This paper appears in the 2013 International Symposium on Computer Architecture (ISCA)
Tel Aviv, Israel, June 2013

Exploring Memory Consistency for Massively-Threaded Throughput-Oriented Processors

Blake A. Hechtman and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University
Durham, NC, USA
bah13@duke.edu, sorin@ee.duke.edu

ABSTRACT

We re-visit the issue of hardware consistency models in the new context of massively-threaded throughput-oriented processors (MTTOPs). A prominent example of an MTTOP is a GPGPU, but other examples include Intel's MIC architecture and some recent academic designs. MTTOPs differ from CPUs in many significant ways, including their ability to tolerate latency, their memory system organization, and the characteristics of the software they run. We compare implementations of various hardware consistency models for MTTOPs in terms of performance, energy-efficiency, hardware complexity, and programmability. Our results show that the choice of hardware consistency model has a surprisingly minimal impact on performance and thus the decision should be based on hardware complexity, energy-efficiency, and programmability. For many MTTOPs, it is likely that even a simple implementation of sequential consistency is attractive.

1. INTRODUCTION

Today's general purpose (CPU) multicore processors all precisely specify their hardware memory consistency models (e.g., x86 [29], IBM Power, SPARC TSO [36], etc.). These memory consistency models guarantee the specified orderings of reads and writes by cores [1][33], and differences between consistency models can have significant impacts on programmability, performance, and hardware implementation complexity. The choice of a consistency model is thus quite important and, in the 1990s, there was a significant body of influential research delving into the trade-offs between hardware consistency models for multiprocessors consisting of multiple CPU cores [15][13][2].

In addition to CPU multicores, a new system model is emerging. These systems are massively-threaded throughput-oriented processors (MTTOPs—pronounced "empty tops"). The most prominent examples of MTTOPs are GPGPUs, but MTTOPs also include Intel's Many Integrated Core (MIC) architecture [28] and academic accelerator designs such as Rigel [16] and vector-thread architectures [18]. The key features that distinguish MTTOPs from CPU multicores are: a very large number of cores, relatively simple core pipelines, hardware support for a large number of threads per core, and support for efficient data-parallel execution using the SIMT execution model. The generalized MTTOP that we evaluate in this paper also provides cache-coherent shared memory, because we believe the trend is in that direction. Some current MTTOPs, such as MIC and ARM's Mali GPU [34], already provide cache-coherent shared memory or shared memory without cache coherence, like AMD's Heterogeneous System Architecture [35]. Recent academic research has also explored cache coherence for GPUs [30] and CPU/GPU chips [14].

Given that MTTOPs differ from multicore CPUs in significant ways and that they tend to run different kinds of workloads, we believe it is time to re-visit the issue of hardware memory consistency models for MTTOPs. It is not clear how these differences affect the trade-offs between consistency models, although one might expect that the extraordinary amount of concurrency in MTTOPs would make the choice of consistency model crucial. It is widely believed that the most prominent MTTOPs, GPUs, provide only very weak ordering guarantees[1], and conventional wisdom is that weak ordering is most appropriate for GPUs. We largely agree with this conventional wisdom—but only insofar as it applies to graphics applications.

For GPGPU computing and MTTOP computing, in general, the appropriate choice of hardware consistency model is less clear. Even though current HLLs for GPGPUs provide very weak ordering, that does not imply that weakly ordered hardware is desirable. Recall that many HLLs for CPUs have weak memory models (e.g., C++ [6], Java [22]), yet that does not imply that all CPU memory models should be similarly weak [15].

In this paper, we compare various hardware consistency models for MTTOPs in terms of performance, energy-efficiency, hardware complexity, and programmability. Perhaps surprisingly, we show that hardware consistency models have little impact on the performance of our MTTOP system model running MTTOP workloads. The MTTOP can be strongly ordered and often incur only negligible performance loss compared to weaker consistency models. Furthermore, stronger models enable simpler and more energy-efficient hardware implementations and are likely easier for programmers to reason about.

