DeNovo: Rethinking Hardware for Disciplined Parallelism [*]

Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Bocchino, Sarita Adve, and Vikram Adve
University of Illinois at Urbana-Champaign
denovo@cs.illinois.edu

Abstract

We believe that future large-scale multicore systems will require disciplined parallel programming practices, including data-race-freedom, deterministic-by-default semantics, and structured, explicit parallel control and side-effects. We argue that this software evolution presents far-reaching opportunities for parallel hardware design to greatly improve complexity, power-efficiency, and performance scalability. The DeNovo project is rethinking hardware design from the ground up to exploit these opportunities. This paper presents the broad research agenda of DeNovo, including a holistic rethinking of cache coherence, memory consistency, communication, and cache architecture.

1 Introduction

Achieving the promise of Moore's law will require harnessing increasing amounts of parallelism using multicore architectures. Industry experts project over a thousand cores per chip in about a decade [16]. Unfortunately, designing easily programmable large-scale parallel hardware that provides scalable and power-efficient performance at low-cost remains a major challenge. Current designs for large-scale shared-memory systems rely on directory-based cache coherence protocols for scalability [37], which are extremely complex and inefficient. Moreover, current memory hierarchies are based on outdated organizing principles such as contiguous cache lines that worked well for dense-array codes but are not well-suited for modern object-oriented codes and pointer-based data structures.

On the software side, threads-based shared memory, arguably the most widely used general-purpose parallel programming model, is known to be difficult to program, debug, and maintain [36]. Current models are not only conceptually more difficult to understand than the sequential model (e.g., due to data races and ubiquitous nondeterminism), but require abandoning decades of advances at the core of robust sequential software engineering practices (e.g., safety, modularity, and composability).

For both hardware and software, the problem of formally specifying the fundamental property of the memory model or memory consistency model has been surprisingly challenging. The memory model specifies what values a shared-memory read may return [4]. After decades of research and vigorous debate, there has finally been a convergence centered on providing sequential consistency for data-race-free programs [15, 39]. The results, however, are deeply unsatisfactory and have exposed fundamental shortcomings in today's hardware and software systems [3]. First, safe languages such as Java require precisely defined behavior even with (unintended) data races – this has made the Java model incredibly complex [39] (it currently has an unresolved bug [46]). Second, the software-oblivious evolution of legacy hardware has led to an unnecessary performance compromise with simple models – this has forced a complex supplemental model for C and C++, intended only for performance-driven experts [15].

We believe the above problems are not inherent to a shared address space paradigm. Instead, they occur due to undisciplined programming models that allow the use of arbitrary reads and writes in implicit and unstructured communication and synchronization. This results in "wild shared-memory" behaviors with unintended data races, non-deterministic executions, and implicit side effects that make programs hard to understand, debug, and maintain. The same phenomena result in complex hardware that must assume that any memory access may trigger communication, and inefficient hardware that is unable to exploit communication patterns known to the programmer but obfuscated by the programming model.

Our thesis is that more disciplined programming models with explicit and structured communication and synchronization can address the above problems in both hardware and software. Previously, we described a research agenda for deterministic-by-default languages to provide the discipline needed to address the software programmability [13] and memory model related issues [3]. This paper describes a hardware research agenda pursued by the Illinois DeNovo project. DeNovo aims to show that such disciplined programming models allow far more scalable and power-efficient hardware at much lower complexity than state-of-the-art software-oblivious design approaches.

We expect three features in future disciplined parallel programs: (1) data-race-freedom, and even guaranteed deterministic semantics in many cases; (2) structured par-

[*] This work is supported in part by the Intel and Microsoft funded Universal Parallel Computing Research Center at Illinois.

allel control; and (3) explicit specification of the effects of shared-memory accesses; e.g., which (possibly non-contiguous) regions of memory will be read or written in a parallel section (Section 2). We show these features enable a fundamental rethinking of shared-memory hardware for superior performance, power efficiency, and complexity, as follows.

