

DOI:10.1145/1610252.1610271

Easing the programmer's burden does not compromise system performance or increase the complexity of hardware implementation.

**BY JOSEP TORRELLAS, LUIS CEZE, JAMES TUCK,
CALIN CASCAVAL, PABLO MONTESINOS, WONSUN AHN,
AND MILOS PRVULOVIC**

The Bulk Multicore Architecture for Improved Programmability

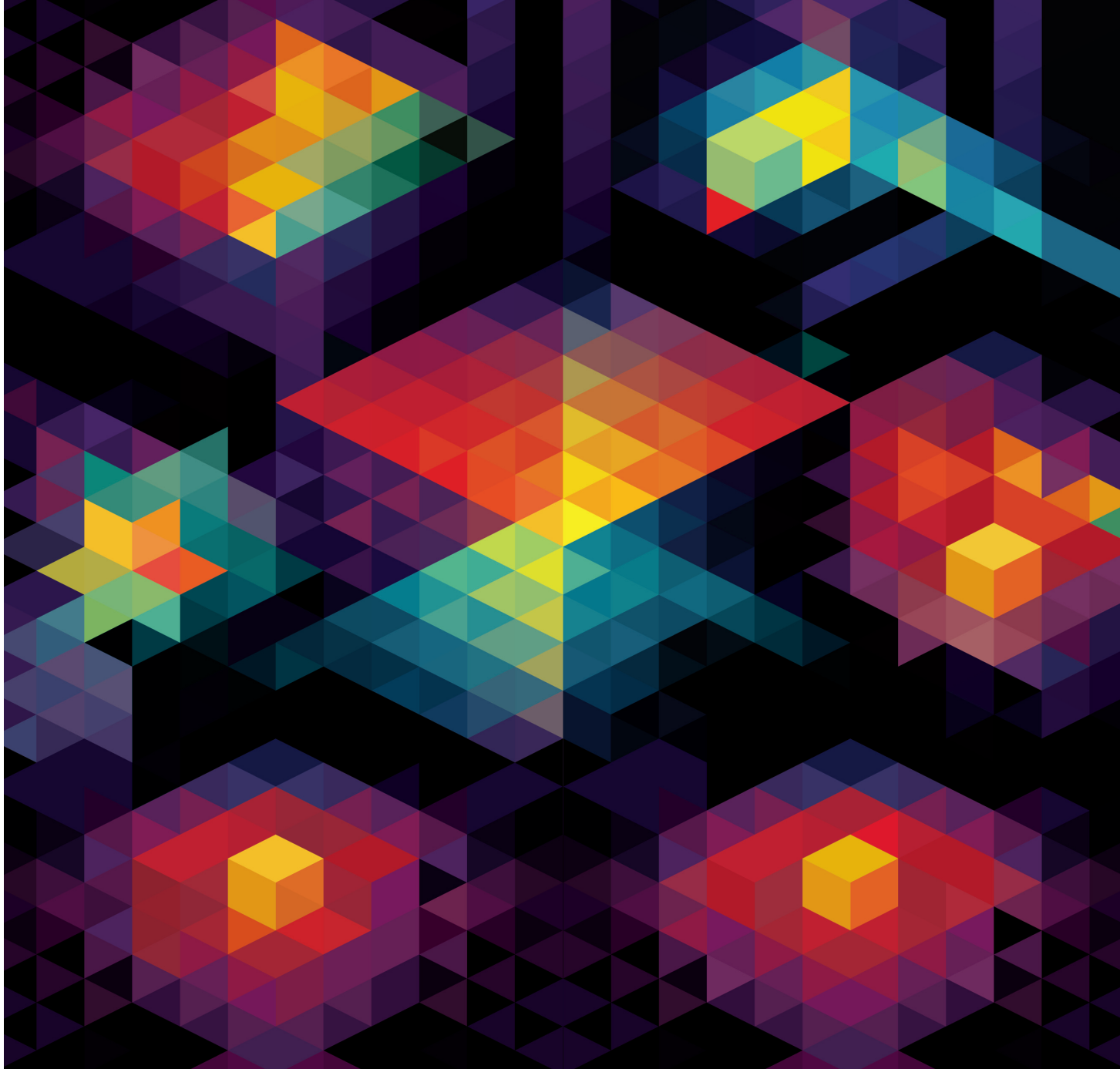
MULTICORE CHIPS AS commodity architecture for platforms ranging from handhelds to supercomputers herald an era when parallel programming and computing will be the norm. While the computer science and engineering community has periodically focused on advancing the technology for parallel processing,⁸ this time around the stakes are truly high, since there is no obvious route to higher performance other than through parallelism. However, for parallel computing to become widespread, breakthroughs are needed in all layers of the computing stack, including languages, programming models, compilation and runtime software, programming and debugging tools, and hardware architectures.

At the hardware-architecture layer, we need to change the way multicore architectures are designed.

In the past, architectures were designed primarily for performance or for energy efficiency. Looking ahead, one of the top priorities must be for the architecture to enable a programmable environment. In practice, programmability is a notoriously difficult metric to define and measure. At the hardware-architecture level, programmability implies two things: First, the architecture is able to attain high efficiency while relieving the programmer from having to manage low-level tasks; second, the architecture helps minimize the chance of (parallel) programming errors.