In this paper, we make three primary contributions:

[1] Public documentation of GPU hardware is limited, especially because the hardware ISAs are not public.

- We discuss the issues involved in implementing hardware consistency models for MTTOPs.
- We explore the trade-offs between hardware consistency models for MTTOPs.
- We experimentally demonstrate that the choice of consistency model often has negligible impact on performance.

The rest of the paper is as follows. In Section 2, we present the memory systems of some current MTTOPs, in order to understand the current state of the art. In Section 3, we provide a brief tutorial on hardware consistency models for CPUs; our MTTOP consistency models are natural extensions of existing CPU models. In Section 4, we present our MTTOP system model. In Section 5, we discuss the numerous reasons for re-visiting the issue of hardware consistency models in the context of MTTOPs. In Section 6, we present implementations of several MTTOP hardware consistency models, and we experimentally evaluate them in Section 7. In Section 8, we re-visit the programmability issue. In Section 9, we discuss some limitations of our analysis, including the impact of assumptions we make. Lastly, in Section 10, we compare this paper to prior work.

2. Current MTTOP Memory Systems

We now present the state of the art in MTTOP memory systems.

2.1. GPGPU Memory Systems

Because no vendor has published a hardware consistency model for a GPGPU, we describe the memory system implementations and infer—to the best of our ability given public documentation—what consistency guarantees they provide. We present two GPGPUs, focusing on hardware designed to be general-purpose rather than on hardware focused solely on graphics. We omit a discussion of Intel's HD Graphics, because we cannot find sufficient publicly available documentation.

In Figure 1, we illustrate a GPGPU core in the context of a complete GPGPU, and we show the commercial names used by AMD and Nvidia. GPU terminology is inconsistent, particularly across commercial products, and we prefer to use more generic terms. In particular, we use the terminology of a GPGPU "core", rather than terminology specific to a commercial product. A GPGPU core is a single core pipeline, and multiple cores are grouped together to form "core clusters" that execute in a SIMT fashion. A core cluster often shares Fetch and Decode stages. Commercial designs often group core clusters into larger groups, but this level of hierarchy is irrelevant to this paper. The generic MTTOP design we evaluate later in the paper has this same hierarchical structure with simple cores grouped into core clusters that execute in a SIMT fashion.

We also unify our terminology for common GPGPU (and MTTOP) synchronization instructions. We use "Fence" (often known as a "Memory Barrier") to denote an instruction that is used by a programmer to enforce ordering between memory instructions (loads, stores, atomic read-modify-writes) in a given thread. We use "Barrier" to denote an instruction that synchronizes the execution of a subset of threads, such that all threads in the subset must reach the Barrier instruction before any of them may resume execution beyond the Barrier.

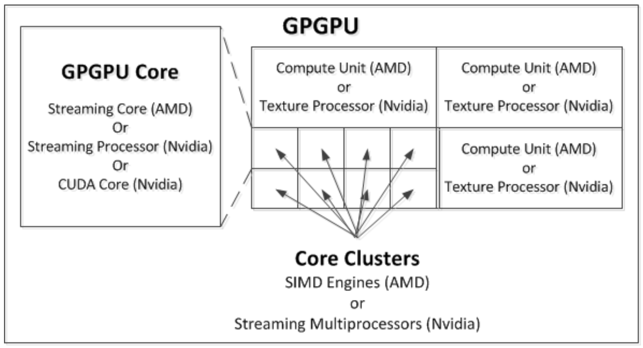

Figure 1. GPGPU Hierarchy and Terminology

2.1.1 AMD's Southern Islands

The AMD Southern Islands GPU[2] provides a single virtual address space that is partitioned into three types of memory: global memory, local data store (LDS), and global data store (GDS). Global memory consists of write-through L1 and L2 caches at each GPGPU core. The LDS is a single structure shared by an entire core. The GDS is a single structure shared by the entire GPU.

