

The Bulk Multicore Architecture for Improved Programmability

Josep Torrellas, Luis Ceze, James Tuck, Calin Cascaval[†], Pablo Montesinos,
Wonsun Ahn, and Milos Prvulovic

Department of Computer Science
University of Illinois at Urbana-Champaign*

[†]IBM T.J. Watson Research Center

February 6th, 2009
To Appear in Communications of the ACM, December 2009

1 Introduction

The arrival of multicore chips as the commodity architecture for platforms ranging from handhelds to supercomputers foretells an era where parallel programming and computing will be the norm. While the computer science and engineering community has periodically focused on pushing forward the technology for parallel processing [7], this time around the stakes are truly high, since there is no easy route to higher performance other than through parallelism. However, for parallel computing to become widespread, we need breakthroughs in all the layers of the computing stack — including languages, programming models, compilation and run time software, programming and debugging tools, and hardware architectures.

At the hardware architecture layer, we need to change the way in which multicore architectures are designed. In the past, architectures have been primarily designed for performance or for energy efficiency. Moving forward, one of the top priorities should be for the architecture to enable a programmable environment. In practice, programmability is a notoriously difficult metric to define and measure. We suggest that, at the hardware architecture level, programmability implies two things. First, the architecture is able to attain high efficiency while relieving the programmer from managing low-level tasks. Secondly, the architecture helps minimize the chance of (parallel) programming errors.

This paper describes a novel, general-purpose multicore architecture called the *Bulk Multicore* that is designed to enable a highly programmable environment. In the Bulk Multicore, the programmer and run time system are relieved from managing 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. Such primitives can be used to build a sophisticated program development and debugging environment — one with low-overhead data race detection, deterministic replay of parallel programs, and high-speed disambiguation of sets of addresses. Such primitives have an overhead low enough to be “on” during production runs.

*Luis Ceze is now with University of Washington, James Tuck with North Carolina State University, and Milos Prvulovic with Georgia Tech.

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 or *Chunks*. Such chunks are invisible to the software and, therefore, put no restriction on the programming language or model. Secondly, the Bulk Multicore introduces the use of *hardware address signatures* as a low-overhead mechanism to ensure atomic and isolated execution of chunks, and to maintain hardware cache coherence.

The programmability advantages of the Bulk Multicore do not come at the expense of performance. Indeed, the Bulk Multicore enables high performance because the processor hardware (and the software) are free to aggressively reorder and overlap the memory accesses of a program within chunks without risking breaking their expected behavior in a multiprocessor environment. Finally, we will see that the Bulk Multicore organization arguably decreases hardware design complexity by freeing processor designers from worrying about many corner cases that appear when designing multiprocessors.

In the following, we describe the Bulk Multicore organization (Section 2), its programmability advantages (Section 3), how it delivers high performance and reduced hardware complexity (Section 4), and related work (Section 5).

2 The Bulk Multicore 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 — has contributed to the complexity of current system designs. This is because, in such an environment, memory system accesses take many cycles, and multiple loads and stores from the same and different processors overlap their execution.

In the Bulk Multicore, the default execution mode of a processor is to commit only *Chunks* of instructions at a time [2]. A chunk is a group of *dynamically* contiguous instructions, for example 2,000. Such “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.

Each chunk executes on the processor *atomically* and *in isolation*. Atomic execution means that none of the actions of the chunk are made visible to the rest of the system (processors or main memory) until when 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 gets squashed and has to re-execute.