In this article, we describe a novel, general-purpose multicore architecture—the Bulk Multicore—we designed to enable a highly programmable environment. In it, the programmer and runtime system are relieved of having to manage the sharing of data thanks to novel support for scalable hardware cache coherence. Moreover, to help minimize the chance of parallel-programming errors, the Bulk Multicore provides to the software high-performance sequential memory consistency and also introduces several novel hardware primitives. These primitives can be used to build a sophisticated program-development-and-debugging environment, including low-overhead data-race detection, deterministic replay of parallel programs, and high-speed disambiguation of sets of addresses. The primitives have an overhead low enough to always be “on” during production runs.

The key idea in the Bulk Multicore is twofold: First, the hardware automatically executes all software as a series of atomic blocks of thousands of dynamic instructions called Chunks. Chunk execution is invisible to the software and, therefore, puts no restriction on the programming language or model. Second, the Bulk Multicore introduces the use of Hardware Address Signatures as a low-overhead mechanism to ensure atomic and isolated execution of chunks and help



maintain hardware cache coherence.

The programmability advantages of the Bulk Multicore do not come at the expense of performance. On the contrary, the Bulk Multicore enables high performance because the processor hardware is free to aggressively reorder and overlap the memory accesses of a program within chunks without risk of breaking their expected behavior in a multiprocessor environment. Moreover, in an advanced Bulk Multicore design where the compiler observes the chunks, the compiler can further improve performance by heavily optimizing the instructions within each chunk. Finally, the Bulk Multicore organization decreases hardware

design complexity by freeing processor designers from having to worry about many corner cases that appear when designing multiprocessors.

Architecture

The Bulk Multicore architecture eliminates one of the traditional tenets of processor architecture, namely the need to commit instructions in order, providing the architectural state of the processor after every single instruction. Having to provide such state in a multiprocessor environment—even if no other processor or unit in the machine needs it—contributes to the complexity of current system designs. This is because, in such an environ-

ment, memory-system accesses take many cycles, and multiple loads and stores from both the same and different processors overlap their execution.

In the Bulk Multicore, the default execution mode of a processor is to commit chunks of instructions at a time.² A chunk is a group of dynamically contiguous instructions (such as 2,000 instructions). Such a “chunked” mode of execution and commit is a hardware-only mechanism, invisible to the software running on the processor. Moreover, its purpose is not to parallelize a thread, since the chunks in a thread are not distributed to other processors. Rather, the purpose is to

improve programmability and performance.

Each chunk executes on the processor atomically and in isolation. Atomic execution means that none of the chunk's actions are made visible to the rest of the system (processors or main memory) until the chunk completes and commits. Execution in isolation means that if the chunk reads a location and (before it commits) a second chunk in another processor that has written to the location commits,

then the local chunk is squashed and must re-execute.

To execute chunks atomically and in isolation inexpensively, the Bulk Multicore introduces hardware address signatures.³ A signature is a register of $\approx 1,024$ bits that accumulates hash-encoded addresses. Figure 1 outlines a simple way to generate a signature (see the sidebar "Signatures and Signature Operations in Hardware" for a deeper discussion). A signature, therefore, represents a set of

addresses.

In the Bulk Multicore, the hardware automatically accumulates the addresses read and written by a chunk into a read (R) and a write (W) signature, respectively. These signatures are kept in a module in the cache hierarchy. This module also includes simple functional units that operate on signatures, performing such operations as signature intersection (to find the addresses common to two signatures) and address membership test (to find out whether an address belongs to a signature), as detailed in the sidebar.

Atomic chunk execution is supported by buffering the state generated by the chunk in the L1 cache. No update is propagated outside the cache while the chunk is executing. When the chunk completes or when a dirty cache line with address in the W signature must be displaced from the cache, the hardware proceeds to commit the chunk. A successful commit involves sending the chunk's W signature to the subset of sharer processors indicated by the directory² and clearing the local R and W signatures. The latter operation erases any record of the updates made by the chunk, though the written lines remain dirty in the cache.

The W signature carries enough information to both invalidate stale lines from the other coherent caches (using the δ signature operation on W, as discussed in the sidebar) and enforce that all other processors execute their chunks in isolation. Specifically, to enforce that a processor executes a chunk in isolation when the processor receives an incoming signature W_{inc} , its hardware intersects W_{inc} against the local R_{loc} and W_{loc} signatures. If any of the two intersections is not null, it means (conservatively) that the local chunk has accessed a data element written by the committing chunk. Consequently, the local chunk is squashed and then restarted.

Figure 2 outlines atomic and isolated execution. Thread 0 executes a chunk that writes variables B and C, and no invalidations are sent out. Signature W_0 receives the hashed addresses of B and C. At the same time, Thread 1 issues reads for B and C, which (by construction) load the non-

Signatures and Signature Operations in Hardware

Figure 1 in the main text shows a simple implementation of a signature. The bits of an incoming address go through a fixed permutation to reduce collisions and are then separated in bit-fields C_i . Each field is decoded and accumulated into a bit-field V_j in the signature. Much more sophisticated implementations are also possible.