Because there is one LDS per "work group" (or "thread block"), the GPU is coherent and provides strong ordering (what we later explain to be sequential consistency in Section 3.1) for accesses within a work group. Similarly, the GDS is coherent and provides strong ordering for all threads. The global memory is not coherent and its ordering is less strict, by default, but there are hooks for software to add coherence and ordering when desired. Specifically, when software performs synchronization instructions—Fences and Barriers—the dirty data in the L1 is forced back to lower levels of the memory system, where it becomes visible. Synchronization instructions, after draining the L1, then invalidate all L1 blocks. There is not yet a formal, implementation-independent memory consistency model for global memory.

2.1.2 Nvidia's Fermi

The Nvidia Fermi GPGPU[3] provides a unified address space that is divided into local, shared, and global sub-spaces. Each core can access an L1 storage structure that is partitioned between L1 cache and a scratchpad memory that is shared by all threads on the local group of cores. The L1 cache is backed by an L2 cache that is shared by all threads on all cores.

Fermi provides limited hooks for performing synchronization and ordering memory operations. The primary mechanism is a Fence instruction. When a thread reaches a Fence instruction, the Fence guarantees that all prior (in program order) writes by that thread are propagated to the L2 cache or memory and the entire L1 cache is invalidated before the thread is able to continue. Using these hooks, a programmer can synchronize access to shared data, e.g., prohibit concurrent reads and writes to the same address.

The limited publicly available documentation describes the hardware's behavior, but we have not discovered any

[2] http://developer.amd.com/afds/assets/presentations/2620_final.pdf
[3]http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

documentation of an implementation-independent hardware memory model.

2.2. Intel's Many Integrated Core (MIC)

Intel's MIC architecture [28] features dozens of multithreaded and otherwise relatively simple cores, and its goal is throughput rather than single-thread performance. The memory system provides cache-coherent shared memory, and the coherence protocol operates over a ring-based interconnection network.

As an Intel product based on the x86 architecture, MIC satisfies Intel's longstanding x86/TSO consistency model [25][29][28]. This model, which we describe in detail in Section 3.2, is a CPU model that was developed to enable the use of FIFO write buffers to hide the latency of stores. This consistency model predates MIC by many years and was thus not designed for MTTOP hardware or software.

3. HARDWARE CONSISTENCY FOR CPUs

This section is intended as a brief review of hardware memory consistency models for shared memory chips with general purpose cores (CPUs). We refer readers seeking more depth to tutorials on this topic [1][33].

3.1. Sequential Consistency (SC)

The memory consistency model specifies the legal orderings of loads and stores, by all threads, to shared memory. The simplest model, sequential consistency [20], specifies that there must appear to be a total order of all loads and stores, across all threads, that respects the program order at each thread. Each load obtains the value of the most recent store to the same address in the total order. Sequential consistency (SC) is the most restrictive model in that it permits no reordering of loads or stores with respect to per-thread program orders.

3.2. Total Store Order (TSO) / x86

Consistency models other than SC permit more reorderings and thus may enable greater performance. A well-known example is the total store order (TSO) model used in the SPARC architecture [36] and later formalized to encompass the x86 model as x86-TSO [25]. TSO is similar to SC except it does not enforce order from a thread's store to a subsequent (in program order) load to a different address. This reordering may seem somewhat arbitrary, but it crucially enables the use of a FIFO write buffer for committed stores. This write buffer enables the core to commit a store without waiting for the store to access the cache, thus freeing the core from maintaining state about the store and enabling the core to proceed. If software needs order between a store and a subsequent load, the software—programmer or compiler—must insert a Fence instruction between them. A Fence guarantees that all instructions prior (in program order) to the Fence are visible to other threads before the Fence completes, and it guarantees that none of the instructions after the Fence are visible to other threads until the Fence completes.