Coherence and Consistency. First, structured parallel control and knowing which memory regions will be read/written enable a cache to take responsibility for invalidating its own stale data. Such self-invalidations remove the scalability-limiting need for a hardware directory to track sharer lists and to send invalidations on writes. Second, data-race-freedom eliminates concurrent conflicting accesses and corresponding transient states in coherence protocols, eliminating a major source of complexity. Third, since there is no need to track sharers or serialize conflicts, cache-to-cache transfers can occur without indirection through the directory, significantly reducing latency. Fourth, if software guarantees data-race-freedom, then hardware can easily make strong memory model guarantees. The result is much simpler, lower-latency coherence protocols and simple (yet high-performance) consistency models (Section 3.1).

Communication and Storage Layout. A key organizing principle for memory hierarchies is a cache line, which is used for address, communication (transfer), and coherence granularity. While this works well for uniprocessors with dense array codes, it does not naturally extend to multicores (e.g., it can incur false sharing) or to object-oriented codes (where a computation phase may access only one field in a contiguous array of large structs, wasting bandwidth and cache storage). Explicit shared-memory effects allow customizing the address, communication, and coherence granularity around software-specified regions. Such a reorganization can give much higher efficiency in communication latency, bandwidth, and cache storage and bookkeeping; e.g., bulk data transfers and cache space usage for only the needed data (Sections 3.2 and 3.3).

Together, our observations lead to systems that enjoy the benefits of a global address space along with the efficiencies of message passing; e.g., point to point communication without indirection, bulk transfer of only the required data, and simple, scalable hardware with clear semantics.

2 DeNovo Research Strategy and Disciplined Languages

Future systems will run a mix of disciplined software and legacy, "wild shared memory" code. We expect, however, that the latter will be a decreasing fraction for the reasons described above. There are already a large number of research and commercial projects developing new disciplined parallel programming models for deterministic and non-deterministic algorithms [5]; e.g., Ct [24], CnC [17], Cilk++ [11], Galois [33], SharC [7], Kendo [44], Prometheus [6], Grace [10], Axum [26], and DPJ [14]. Most of these, including all but one of the commercial systems, *guarantee* the absence of data races for programs that type-check, satisfying the first requirement of our work immediately. Moreover, most of these 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.

We approach our goal of exploiting disciplined programming for hardware in stages. We begin with deterministic codes for three reasons: (1) there is a growing view that deterministic algorithms will be common, at least for client-side computing [5]; (2) focusing on these codes allows us to investigate the "best case," i.e., the potential for gains from exploiting strong discipline; and (3) these investigations will form a basis on which we develop the extensions needed for other classes of codes. We then investigate how to extend this system to support disciplined non-determinism. Finally, we consider legacy software and programming models. Synchronization mechanisms are used with all three kinds of codes, but we discuss them with legacy software because synchronization inherently involves races.

We take Deterministic Parallel Java (DPJ) [14] as an exemplar of the emerging class of disciplined languages. We use it to explore how hardware can take advantage of data-race-freedom, structured parallel control, and explicit read and write effects of concurrent tasks. 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 from our initial study the types of information that is most valuable, we will explore how it can be extracted from programs in different languages.

2.1 Deterministic Parallel Java (DPJ)

DPJ is an extension to Java that enforces *deterministic-by-default* semantics via compile-time type checking. Using Java is not essential; similar extensions for C++ are underway. DPJ provides a new type and effect system for expressing important patterns of deterministic and non-deterministic parallelism in imperative, object-oriented programs. Non-deterministic behavior can only be obtained via certain explicit constructs (Section 4). 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 implies an obvious sequential equivalent of the parallel program (e.g., `for` replaces foreach), and restricts the result of a parallel execution to that of 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 and data layout, discussed next.

3 DeNovo for Deterministic Codes

3.1 Coherence and Consistency

Sequential equivalence for deterministic codes implies a read should simply return the value of the last conflicting write before it in the sequential program order. This write is either from the reader's own task or from a task in a previous parallel phase, since there can be no concurrent conflicting write. In contrast, conventional coherence protocols, typically based on directories, assume that writes and reads to the same location can occur concurrently, resulting in significant complexity and inefficiency.