To execute chunks atomically and in isolation inexpensively, the Bulk Multicore relies on *hardware address signatures* [1]. A signature is a register of $\approx 1,024$ bits that accumulates hash-encoded addresses. Figure 1 shows a simple way to generate a signature. The details of this implementation are discussed in Side Bar 1. 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

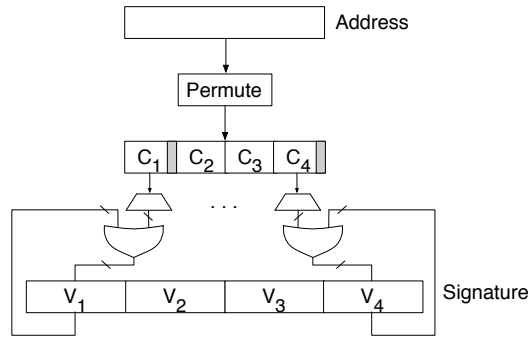


Figure 1: A simple way to generate a signature.

includes simple functional units that operate on signatures efficiently, performing operations such signature intersection (to find the addresses common to two signatures), or address membership test (to find whether an address belongs to a signature). Details of such operations are given in Side Bar 1.

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 whose address is in the W signature needs to be displaced from the cache, the hardware proceeds to commit the chunk. A successful commit simply involves sending the chunk’s W signature to the subset of sharer processors indicated by the directory [2], and clearing the local R and W signatures. The latter operation erases any record of the updates made by the chunk, although the written lines remain dirty in the cache.

The W signature carries enough information both to invalidate stale lines from the other coherent caches (using the δ signature operation on W, as discussed in Side Bar 1), and to 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 (conservatively) means 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 graphically shows atomic and isolated execution. Thread 0 executes a chunk that writes B and C. No invalidations are sent out. At the same time, Thread 1 issues reads for B and C, which by construction load the non-speculative values of the variables. When Thread 0’s chunk commits, signature W_0 is sent 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 we find that $W_0 \cap R_1$ is not null, the chunk in Thread 1 is squashed.

The commit of chunks is globally serialized. 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, the Bulk Multicore supports Sequential Consistency (SC) [9] at the chunk level. As a consequence, the machine

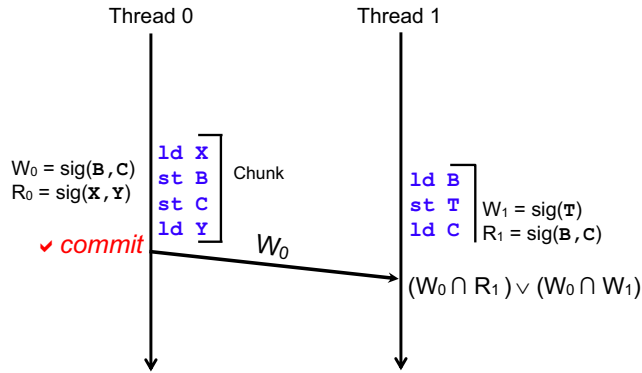


Figure 2: Executing chunks atomically and in isolation with signatures.

also supports SC at the instruction level. Importantly, it supports *high performance SC* at a *low hardware complexity*.

The performance of this SC implementation is high because the Bulk Multicore allows any reordering and overlapping of memory accesses within a chunk. As we will see in Section 4, synchronization instructions induce no reordering constraint within a chunk.

At the same time, the 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 a load to line L is executed out of order until it reaches its commit time, the hardware has to check for writes to L by other processors — lest an inconsistent state was observed. Such checks typically involve sending, for each external coherence event, a signal up the cache hierarchy. The signal snoops the load queue to check for an address match. The checks also involve preventing cache displacements that could risk missing a coherence event. Consequently, load queues, L1 caches and other critical processor components need to be augmented with extra hardware.

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

Finally, note that the Bulk Multicore’s execution mode is not like transactional memory [5]. While one could intuitively think of the Bulk Multicore as an environment with transactions all the time, the key difference is that chunks are *dynamic* entities rather than static, and they are invisible to the software.

In the rest of the paper, we describe how this execution fabric provides three advantages: high programmability, high performance, and hardware simplicity.

3 High Programmability

Since chunked execution is invisible to the software, it places no restriction on programming model, language, or run time system. However, it enables a highly programmable environment by virtue of providing two features: (1) high-

performance SC at the hardware level, and (2) several novel hardware primitives that can be used to build a sophisticated program development and debugging environment. We consider each feature in turn.