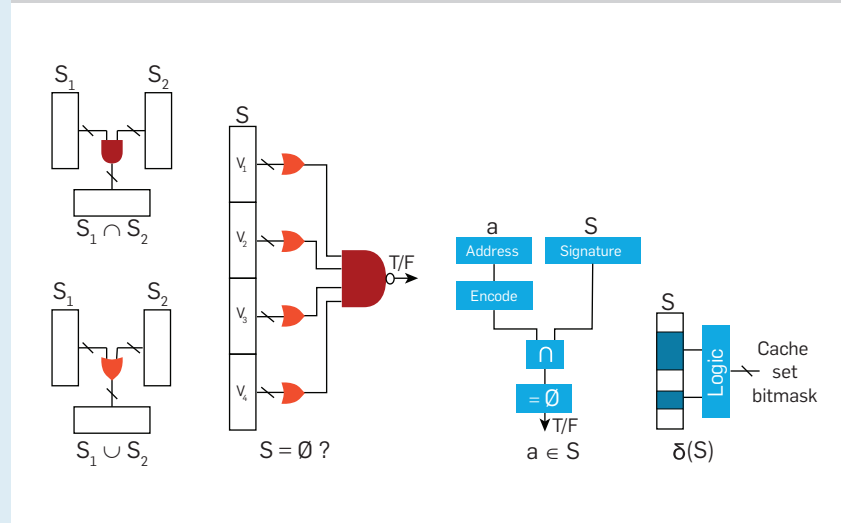
A module called the Bulk Disambiguation Module contains several signature registers and simple functional units that operate efficiently on signatures. These functional units are invisible to the instruction-set architecture. Note that, given a signature, we can recover only a superset of the addresses originally encoded into the signature. Consequently, the operations on signatures produce conservative results.

The figure here outlines five signature functional units: intersection, union, test for null signature, test for address membership, and decoding (δ). Intersection finds the addresses common to two signatures by performing a bit-wise AND of the two signatures. The resulting signature is empty if, as shown in the figure, any of its bit-fields contains all zeros. Union finds all addresses present in at least one signature through a bit-wise OR of the two signatures. Testing whether an address a is present (conservatively) in a signature involves encoding a into a signature, intersecting the latter with the original signature and then testing the result for a null signature.

Decoding (δ) a signature determines which cache sets can contain addresses belonging to the signature. The set bitmask produced by this operation is then passed to a finite-state machine that successively reads individual lines from the sets in the bitmask and checks for membership to the signature. This process is used to identify and invalidate all the addresses in a signature that are present in the cache.

Overall, the support described here enables low-overhead operations on sets of addresses.³

Operations on signatures.



speculative values of the variables—namely, the values before Thread 0's updates. When Thread 0's chunk commits, the hardware sends signature W_0 to Thread 1, and W_0 and R_0 are cleared. At the processor where Thread 1 runs, the hardware intersects W_0 with the ongoing chunk's R_1 and W_1 . Since $W_0 \cap R_1$ is not null, the chunk in Thread 1 is squashed.

The commit of chunks is serialized globally. In a bus-based machine, serialization is given by the order in which W signatures are placed on the bus. With a general interconnect, serialization is enforced by a (potentially distributed) arbiter module.² W signatures are sent to the arbiter, which quickly acknowledges whether the chunk can be considered committed.

Since chunks execute atomically and in isolation, commit in program order in each processor, and there is a global commit order of chunks, the Bulk Multicore supports sequential consistency (SC)⁹ at the chunk level. As a consequence, the machine also supports SC at the instruction level. More important, it supports high-performance SC at low hardware complexity.

The performance of this SC implementation is high because (within a chunk) the Bulk Multicore allows memory access reordering and overlap and instruction optimization. As we discuss later, synchronization instructions induce no reordering constraint within a chunk.

Meanwhile, hardware-implementation complexity is low because memory-consistency enforcement is largely decoupled from processor structures. In a conventional processor that issues memory accesses out of order, supporting SC requires intrusive processor modifications. For example, from the time the processor executes a load to line L out of order until the load reaches its commit time, the hardware must check for writes to L by other processors—in case an inconsistent state was observed. Such checking typically requires sending, for each external coherence event, a signal up the cache hierarchy. The signal snoops the load queue to check for an address match. Additional modifications involve preventing cache displacements that could risk missing a

Figure 1. A simple way to generate a signature.