DeNovo eliminates the drawbacks of conventional directory protocols as follows. For now, assume a single-word cache line and no data races at this granularity (relaxed later). Without loss of generality, assume private, writeback L1 caches, a shared last-level on-chip L2 cache inclusive of only the modified lines in L1, a single (multicore) processor chip system, and no task-migration.

No directory storage or write invalidation overhead. In conventional 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. 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.

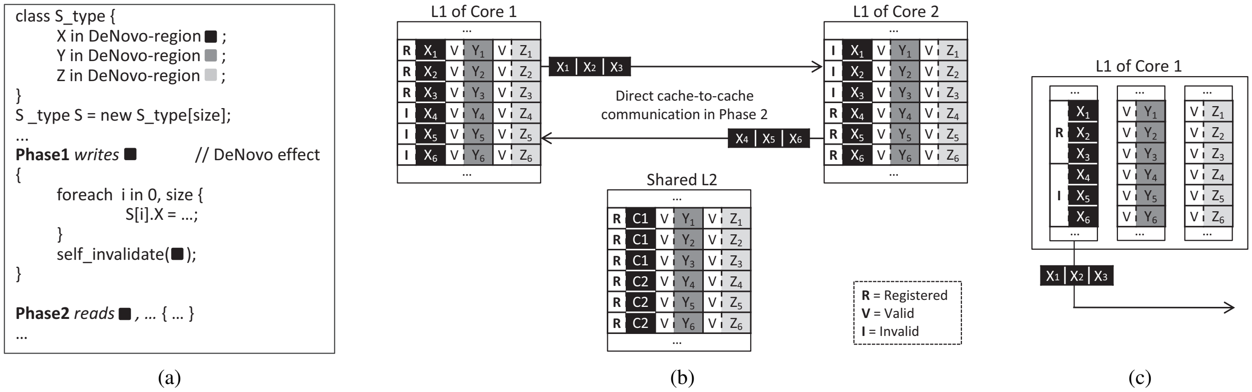
For (i), each cache simply uses the known write effects of the current phase to invalidate its outdated data before the next phase begins. The compiler inserts self-invalidation instructions for each region with these write effects. Each L1 invalidates its data that belongs to these regions except ones that it has "touched" in this phase, since such data are known to be up-to-date. We augment each line with one "touched" bit for this purpose. For (ii), DeNovo requires that on a write, a core register itself at (i.e., inform) the shared L2. The L2 data banks serve as the registry – they either keep the identity of an L1 that has the up-to-date data or the data itself. Thus, DeNovo entails *zero overhead for directory (registry) storage*.

No transient states. The DeNovo protocol has three states in the L1 and L2 – registered, valid, and invalid – with obvious meaning. (The touched bit mentioned above is local to its cache and irrelevant to external coherence transactions.) Although textbook descriptions of conventional directory protocols also describe 3 to 5 states (e.g., MSI) [30], it is well-known that they contain many hidden transient states due to races, making them notoriously complex and difficult to verify [2]. 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, and they can be handled as in uniprocessors. We are currently investigating formal verification to quantify the impact of this significant simplification.

Eliminating indirection. In a conventional protocol, even misses that are eventually serviced by another cache (cache-to-cache transfer) must go through the directory, incurring an additional latency due to the indirection. Since DeNovo does not maintain sharer lists, a reader can potentially directly obtain data from a cache that has it, without informing the registry. Knowledge of which cache may have the data can either be obtained through hardware prediction or compiler/runtime support using effects information. This can be conservative since the request can always be sent to the registry if the predicted cache does not have the line.