The Bulk Multicore’s support for high-performance SC at the hardware level, if combined with SC support in the software stack, will enable a high-performance, fully SC platform. Such a platform is unavailable today due to performance and/or hardware complexity reasons. However, there are a few good reasons why such a platform would be highly programmable. The first one is that SC is the natural way in which programmers think about the interleaving of memory accesses across threads. Secondly, many existing software correctness tools implicitly assume SC — for example, Microsoft’s CHES [13]. Non-SC platforms may produce memory reference interleavings that are impossible to reproduce with the correctness tool. In the next few years, we expect that correctness verification tools will play an increasingly larger role, as more parallel software is developed. Using them in combination with an SC platform would make them most effective.

A third reason for the programmability of an SC platform is that debugging concurrent programs is easier. Indeed, subtle data races can be hard to debug under relaxed memory models. A final, important reason is that supporting SC also simplifies the use of safe languages such as Java, in that the semantics for data races are clear.

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 “on” even during production runs. The key insight is that chunks and signatures free the development and debugging tools from having to record or be concerned with individual loads and stores. This can reduce the amount of bookkeeping and state required substantially and, consequently, eliminate most of the time overhead. In the following, we give three examples of this fact, which are related to deterministic replay of parallel programs, data-race detection, and high-speed disambiguation of sets of addresses.

Finally, we also note that chunks provide an excellent primitive to support popular atomic-section based techniques for programmability such as Thread-Level Speculation (TLS) [16] and Transactional Memory (TM) [5]. We will not address this issue further.

3.1 Deterministic Replay of Parallel Programs *with Practically No Log*

A promising technique for debugging parallel programs is hardware-assisted deterministic replay of parallel programs. This technique involves a two-step process [19]. During the *Recording* step, the parallel program executes while special hardware records into a log the order of data dependences observed between the multiple threads. Effectively, the log captures the “interleaving” of the program’s threads. Then, during the *Replay* step, the parallel program is re-executed, while the system enforces the interleaving orders encoded in the log.

In current proposals, the log stores individual data dependences between threads or groups of dependences bundled together. In the Bulk Multicore, the log only needs to store the total order of chunk commits. We call this approach *DeLorean* [12]. The logged information can be as minimalist as a list of committing Processor-IDs — assuming that the chunking is performed in a deterministic manner and, therefore, the chunk sizes can be deterministically reproduced on replay. We call this design *OrderOnly*. This design reduces the log size by nearly one order of magnitude over current

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 between the different threads — e.g., by committing one chunk from each processor round-robin. In this case, the log practically disappears. During the Replay step, the arbiter re-enforces the original commit algorithm, therefore forcing the same total order of chunk commits as in the Recording step. This design typically has a performance cost because it may force some processors to wait during Recording. We call this design *PicoLog*.

Figure 3(a) shows a parallel execution where the boxes are chunks and the arrows are cross-thread data dependences. Figures 3(b) and (c) show the resulting execution log in the *OrderOnly* and *PicoLog* designs, respectively.