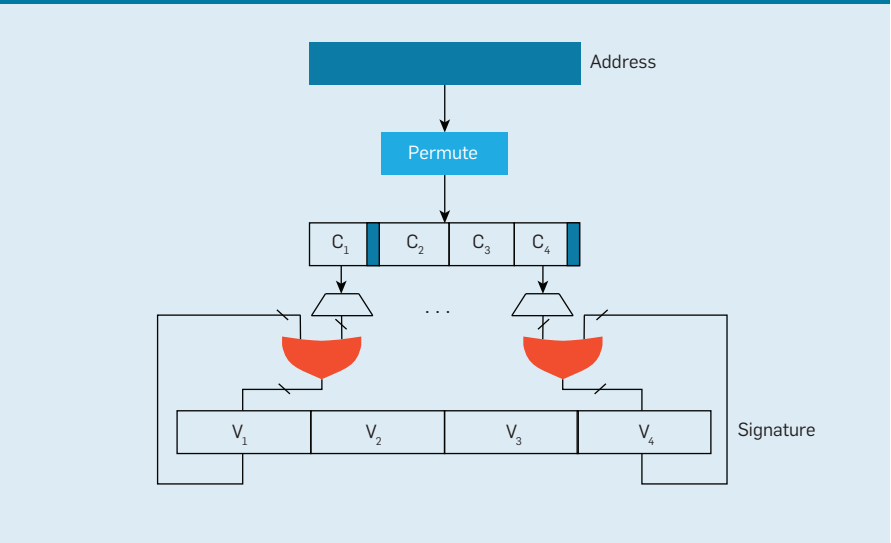
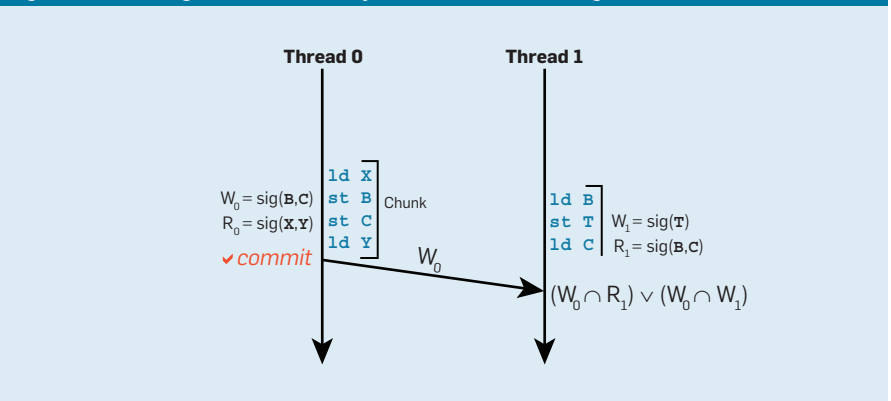


Figure 2. Executing chunks atomically and in isolation with signatures.



coherence event. Consequently, load queues, L1 caches, and other critical processor components must be augmented with extra hardware.

In the Bulk Multicore, SC enforcement and violation detection are performed with simple signature intersections outside the processor core. Additionally, caches are oblivious to what data is speculative, and their tag and data arrays are unmodified.

Finally, note that the Bulk Multicore's execution mode is not like transactional memory.⁶ While one could intuitively view the Bulk Multicore as an environment with transactions occurring all the time, the key difference is that chunks are dynamic entities, rather than static, and invisible to the software.

High Programmability

Since chunked execution is invisible to the software, it places no restriction on programming model, language,

or runtime system. However, it does enable a highly programmable environment by virtue of providing two features: high-performance SC at the hardware level and several novel hardware primitives that can be used to build a sophisticated program-development-and-debugging environment.

Unlike current architectures, the Bulk Multicore supports high-performance SC at the hardware level. If we generate code for the Bulk Multicore using an SC compiler (such as the BulkCompiler¹), we attain a high-performance, fully SC platform. The resulting platform is highly programmable for several reasons. The first is that debugging concurrent programs with data races would be much easier. This is because the possible outcomes of the memory accesses involved in the bug would be easier to reason about, and the debugger would in fact be able to reproduce the buggy interleaving. Second, most existing

software correctness tools (such as Microsoft's CHES¹⁴) assume SC. Verifying software correctness under SC is already difficult, and the state space balloons if non-SC interleavings need to be verified as well. In the next few years, we expect that correctness-verification tools will play a larger role as more parallel software is developed. Using them in combination with an SC platform would make them most effective.

A final reason for the programmability of an SC platform is that it would make the memory model of safe languages (such as Java) easier to understand and verify. The need to provide safety guarantees and enable performance at the same time has resulted in an increasingly complex and unintuitive memory model over the years. A high-performance SC memory model would trivially ensure Java's safety properties related to memory ordering, improving its security and usability.

The Bulk Multicore's second feature is a set of hardware primitives that can be used to engineer a sophisticated program-development-and-debugging environment that is always "on," even during production runs. The key insight is that chunks and signatures free development and debugging tools from having to record or be concerned with individual loads and stores. As a result, the amount of bookkeeping and state required by the tools is substantially reduced, as is the time overhead. Here, we give three examples of this benefit in the areas of deterministic replay of parallel programs, data-race detection, and high-speed disambiguation of sets of addresses.

Note, too, that chunks provide an excellent primitive for supporting popular atomic-section-based techniques for programmability (such as thread-level speculation¹⁷ and transactional memory⁶).