Hardware regions, an example, and evaluation. A key research question is how to distinguish regions in hardware for self-invalidations. Language-level regions are more fine-grain than may be practical, or needed, in hardware. The language may need to distinguish fields of each object in an array or tree to prove non-interference. Hardware only needs to identify the aggregate set of data that could be written in a phase, not which core wrote what. The compiler can thus summarize one or more fields of an entire array or tree of objects as a single region, dramatically reducing the number of regions for hardware. At the same time, over-coarsening of regions may lead to conservative write effects and unnecessary "collateral" self-invalidations, requiring the compiler to balance the number of hardware regions against the precision of effects. The ideal hardware-software interface through which region information can be conveyed also remains a research question (e.g., through memory instructions or data). Regardless of how it is conveyed, the caches need to track the region information with the data. Section 3.3 proposes a new cache design that can eliminate this overhead.

Figure 1 illustrates the above concepts. Figure 1(a) shows a code fragment with parallel phases accessing an



```
class S_type {
        X in DeNovo-region ■ ;
        Y in DeNovo-region ▩ ;
        Z in DeNovo-region □ ;
}
S _type S = new S_type[size];
...
Phase1 writes ■             // DeNovo effect
{
        foreach  i in 0, size {
                    S[i].X = ...;
        }
        self_invalidate(■);
}

Phase2 reads ■ , ... { ... }
...
```

Figure 1: (a) Code with DeNovo regions and self-invalidations, (b) cache state after phase 1 self-invalidations and direct core-to-core communication at the beginning of phase 2, and (c) region-driven cache layout. 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.

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.

We implemented the DeNovo protocol without the optimization to eliminate indirection, using the Wisconsin GEMS [42] framework. We manually performed region aggregation and inserted self-invalidations into three SPLASH-2 applications [50] (`barnes`, `LU`, and `FFT`) and a complex graphics code [20], covering both pointer- and array-intensive codes. We found that a small number (<9) of DeNovo regions minimized collateral invalidations in all cases, and the DeNovo L1 cache miss rates and execution times were almost identical to those of the GEMS MESI protocol (with single word lines). These results show that the simplicity of the DeNovo protocol does not compromise performance, and requires distinguishing only a few regions. We next address performance.

3.2 Communication Efficiency

Conventionally, cache lines form the basis of address (tag), communication (transfer), and coherence granularity. So far, DeNovo operates on single word lines, sacrificing efficiencies from higher communication and address granularity for no false sharing. This section describes how effect information can enable much more flexible (hence performance- and power-efficient) communication granularity than possible today, while the next section enables flexible address and coherence granularities.

Our key insight is that any valid or registered data can be proactively copied to another cache as valid (but not touched), without involving the registry. When (if) a demand read accesses this copy, it is marked touched. A demand read implies there is no concurrent conflicting write, so it is correct to read this value (valid) and not self-invalidate at the end of the phase (touched). Thus, when servicing a demand read, a cache may send an arbitrary amount of valid data along with the accessed word. Such a transfer does not incur false sharing or state downgrades since nobody loses "ownership."

Using the above insight, DeNovo can easily exploit conventional cache line sizes for communication and address granularity. A read miss response can always return a cache line worth of information although some words may be invalid (marked using per-word coherence state, analogous to sector caches [38]). This reduces address tag overhead and exploits spatial locality without false sharing.

For higher efficiency than afforded by conventional cache lines, we observe that often only a few words from a cache line are used; the rest simply waste bandwidth and storage. For example, in object-oriented programs, data structures are often in "array of structs" (AoS) rather than "struct of arrays" (SoA) layout. AoS is wasteful if only a few fields of the structs are accessed. An AoS-to-SoA transformation in software is challenging [21, 31]. DeNovo can exploit effects information to easily achieve the same goal. Thus, a read miss response can transfer only the words in regions that will be accessed in this phase. More generally, the compiler may associate a default granularity with each region that defines the size of each contiguous region element and the number of such elements to transfer at a time, to provide a highly flexible bulk communication mechanism.

The above flexible bulk transfers can occur between a producer and consumer without registry indirection, and can be either producer- or consumer-initiated. The net effect is that of seamlessly integrated message-passing-like interactions, with corresponding efficiencies where applicable. Figure 1(b) illustrates these concepts for our example, showing direct communication between cores, transferring only the region for X.