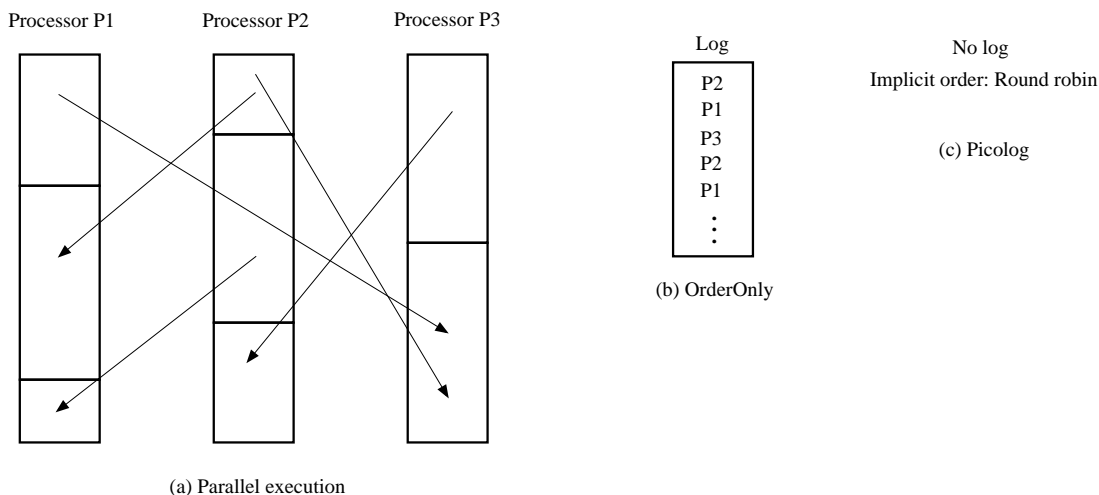


Figure 3: When the parallel execution in (a) is recorded, the Bulk Multicore under *OrderOnly* and *PicoLog* produces the execution logs in (b) and (c), respectively.

3.2 Data Race Detection at *Production-Run Speed*

We can build an efficient data race detector based on the “happens-before” method [8] if we cut the chunks at synchronization points, rather than at arbitrary dynamic points. Synchronization points can be easily recognized by hardware or software, since synchronization operations are executed by special instructions. This approach is described in ReEnact [15]. Figure 4 shows examples with a lock, a flag, and a barrier.

Each chunk is given a 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 4(a), the chunk in Thread 2 that follows 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, we follow a similar procedure. For example, for the barrier in Figure 4(c), 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 have given a partial ordering to the chunks. For example, in Figure 4(a), Chunks 1 and 6 are ordered, but Chunks 3 and 4 are not. Such ordering will help detect data races that occur in a particular execution.

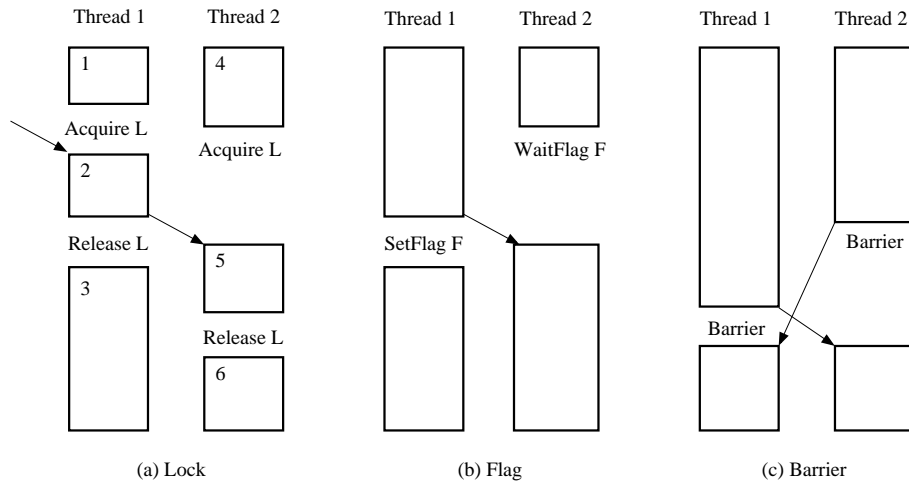


Figure 4: Forming chunks for data race detection in the presence of a lock (a), flag (b), and barrier (c).

Specifically, when two chunks from different threads are found to have a data dependence at run time, their two ChunkIDs are compared. If the ChunkIDs are ordered, then 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 find when two chunks have a data dependence is to use the Bulk Multicore signatures, which can 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 thanks to the hardware support. Consequently, the Bulk Multicore can detect data races without significantly slowing down the program, which makes it ideal for debugging production runs.

3.3 Enhancing Programmability Further: Making Signatures Visible to Software

Finally, a technique to improve programmability further is to make some additional signatures visible to the software. This support enables very inexpensive monitoring of memory accesses, as well as novel compiler optimizations that require dynamic disambiguation of sets of addresses. Side Bar 2 describes this technique and what it can provide.