Deterministic replay of parallel programs with practically no log. Hardware-assisted deterministic replay of parallel programs is a promising technique for debugging parallel programs. It involves a two-step process.²⁰ In the recording step, while the parallel program executes, special hardware records into a log the



The Bulk Multicore supports high-performance sequential memory consistency at low hardware complexity.



order of data dependences observed among the multiple threads. The log effectively captures the "interleaving" of the program's threads. Then, in the replay step, while the parallel program is re-executed, the system enforces the interleaving orders encoded in the log.

In most proposals of deterministic replay schemes, the log stores individual data dependences between threads or groups of dependences bundled together. In the Bulk Multicore, the log must store only the total order of chunk commits, an approach we call DeLorean.¹³ The logged information can be as minimalist as a list of committing-processor IDs, assuming the chunking is performed in a deterministic manner; therefore, the chunk sizes can be deterministically reproduced on replay. This design, which we call OrderOnly, reduces the log size by nearly an order of magnitude over previous proposals.

The Bulk Multicore can further reduce the log size if, during the recording step, the arbiter enforces a certain order of chunk commit interleaving among the different threads (such as by committing one chunk from each processor round robin). In this case of enforced chunk-commit order, the log practically disappears. During the replay step, the arbiter reinforces the original commit algorithm, forcing the same order of chunk commits as in the recording step. This design, which we call PicoLog, typically incurs a performance cost because it can force some processors to wait during recording.

Figure 3a outlines a parallel execution in which the boxes are chunks and the arrows are the observed cross-thread data dependences. Figure 3b shows a possible resulting execution log in OrderOnly, while Figure 3c shows the log in PicoLog.

Data-race detection at production-run speed. The Bulk Multicore can support an efficient data-race detector based on the "happens-before" method¹⁰ if it cuts the chunks at synchronization points, rather than at arbitrary dynamic points. Synchronization points are easily recognized by hardware or software, since synchronization operations are executed by special instructions. This approach

is described in ReEnact¹⁶; Figure 4 includes examples with a lock, flag, and barrier.

Each chunk is given a counter value called ChunkID following the happens-before ordering. Specifically, chunks in a given thread receive ChunkIDs that increase in program order. Moreover, a synchronization between two threads orders the ChunkIDs of the chunks involved in the synchronization. For example, in Figure 4a, the chunk in Thread 2 following the lock acquire (Chunk 5) sets its ChunkID to be a successor of both the previous chunk in Thread 2 (Chunk 4) and the chunk in Thread 1 that released the lock (Chunk 2). For the other synchronization primitives, the algorithm is similar. For example, for the barrier in Figure 4c, each chunk immediately following the barrier is given a ChunkID that makes it a successor of all the chunks leading to the barrier.

Using ChunkIDs, we've given a partial ordering to the chunks. For example, in Figure 4a, Chunks 1 and 6 are ordered, but Chunks 3 and 4 are not. Such ordering helps detect data races that occur in a particular execution. Specifically, when two chunks from different threads are found to have a data-dependence at runtime, their two ChunkIDs are compared. If the ChunkIDs are ordered, this is not a data race because there is an intervening synchronization between the chunks. Otherwise, a data race has been found.

A simple way to determine when two chunks have a data-dependence is to use the Bulk Multicore signatures to tell when the data footprints of two chunks overlap. This operation, together with the comparison and maintenance of ChunkIDs, can be done with low overhead with hardware support. Consequently, the Bulk Multicore can detect data races without significantly slowing the program, making it ideal for debugging production runs.

Enhancing programmability by making signatures visible to software. Finally, a technique that improves programmability further is to make additional signatures visible to the software. This support enables inexpensive monitoring of memory accesses, as well as

Making Signatures Visible to Software

We propose that the software interact with some additional signatures through three main primitives:¹⁸

The first is to explicitly encode into a signature either one address (Figure 1a) or all addresses accessed in a code region (Figure 1b). The latter is enabled by the *bcollect* (begin collect) and *ecollect* (end collect) instructions, which can be set to collect only reads, only writes, or both.

The second primitive is to disambiguate the addresses accessed by the processor in a code region against a given signature. It is enabled by the *bdisamb.loc* (begin disambiguate local) and *edisamb.loc* (end disambiguate local) instructions (Figure 1c), and can disambiguate reads, writes, or both.

The third primitive is to disambiguate the addresses of incoming coherence messages (invalidations or downgrades) against a given local signature. It is enabled by the *bdisamb.rem* (begin disambiguate remote) and *edisamb.rem* (end disambiguate remote) instructions (Figure 1d) and can disambiguate reads, writes, or both. When disambiguation finds a match, the system can deliver an interrupt or set a bit.

Figure 2 includes three examples of what can be done with these primitives: Figure 2a shows how the machine inexpensively supports many watchpoints. The processor encodes into signature *Sig2* the address of variable *y* and all the addresses accessed in function *foo()*. It then watches all these addresses by executing *bdisamb.loc* on *Sig2*.