3.3 Storage Efficiency

DeNovo's address granularity is still a contiguous cache line. Thus, even if a read returns only the parts of the line(s) that will be used, the cache must allocate (invalid and wasted) space for the rest of the line(s). We use region/effect information for a more efficient storage layout, with flexible address and coherence granularity.

We first use DeNovo's aggregated regions to control main memory layout in software. The key idea is to lay out a region holding a field of a data structure in strided fashion (e.g., by allocating all elements of the data structure from a contiguous memory pool [34]), to enable regular addressing. For cache layout, we can now partition the cache into multiple banks corresponding to different aggregated regions. Regions accessed together in a phase should be aggregated together in the cache and form the basis for address and transfer granularity. Regions that have similar sharing behavior in a phase should form the basis for coherence granularity. These granularities can be further increased by incorporating task granularity information from the scheduler, further amortizing state maintenance overhead. Overall, region-based cache layout can significantly improve cache utilization and state overhead (along with the previously discussed improved bandwidth, latency, and flexible transfer granularity). Figure 1(c) illustrates these concepts for our example. It shows separate banks for the regions of X, Y, and Z. Each bank merges coherence states of the fields accessed together as shown.

4 Nondeterministic Codes

The key difference between nondeterministic and deterministic codes is that the former may incur conflicting accesses between concurrent tasks, while the latter prohibit them. These accesses usually need to be synchronized using atomicity primitives, which also ensures data-race-freedom. While the specific mechanisms and semantics for disciplined nondeterminism are still a matter of research, we believe some basic principles are required [12]: (1) a guarantee of data-race-freedom by enclosing concurrent conflicting accesses within atomic sections; (2) strong isolation between nondeterministic and deterministic constructs; and (3) serializability for deterministic and nondeterministic constructs to simplify reasoning.

DPJ uses atomic regions and atomic effects as one way to give these guarantees of data-race-freedom, isolation, and serializability [12], and we use them initially to develop support for disciplined nondeterminism in DeNovo. We then discuss how we aim to support less disciplined forms of nondeterminism.

To support atomic sections, DeNovo requires mechanisms to (i) ensure their isolation, and (ii) return appropriate values for their reads. For (i), a naive approach is to use a single lock for each atomic section, which can be efficiently implemented in DeNovo's simplified coherence model by using queue-based locks [25]. Two optimizations are to assign different lock variables to atomic sections that have non-overlapping atomic effects and to enable speculative execution of atomic sections. For (ii), a naive solution is to self-invalidate at the start of each atomic section. Two optimizations are to not invalidate data that have non-conflicting effects, and if the core already owns the lock.

There are also likely to be several sources of less disciplined codes. First, low-level libraries may use wait-free or other "roll-your-own" synchronization. We can treat the regular, but synchronizing, reads and writes in these codes as singleton atomic sections, a discipline similar to DPJ. However, there is little understanding of the access patterns in these codes and how hardware can best support them. We are studying several such codes for a better understanding. Second, we must correctly execute legacy software. One solution is to make a small cluster of the chip fully coherent to execute non-compliant software. An alternative may be to use software distributed shared memory techniques. These solutions do not exploit the DeNovo optimizations, but should be close to what can be achieved through incremental improvements over today's systems, with much lower overall hardware complexity.

Finally, all software must use (racing) synchronization operations, which the DeNovo coherence protocol does not support directly. We propose to implement minimal hardware support for key synchronization primitives, including (queue-based) locks [25], sender-initiated cache-to-cache transfers or "remote writes" [1]) for flag synchronization, and some native support for barriers. More sophisticated mechanisms can be built on top of these primitives.

5 Related Work and Summary

There is a vast body of work on improving shared-memory hierarchy, including coherence protocol optimizations (e.g., [35, 40, 41, 45, 48]), relaxed consistency models [22, 23], using coarse-grained (multiple *contiguous* cache lines, also referred to as regions) cache state tracking (e.g., [18, 43, 51]), smart spatial and temporal prefetching (e.g., [47,49]), bulk transfers (e.g., [8,19,28,29], producer-initiated communication [1, 32]), recent work specifically for multicore hierarchies (e.g., [9, 27, 52]), and more.