4 High Performance and Reduced Hardware Complexity

The Bulk Multicore also has advantages in performance and, arguably, in hardware simplicity. It delivers high performance because the processor hardware (and the software) 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 induce no reordering constraint. Indeed, fences inside a chunk are transformed into noops. Their traditional functionality of delaying execution until certain references are performed is useless. This is because, by construction, no other processor will observe the actual order of instruction execution within a chunk.

Moreover, a processor can have multiple concurrently-executing chunks, and memory accesses from these chunks can *also* overlap. Each of the concurrently-executing chunks 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 and, 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.

The Bulk Multicore needs simpler processor hardware than current systems. As indicated in Section 2, the reason is that 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. More details are given in [2]. As an example, this property enables a new environment where cores and accelerators can be designed without worrying about satisfying any particular set of access ordering constraints. This gives a lot of freedom to the hardware designers, allowing them to focus on the novel aspects of their design, rather than on the interaction with the legacy memory consistency model of the target machine. It also motivates the development of commodity processors and accelerators.

5 Related Work

There have been numerous proposals for multiprocessor architecture designs that focus on improving programmability. In particular, architectures for Thread-Level Speculation (TLS) [16] and Transactional Memory (TM) [5] have received significant attention in the past 15 years. These two techniques share some key primitive mechanisms with the Bulk Multicore, namely speculative state buffering and undo, and detection of cross-thread conflicts. However, they have a different goal, namely to 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 goal of the Bulk Multicore is to provide a broadly-usable architectural platform that is easier to program for, while delivering advantages in performance and hardware simplicity.

There are two architecture proposals where processors continuously execute blocks of instructions atomically and in isolation. One of them is Transactional Memory Coherence and Consistency (TCC) [4], a TM environment with transactions all the time. TCC mainly differs from the Bulk Multicore in that TCC’s transactions are statically specified in the code, while chunks are created dynamically by the hardware and are invisible to the software. The second proposal is Implicit Transactions [18], a multiprocessor environment with checkpointed processors. These processors 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) [6] is a programming model where 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 presented here has been adapted to uses in TM, such as transaction footprint collection and address disambiguation [11, 20].

There have been several proposals that implement data-race detection, deterministic replay of multiprocessor programs, and other debugging techniques discussed here without operating in chunks (e.g., [3, 10, 14, 19]). A comparison of their operation to chunk operation is the subject of future work.

6 Future Directions

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

We are now exploring several issues. 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 with many discussions, seminars, and brainstorming sessions. This work has been supported by NSF, DARPA, DOE, Intel and Microsoft under the Universal Parallel Computing Research Center, and gifts from IBM and Sun Microsystems.

References

- [1] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, “Bulk Disambiguation of Speculative Threads in Multiprocessors,” in *International Symposium on Computer Architecture*, June 2006.
- [2] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas, “BulkSC: Bulk Enforcement of Sequential Consistency,” in *International Symposium on Computer Architecture*, June 2007.
- [3] J. Choi, K. Lee, A. Loginov, R. O’Callahan, V. Sarkar, and M. Sridharan, “Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs,” in *Programming Language Design and Implementation*, 2002.
- [4] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, “Transactional Memory Coherence and Consistency,” in *International Symposium on Computer Architecture*, June 2004.
- [5] M. Herlihy and J. E. B. Moss, “Transactional Memory: Architectural Support for Lock-free Data Structures,” in *International Symposium on Computer Architecture*, June 1993.
- [6] M. Isard and A. Birrell, “Automatic Mutual Exclusion,” in *Workshop on Hot Topics in Operating Systems*, May 2007.
- [7] D. Kuck, “Facing up to Software’s Greatest Challenge: Practical Parallel Processing,” *Computers in Physics*, vol. 11, no. 3, 1997.