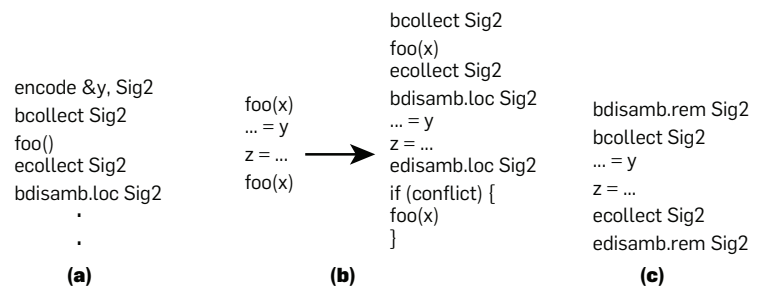
Figure 2b shows how a second call to a function that reads and writes memory in its body can be skipped. In the figure, the code calls function *foo()* twice with the same input value of *x*. To see if the second call can be skipped, the program first collects all addresses accessed by *foo()* in *Sig2*. It then disambiguates all subsequent accesses against *Sig2*. When execution reaches the second call to *foo()*, it can skip the call if two conditions hold: the first is that the disambiguation did not find a conflict; the second (not shown in the figure) is that the read and write footprints of the first *foo()* call do not overlap. This possible overlap is checked by separately collecting the addresses read in *foo()* and those written in *foo()* in separate signatures and intersecting the resulting signatures.

Finally, Figure 2c shows a way to detect data dependences between threads running on different processors. In the figure, *collect* encodes all addresses accessed in a code section into *Sig2*. Surrounding the collect instructions, the code places *disamb.rem* instructions to monitor if any remotely initiated coherence-action conflicts with addresses accessed locally. To disregard read-read conflicts, the programmer can collect the reads in a separate signature and perform remote disambiguation of only writes against that signature.

Figure 1. Primitives enabling software to interact with additional signatures: collection (a and b), local disambiguation (c), and remote disambiguation (d).

	<code>bcollect Sig1</code>	<code>bdisamb.loc Sig1</code>	<code>bdisamb.rem Sig1</code>
	<code>x = ...</code>	<code>x = ...</code>	<code>x = ...</code>
	<code>... = y</code>	<code>... = y</code>	<code>... = y</code>
Encode Addr, Sig1	<code>ecollect Sig1</code>	<code>edisamb.loc Sig1</code>	<code>edisamb.rem Sig1</code>
(a)	(b)	(c)	(d)

Figure 2. Using signatures to support data watchpoints (a), skip execution of functions (b), and detect data dependencies between threads running on different processors (c).



novel compiler optimizations that require dynamic disambiguation of sets of addresses (see the sidebar “Making Signatures Visible to Software”).

Reduced Implementation Complexity

The Bulk Multicore also has advantages in performance and in hardware simplicity. It delivers high performance because the processor hardware can reorder and overlap all memory accesses within a chunk—except, of course, those that participate in single-thread dependences. In particular, in the Bulk Multicore, synchronization instructions do not constrain memory access reordering or overlap. Indeed, fences inside a chunk are transformed into null instructions. Fences’ traditional functionality of delaying execution until certain references are performed is useless; by construction, no other processor observes the actual order of instruction execution within a chunk.

Moreover, a processor can concurrently execute multiple chunks from the same thread, and memory accesses from these chunks can also overlap. Each concurrently executing chunk in the processor has its own R and W signatures, and individual accesses update the corresponding chunk’s signatures. As long as chunks within a processor commit in program order (if a chunk is squashed, its successors are also squashed), correctness is guaranteed. Such concurrent chunk execution in a processor hides the chunk-commit overhead.

Bulk Multicore performance increases further if the compiler generates the chunks, as in the BulkCompiler.¹ In this case, the compiler can aggressively optimize the code within each chunk, recognizing that no other processor sees intermediate states within a chunk.

Finally, the Bulk Multicore needs simpler processor hardware than current machines. As discussed earlier, much of the responsibility for memory-consistency enforcement is taken away from critical structures in the core (such as the load queue and L1 cache) and moved to the cache hierarchy where signatures detect violations of SC.² For example, this property could enable a new environment in

Figure 3. Parallel execution in the Bulk Multicore (a), with a possible OrderOnly execution log (b) and PicoLog execution log (c).