The starting point for our work is that current shared-memory programming models are unsustainable for mass-scale parallel programming, motivating more disciplined shared-memory models. With such models as drivers, we rethink the entire memory hierarchy design from the ground up. To our knowledge, our proposed vision is the first that views the cache hierarchy design in such a holistic way, and co-designed together with a disciplined software model. This view allows new ideas (e.g., flexible cache partitions based on software specified regions), simpler and more efficient incarnations of previous ideas (e.g., use of bulk transfer, but with flexible software-driven granularity and with no directory serialization), and a synergistic collection of previously proposed optimizations. The result is a simpler system design that is more performance- and power-efficient and yet more programmable.

References

[1] H. Abdel-Shafi, J. Hall, S. V. Adve, and V. S. Adve. An Evaluation of Fine-Grain Producer-Initiated Communication in Cache-Coherent Multiprocessors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture*, 1997.

[2] D. Abts, S. Scott, and D. J. Lilja. So Many States, So Little Time: Verifying Memory Coherence in the Cray X1. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

[3] S. V. Adve and H.-J. Boehm. Memory Models: A Case for Rethinking Parallel Languages and Hardware. *To appear in the Communications of the ACM*. Author's version is available at http://denovo.cs.illinois.edu/Pubs/10-cacm-memory-models.pdf.

[4] S. V. Adve and K. Gharachorloo. Shared Memory Consistency Models: A Tutorial. *IEEE Computer, Special Issue on Shared-Memory Multiprocessing*, pages 66–76, December 1996.

[5] V. S. Adve and L. Ceze. *Workshop on Deterministic Multiprocessing and Parallel Programming, University of Washington*, Nov 2009.

[6] M. D. Allen, S. Sridharan, and G. S. Sohi. Serialization Sets: A Dynamic Dependence-based Parallel Execution Model. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 85–96, 2009.

[7] Z. Anderson, D. Gay, R. Ennals, and E. Brewer. SharC: Checking Data Sharing Strategies for Multithreaded C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 149–158, 2008.

[8] R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. In *Proceedings of the 22nd International Symposium on Computer Architecture*, pages 320–331, June 1995.

[9] A. Basu, N. Kirman, M. Kirman, M. Chaudhuri, and J. Martinez. Scavenger: A New Last Level Cache Architecture with Global Block Priority. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 421–432, 2007.

[10] E. D. Berger, T. Yang, T. Liu, and G. Novark. Grace: Safe Multithreaded Programming for C/C++. In *Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 81–96, 2009.

[11] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. In *Proceeding of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 207–216, 1995.

[12] R. L. Bocchino, S. Heumann, N. Honarmand, S. Adve, V. Adve, A. Welc, T. Shpeisman, and Y. Ni. Safe Nondeterminism in a Deterministic-by-Default Parallel Language. Technical report, University of Illinois, 2010. Available at http://dpj.cs.uiuc.edu.

[13] R. L. Bocchino, Jr., V. S. Adve, S. V. Adve, and M. Snir. Parallel Programming Must Be Deterministic by Default. In *First USENIX Workshop on Hot Topics in Parallelism*, 2009.

[14] 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 *Proc. 24th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 97–116, 2009.

[15] H.-J. Boehm and S. V. Adve. Foundations of the C++ Concurrency Memory Model. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 68–78, 2008.

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

[17] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari. Multi-core Implementations of the Concurrent Collections Programming Model. In *the 14th International Workshop on Compilers for Parallel Computers*, January 2009.

[18] J. Cantin, M. Lipasti, and J. Smith. Improving Multiprocessor Performance with Coarse-Grain Coherence Tracking. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 246–257, June 2005.

[19] R. Chandra, K. Gharachorloo, V. Soundararajan, and A. Gupta. Performance Evaluation of Hybrid Hardware and Software Distributed Shared Memory Protocols. In *Proceedings of the 8th ACM International Conference on Supercomputing*, Manchester, England, July 1994.