- [8] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, July 1978.
- [9] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Trans. Comp.*, vol. C-28, no. 9, 1979.
- [10] S. Lu, J. Tucek, F. Qin, and Y. Zhou, "AVIO: Detecting Atomicity Violations via Access Interleaving Invariants," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.
- [11] C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees," in *International Symposium on Computer Architecture*, June 2007.
- [12] P. Montesinos, L. Ceze, and J. Torrellas, "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Efficiently," in *International Symposium on Computer Architecture*, June 2008.
- [13] M. Musuvathi and S. Qadeer, "Fair Stateless Model Checking," in *Programming Language Design and Implementation*, June 2008.
- [14] S. Narayanasamy, C. Pereira, and B. Calder, "Recording Shared Memory Dependencies Using Strata," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2006.
- [15] M. Prvulovic and J. Torrellas, "ReEnact: Using Thread-Level Speculation Mechanisms to Debug Data Races in Multithreaded Codes," in *International Symposium on Computer Architecture*, June 2003.
- [16] G. Sohi, S. Breach, and T. Vijayakumar, "Multiscalar Processors," in *International Symposium on Computer Architecture*, June 1995.
- [17] J. Tuck, W. Ahn, L. Ceze, and J. Torrellas, "SoftSig: Software-Exposed Hardware Signatures for Code Analysis and Optimization," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.
- [18] E. Vallejo, M. Galluzzi, A. Cristal, F. Vallejo, R. Beivide, P. Stenstrom, J. E. Smith, and M. Valero, "Implementing Kilo-Instruction Multiprocessors," in *International Conference on Pervasive Systems*, July 2005.
- [19] M. Xu, R. Bodik, and M. D. Hill, "A "Flight Data Recorder" for Enabling Full-System Multiprocessor Deterministic Replay," in *International Symposium on Computer Architecture*, June 2003.
- [20] L. Yen, J. Bobba, M. Marty, K. Moore, H. Volos, M. Hill, M. Swift, and D. Wood, "LogTM-SE: Decoupling Hardware Transactional Memory from Caches," in *International Symposium on High Performance Computer Architecture*, February 2007.

Side Bar 1: Signatures and Signature Operations in Hardware

Figure 1 showed a simple implementation of a signature. In the figure, 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 possible.

A module called the Bulk Disambiguation Module (BDM) contains several signature registers and simple Functional Units (FUs) that operate on signatures efficiently. These FUs are invisible to the Instruction Set Architecture (ISA). Note that, given a signature, we can only recover a *superset* of the addresses that were originally encoded into that signature. Consequently, the operations on signatures will produce conservative results.

Figure 5 shows five of these operations: intersection, union, test for null signature, test for address membership, and decoding (δ). Intersection finds the addresses common to two signatures. It is done 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 is all zeros. Union finds all the addresses present in at least one signature; it is done through a bit-wise OR of the two signatures. Testing whether an address a is (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.

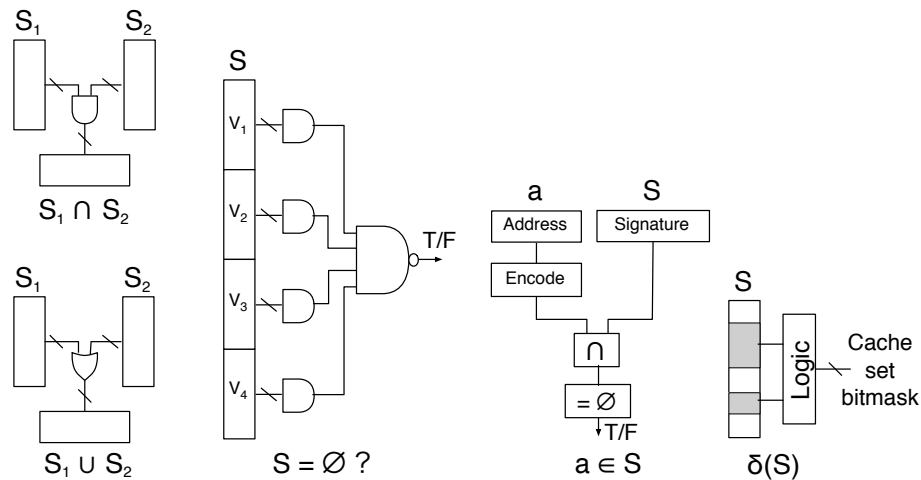


Figure 5: Operations on signatures.