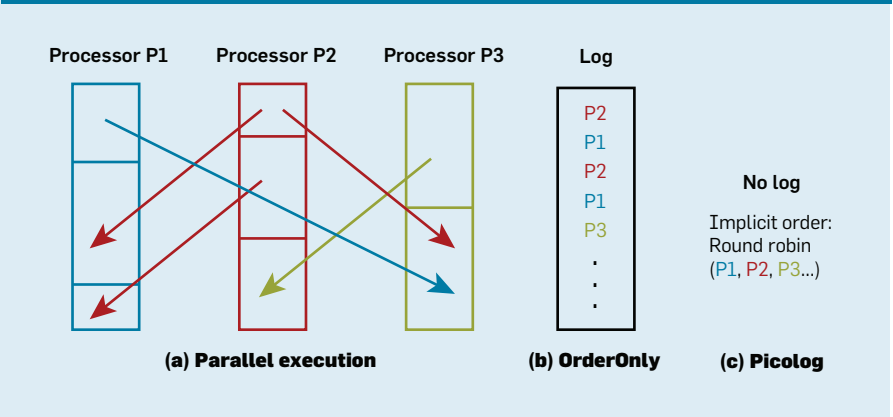
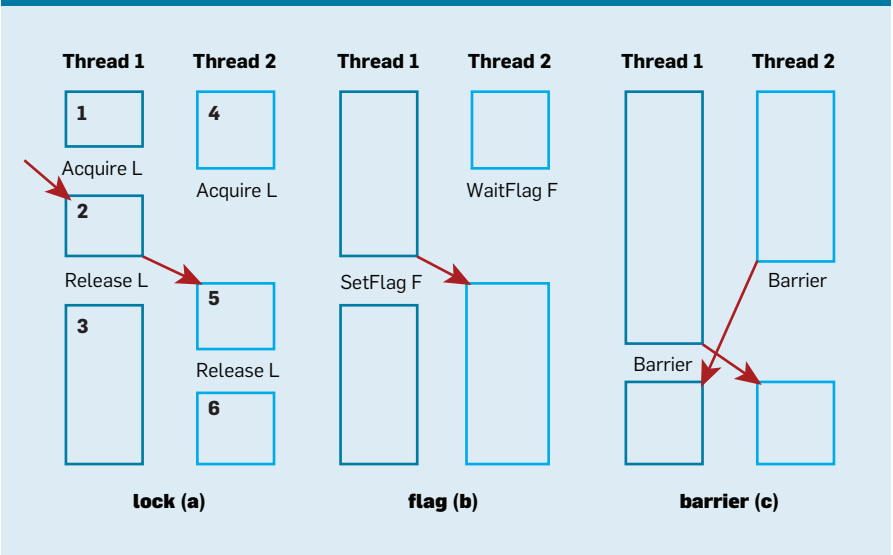


Figure 4. Forming chunks for data-race detection in the presence of a lock (a), flag (b), and barrier (c).



which cores and accelerators are designed without concern for how to satisfy a particular set of access-ordering constraints. This ability allows hardware designers to focus on the novel aspects of their design, rather than on the interaction with the target machine’s legacy memory-consistency model. It also motivates the development of commodity accelerators.

Related Work

Numerous proposals for multiprocessor architecture designs focus on improving programmability. In particular, architectures for thread-level speculation (TLS)¹⁷ and transactional memory (TM)⁶ have received significant attention over the past 15 years. These techniques share key primitive mechanisms with the Bulk Multicore, notably speculative state buffering

and undo and detection of cross-thread conflicts. However, they also have a different goal, namely simplify code parallelization by parallelizing the code transparently to the user software in TLS or by annotating the user code with constructs for mutual exclusion in TM. On the other hand, the Bulk Multicore aims to provide a broadly usable architectural platform that is easier to program for while delivering advantages in performance and hardware simplicity.

Two architecture proposals involve processors continuously executing blocks of instructions atomically and in isolation. One of them, called Transactional Memory Coherence and Consistency (TCC),⁵ is a TM environment with transactions occurring all the time. TCC mainly differs from the Bulk Multicore in that its transactions

are statically specified in the code, while chunks are created dynamically by the hardware. The second proposal, called Implicit Transactions,¹⁹ is a multiprocessor environment with checkpointed processors that regularly take checkpoints. The instructions executed between checkpoints constitute the equivalent of a chunk. No detailed implementation of the scheme is presented.

Automatic Mutual Exclusion (AME)⁷ is a programming model in which a program is written as a group of atomic fragments that serialize in some manner. As in TCC, atomic sections in AME are statically specified in the code, while the Bulk Multicore chunks are hardware-generated dynamic entities.

The signature hardware we've introduced here has been adapted for use in TM (such as in transaction-footprint collection and in address disambiguation^{12,21}).

Several proposals implement data-race detection, deterministic replay of multiprocessor programs, and other debugging techniques discussed here without operating in chunks.^{4,11,15,20} Comparing their operation to chunk operation is the subject of future work.


Future Directions

The Bulk Multicore architecture is a novel approach to building shared-memory multiprocessors, where the whole execution operates in atomic chunks of instructions. This approach can enable significant improvements in the productivity of parallel programmers while imposing no restriction on the programming model or language used.

At the architecture level, we are examining the scalability of this organization. While chunk commit requires arbitration in a (potentially distributed) arbiter, the operation in chunks is inherently latency tolerant. At the programming level, we are examining how chunk operation enables efficient support for new program-development and debugging tools, aggressive autotuners and compilers, and even novel programming models.

Acknowledgments

We would like to thank the many present and past members of the I-acoma

group at the University of Illinois who contributed through many discussions, seminars, and brainstorming sessions. This work is supported by the U.S. National Science Foundation, Defense Advanced Research Projects Agency, and Department of Energy and by Intel and Microsoft under the Universal Parallel Computing Research Center, Sun Microsystems under the University of Illinois OpenSPARC Center of Excellence, and IBM. 