[20] B. Choi, R. Komuravelli, V. Lu, H. Sung, R. L. Bocchino, S. V. Adve, and J. C. Hart. Parallel SAH k-D Tree Construction. Submitted for publication. Also available as a technical report (http://hdl.handle.net/2142/13798), 2010.

[21] S. Curial, P. Zhao, J. Amaral, Y. Gao, S. Cui, R. Silvera, and R. Archambault. MPADS: Memory-Pooling-Assisted Data Splitting. In *Proceedings of the 7th International Symposium on Memory Management*, pages 101–110, 2008.

[22] M. Dubois, J. C. Wang, L. A. Barroso, K. Lee, and Y.-S. Chen. Delayed Consistency and its Effects on the Miss Rate of Parallel Programs. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 197–206, 1991.

[23] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 15–26, May 1990.

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

[25] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. In *Proc. 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 64–75, 1989.

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

[27] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: Near-Optimal Block Placement and Replication in Distributed Caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 184–195, 2009.

[28] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihata, and T. Shindo. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. In *Proc. 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 196–207, 1994.

[29] J. Heinlein, K. Gharachorloo, R. P. Bosch, Jr., M. Rosenblum, and A. Gupta. Coherent Block Data Transfer in the FLASH Multiprocessor. In *Proc. 11th International Symposium on Parallel Processing*, pages 18–27, 1997.

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

[31] T. Jeremiassen and S. J. Eggers. Reducing False Sharing on Shared Memory Multiprocessors through Compile Time Data Transformations. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, 1994.

[32] D. A. Koufaty, X. Chen, D. K. Poulsen, and J. Torrellas. Data Forwarding in Scalable Shared-Memory Multiprocessors. In *Proceedings of the 9th International Conference on Supercomputing*, pages 255–264, 1995.

[33] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. Chew. Optimistic Parallelism Requires Abstractions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 211–222, 2007.

[34] C. Lattner and V. Adve. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[35] 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*, pages 48–59, Jun 1995.

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

[37] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor. In *Proc. 17th Annual International Symposium on Computer Architecture*, 1990.

[38] J. S. Liptay. Structural Aspects of the System/360 Model 85, Part II: The Cache. *IBM Systems Journal*, 7:15 – 21, 1968.

[39] 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*, 2005.

[40] M. M. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. Using Destination-Set Prediction to Improve the Latency/Bandwidth Tradeoff in Shared-Memory Multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[41] M. M. Martin, M. D. Hill, and D. A. Wood. Token Coherence: Decoupling Performance and Correctness. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 182–193, June 2003.

[42] 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.

[43] A. Moshovos. RegionScout: Exploiting Coarse Grain Sharing in Snoop-Based Coherence. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 234–245, June 2005.

[44] M. Olszewski, J. Ansel, and S. Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 97–108, 2009.

[45] A. Raghavan, C. Blundell, and M. M. K. Martin. Token Tenure: PATCHing Token Counting using Directory-Based Cache Coherence. In *Proc. 41st IEEE/ACM International Symposium on Microarchitecture*, pages 47–58, 2008.

[46] J. Ševčík and D. Aspinall. On Validity of Program Transformations in the Java Memory Model. In *Proceedings of European Conference on Object-Oriented Programming*, pages 27–51, 2008.

[47] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Spatial Memory Streaming. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 252–263, 2006.

[48] K. Strauss, X. Shen, and J. Torrellas. Flexible Snooping: Adaptive Forwarding and Filtering of Snoops in Embedded-Ring Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, pages 327–338, 2006.

[49] T. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki, and B. Falsafi. Temporal Streaming of Shared Memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 222–233, 2005.

[50] 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*, pages 24–36, 1995.

[51] J. Zebchuk, E. Safi, and A. Moshovos. A Framework for Coarse-Grain Optimizations in the On-Chip Memory Hierarchy. In *Proc. 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 314–327, 2007.

[52] J. Zebchuk, V. Srinivasan, M. K. Qureshi, and A. Moshovos. A Tagless Coherence Directory. In *Proc. 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.