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

Overall, the support described enables low-overhead operations on sets of addresses. More details can be found in [1].

Side Bar 2: Making Signatures Visible to Software

We propose that the software interact with some additional signatures through three main primitives [17]. The first one is to explicitly encode into a signature either one address (Figure 6(a)) or all the addresses accessed in a code region

(Figure 6(b)). The latter case is enabled by the *bcollect* (begin collect) and *ecollect* (end collect) instructions, which can be set to collect only reads, only writes, or both.

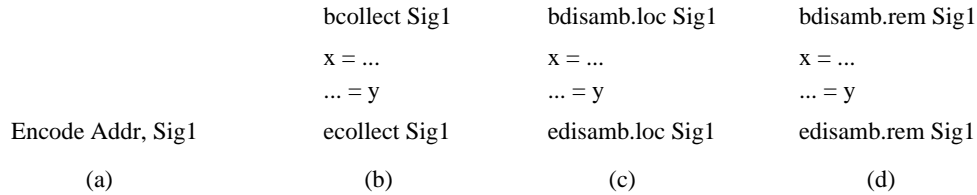


Figure 6: Primitives that enable software to interact with some additional signatures: collection (a and b), local disambiguation (c), and remote disambiguation (d).

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 6(c)), and can disambiguate reads, writes, or both. Finally, 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 6(d)), and can disambiguate reads, writes, or both. When disambiguation finds a match, the system can deliver an interrupt or set a bit.

Figure 7 shows three examples of what can be done with these primitives. Figure 7(a) shows how we support many watchpoints inexpensively. The processor encodes into signature *Sig2* the address of variable *y* and all the addresses accessed in function *foo()*. Then, it watches all these addresses by executing *bdisamb.loc* on *Sig2*.

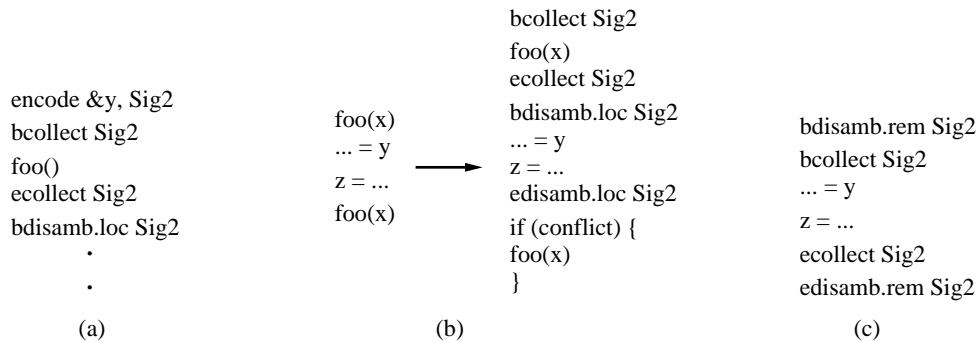


Figure 7: Using signatures to support data watchpoints (a), memoize functions (b), and detect data dependences between threads running on different processors (c).

Figure 7(b) shows how a second call to a function that reads and writes memory in its body can be skipped. In the example, 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 the addresses accessed by *foo()* in *Sig2*. Then, it disambiguates all subsequent accesses against *Sig2*. When we reach the second call to *foo()*, we can skip the call if two conditions hold. The first one is that the disambiguation did not find any conflict. The second — not shown in the figure — is that the read and write footprints of the first *foo()* call do not overlap. This 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 7(c) shows a way to detect data dependences between threads running on different processors. In the

figure, *collect* encodes all the addresses accessed in a code section into *Sig2*. Around the *collect* instructions, we also put *disamb.rem* instructions, which monitor if any remotely-initiated coherence action conflicts with addresses accessed locally. If we want to disregard read-read conflicts, we can collect the reads in a separate signature and only perform remote disambiguation of writes against it.