compiler-hardware coherence approach [58] does not support remote cache hits, instead they require writes to a shared-level cache if there is a potential inter-phase dependency. The SWEL protocol [62] and Atomic Coherence [67] work to simplify the protocol at the expense of relying on limited interconnect substrates. SWEL dynamically places read-write shared data in the lowest common level of shared cache and uses a bus for invalidation. Atomic Coherence uses nanophotonics to guard each coherence action with a mutex. Both protocols eliminate transient states, but limit the network.

Philosophically, the software distributed shared memory literature is also similar, where the system exploits data-race-freedom to allow large granularity communication (virtual pages) without false sharing (e.g., [5, 15, 24, 19]). These techniques mostly rely on heavyweight mechanisms like virtual memory management, and have struggled to find an appropriate high-level programming model. Recent work [31] reduces performance overheads through hardware support.

Some work has also abandoned cache coherence altogether [41] at the cost of significant programming complexity.

VII. Conclusions and Future Work

This paper takes the stance that disciplined programming models will be essential for software programmability and clearly specifiable hardware/software semantics, and asks how such models impact hardware. The paper shows that race-freedom, structured parallel control, and the knowledge of regions and effects in deterministic codes enable much simpler, more extensible, and more efficient cache coherence protocols than the state-of-the-art. This paper is the first step in exploiting what appears to be a tremendous opportunity to rethink multicore memory hierarchies driven by disciplined software models. There are several avenues of future work: extending the ideas here to the main memory system; extending to handle other forms of disciplined and non-disciplined codes (e.g., disciplined non-deterministic codes, synchronization, and legacy codes); using regions to drive address (cache allocation) and coherence granularity; more realistic implementations of the optimizations explored here; and automating the generation of hardware regions and communication spaces through a compiler/runtime implementation.

References

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

[2] H. Abdel-Shafi et al. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *HPCA*, 1997.

[3] D. Abts et al. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *IPDPS*, 2003.

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

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

[6] S. V. Adve and M. D. Hill. Weak Ordering - A New Definition. In *Proc. 17th Intl. Symp. on Computer Architecture*, pages 2–14, May 1990.

[7] V. S. Adve and L. Ceze. *Workshop on Deterministic Multiprocessing and Parallel Programming, U-Washington*, 2009.

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

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

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

[11] R. H. Arpaci et al. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *ISCA*, pages 320–331, June 1995.

[12] A. Basu et al. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *MICRO*, 2007.

[13] A. Baumann et al. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *SOSP*, 2009.

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

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

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

[17] C. Bienia et al. Fidelity and scaling of the parsec benchmark inputs. In *Workload Characterization (IISWC), 2010 IEEE International Symposium on*, 2010.

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

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

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

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

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

[23] J. Cantin et al. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *ISCA*, pages 246–257, June 2005.

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

[25] K. Chakraborty et al. Computation Spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 283–292, New York, NY, USA, 2006. ACM.

[26] R. Chandra et al. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *ICS*, 1994.

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

[28] S. Curial et al. Mpads: memory-pooling-assisted data splitting. In *ISMM*, pages 101–110, 2008.

[29] D. L. Dill et al. Protocol Verification as a Hardware Design Aid. In *ICCD '92*, pages 522–525, Washington, DC, USA, 1992. IEEE Computer Society.

[30] M. Dubois et al. Delayed Consistency and its Effects on the Miss Rate of Parallel Programs. In *SC*, pages 197–206, 1991.

[31] C. Fensch and M. Cintra. An OS-based alternative to full hardware coherence on tiled CMPs. In *HPCA*, 2008.

[32] K. Gharachorloo et al. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *ISCA*, pages 15–26, May 1990.

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

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

[35] N. Gustafsson. Axum: Language Overview. Microsoft Language Specification, 2009.

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

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

[38] K. Hayashi et al. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. In *ASPLOS*, pages 196–207, 1994.

[39] J. Heinlein et al. Coherent Block Data Transfer in the FLASH Multiprocessor. In *ISPP*, pages 18–27, 1997.

[40] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2007.

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

[42] G. C. Hunt and J. R. Larus. Singularity: Rethinking the Software Stack. In *ACM SIGOPS Operating Systems Review*, 2007.

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

[44] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *PPOPP*, pages 179–188, 1995.

[45] S. Kaxiras and G. Keramidas. SARC Coherence: Scaling Directory Cache Coherence in Performance and Power. *IEEE Micro*, 30(5):54 –65, Sept.-Oct. 2010.

[46] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy Release Consistency for Software Distributed Shared Memory. In *ISCA*, pages 13–21, 1992.

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

[48] D. A. Koufaty et al. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *SC*, pages 255–264, 1995.

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

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

[51] A. R. Lebeck and D. A. Wood. Dynamic Self-Invalidation: Reducing Coherence Overhead in Shared-Memory Multiprocessors. In *ISCA*, pages 48–59, Jun 1995.

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

[53] B. Lucia et al. Conflict Exceptions: Simplifying Concurrent Language Semantics with Precise Hardware Exceptions for Data-Races. In *ISCA*, 2010.

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

[55] M. M. Martin et al. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *ISCA*, 2003.

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

[57] M. R. Marty et al. Improving Multiple-CMP Systems Using Token Coherence. In *HPCA*, pages 328–339, 2005.

[58] 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.

[59] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *ISCA*, 2005.

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

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

[62] S. H. Pugsley et al. SWEL: Hardware Cache Coherence Protocols to Map Shared Data onto Shared Caches. In *PACT*, 2010.

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

[64] S. Somogyi et al. Spatial Memory Streaming. In *ISCA*, pages 252–263, 2006.

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

[66] K. Strauss et al. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *ISCA*, pages 327–338, 2006.

[67] D. Vantrease et al. Atomic Coherence: Leveraging Nanophotonics to Build Race-Free Cache Coherence Protocols. In *HPCA*, 2011.

[68] T. Wenisch et al. Temporal Streaming of Shared Memory. In *ISCA*, pages 222–233, 2005.

[69] S. C. Woo et al. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *ISCA*, 1995.

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

[71] J. Zebchuk et al. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In *MICRO*, pages 314–327, 2007.

[72] J. Zebchuk et al. A Tagless Coherence Directory. In *MICRO*, 2009.