References

- Ahn, W., Qi, S., Lee, J.W., Nicolaidis, M., Fang, X., Torrellas, J., Wong, D., and Midkiff, S. BulkCompiler: High-performance sequential consistency through cooperative compiler and hardware support. In *Proceedings of the International Symposium on Microarchitecture* (New York City, Dec. 12–16). IEEE Press, 2009.
- Ceze, L., Tuck, J., Montesinos, P., and Torrellas, J. BulkSC: Bulk enforcement of sequential consistency. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–13). ACM Press, New York, 2007, 278–289.
- Ceze, L., Tuck, J., Cascaval, C., and Torrellas, J. Bulk disambiguation of speculative threads in multiprocessors. In *Proceedings of the International Symposium on Computer Architecture* (Boston, MA, June 17–21). IEEE Press, 2006, 227–238.
- Choi, J., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., and Sridharan, M. Efficient and precise data-race detection for multithreaded object-oriented programs. In *Proceedings of the Conference on Programming Language Design and Implementation* (Berlin, Germany, June 17–19). ACM Press, New York, 2002, 258–269.
- Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., and Olukotun, K. Transactional memory coherence and consistency. In *Proceedings of the International Symposium on Computer Architecture* (München, Germany, June 19–23). IEEE Press, 2004, 102–113.
- Herlihy, M. and Moss, J.E.B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, May 16–19). IEEE Press, 1993, 289–300.
- Isard, M. and Birrell, A. Automatic mutual exclusion. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (San Diego, CA, May 7–9). USENIX, 2007.
- Kuck, D. Facing up to software's greatest challenge: Practical parallel processing. *Computers in Physics* 11, 3 (1997).
- Lampert, L. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers* C-28, 9 (Sept. 1979), 690–691.
- Lampert, L. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565.
- Lu, S., Tucek, J., Qin, F., and Zhou, Y. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 21–25). ACM Press, New York, 2006, 37–48.
- Minh, C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., and Olukotun, K. An effective hybrid transactional memory with strong isolation guarantees. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–13). ACM Press, New York, 2007, 69–80.
- Montesinos, P., Ceze, L., and Torrellas, J. DeLorean: Recording and deterministically replaying shared-memory multiprocessor execution efficiently. In *Proceedings of the International Symposium on Computer Architecture* (Beijing, June 21–25). IEEE

Press, 2008, 289–300.

- Musuvathi, M. and Qadeer, S. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the Conference on Programming Language Design and Implementation* (San Diego, CA, June 10–13). ACM Press, New York, 2007, 446–455.
- Narayanasamy, S., Pereira, C., and Calder, B. Recording shared memory dependencies using strata. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, CA, Oct. 21–25). ACM Press, New York, 2006, 229–240.
- Prvulovic, M. and Torrellas, J. ReEnact: Using thread-level speculation mechanisms to debug data races in multithreaded codes. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–11). IEEE Press, 2003, 110–121.
- Sohi, G., Breach, S., and Vijayakumar, T. Multiscalar processors. In *Proceedings of the International Symposium on Computer Architecture* (Santa Margherita Ligure, Italy, June 22–24). ACM Press, New York, 1995, 414–425.
- Tuck, J., Ahn, W., Ceze, L., and Torrellas, J. SoftSig: Software-exposed hardware signatures for code analysis and optimization. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (Seattle, WA, Mar. 1–5). ACM Press, New York, 2008, 145–156.
- Vallejo, E., Galluzzi, M., Cristal, A., Vallejo, F., Bevide, R., Stenstrom, P., Smith, J.E., and Valero, M. Implementing kilo-instruction multiprocessors. In *Proceedings of the International Conference on Pervasive Services* (Santorini, Greece, July 11–14). IEEE Press, 2005, 325–336.
- Xu, M., Bodik, R., and Hill, M.D. A 'flight data recorder' for enabling full-system multiprocessor deterministic replay. In *Proceedings of the International Symposium on Computer Architecture* (San Diego, CA, June 9–11). IEEE Press, 2003, 122–133.
- Yen, L., Bobba, J., Marty, M., Moore, K., Volos, H., Hill, M., Swift, M., and Wood, D. LogTM-SE: Decoupling hardware transactional memory from caches. In *Proceedings of the International Symposium on High Performance Computer Architecture* (Phoenix, AZ, Feb. 10–14). IEEE Press, 2007, 261–272.

Josep Torrellas (torrellas@cs.uiuc.edu) is a professor and Willett Faculty Scholar in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Luis Ceze (luisceze@cs.washington.edu) is an assistant professor in the Department of Computer Science and Engineering at the University of Washington, Seattle, WA.

James Tuck (jtuck@ncsu.edu) is an assistant professor in the Department of Electrical and Computer Engineering at North Carolina State University, Raleigh, NC.

Calin Cascaval (cascaval@us.ibm.com) is a research staff member and manager of programming models and tools for scalable systems at the IBM T.J. Watson Research Center, Yorktown Heights, NY.

Pablo Montesinos (pmontesi@samsung.com) is a staff engineer in the Multicore Research Group at Samsung Information Systems America, San Jose, CA.

Wonsun Ahn (dahn2@uiuc.edu) is a graduate student in the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Milos Prvulovic (milos@cc.gatech.edu) is an associate professor in the School of Computer Science, College of Computing, Georgia Institute of Technology, Atlanta, GA.