3.3. Relaxed Memory Order (RMO)

Some consistency models are even more relaxed than TSO, and they all have their own particular quirks. Examples include SPARC RMO [36], Alpha [31], and the generalized tutorial example XC [33]. In the rest of this paper, we will use RMO as our exemplar of this class of models. In general, these models enforce no ordering between loads and stores to different addresses. If software needs ordering, it must use Fence instructions to obtain it. The motivation for these relaxed models is performance; with the bare minimum of ordering, these models should enable the most optimizations and thus the best performance. For example, a system with RMO permits the use of an unordered, coalescing write buffer between each core and its data cache; this hardware optimization would lead to executions that violate SC and TSO.

3.4. SC for Data Race Free

Another way proposed to implement strong consistency on CPUs is *Sequential Consistency for Data-race Free* (SC for DRF) programs [2]. The idea is that a program without data races behaves in a sequentially consistent fashion even on weakly ordered hardware. That is, a programmer who writes DRF code can reason about SC, even if the underlying hardware consistency model is weaker than SC. SC for DRF relies on the programmer or compiler to label synchronization variables as well as the data it protects.

All known CPU consistency models provide SC for DRF, and we believe that SC for DRF may be appropriate for MTTOPs. The memory systems of many current MTTOPs, including GPGPUs, provide sufficient architectural support for writing DRF code. They have Fence instructions that guarantee that all prior memory operations are complete, and they have atomic operations that are globally performed to enable synchronization. If an MTTOP provides these architectural mechanisms and supports a consistency model like the models described in this section, then SC for DRF is a possibility.

3.5. The Debate

The intense debates about the relative merits of consistency models ultimately boiled down to programmability versus performance. (These debates were in the era before power and energy were major issues.) The general consensus was that stronger models like SC or TSO/x86 are easier for programming, and the question was how much performance was sacrificed by using strongly ordered models. Researchers showed that there is indeed a non-trivial gap in performance—ranging from around 10% to 40%, depending heavily on system models and how the implementations support the consistency models—but that the performance gaps can often be narrowed (e.g., to 10-15%) by using speculation to accelerate the stronger models [13][27].

Our goal in this paper is to re-visit the issue of hardware consistency models, but in the new context of MTTOPs. MTTOPs differ from CPUs in many significant ways, including their ability to tolerate latency and their memory system organization.

4. MTTOP SYSTEM MODEL

In this paper, we seek to provide a very general model of an MTTOP that retains just those features that fundamentally distinguish MTTOPs from CPUs without concerning ourselves with features from today's MTTOPs that are less fundamental. In particular, we do not model the quirks of today's GPGPUs; although GPGPUs are trending towards even greater generality and fewer GPU-specific constraints, they still retain many unconventional features.

The key features of our MTTOP model are: a very large number of cores, relatively simple core pipelines, hardware support for a large number of threads per core, and support for efficient data-parallel execution using the SIMT execution model. This model shares some but not all features with all of today's MTTOPs.

We assume an MTTOP with many cores, and cores are grouped into 8-core clusters. Each core is a 1-wide, in-order, non-speculative pipeline that can support 64 thread contexts. All 8 cores in a core cluster share a single Fetch pipeline stage and Decode pipeline stage, and they execute in SIMT fashion. A core switches out a thread when the thread issues a load, store, or Fence that is not immediately satisfied. All 8 cores in a core cluster share an L1 instruction cache and a small writeback L1 data cache. Each core can have at most one outstanding load miss.

Our MTTOP's memory system provides cache-coherent shared memory for all cores. This feature is perhaps the most significant departure from most of today's MTTOPs, although recent academic research [30][14] and commercial chips indicate a trend towards coherence. Intel's MIC provides cache-coherent shared memory. ARM has presented the Mali GPU with coherence as well as a CPU/GPU chip with coherence [34]. AMD's Heterogeneous System Architecture (HSA) [35] provides shared virtual memory and virtual-to-physical address translation for CPU/GPU chips. Nvidia GPUs provide global memory and Unified Virtual Addressing via CUDA. Moreover, the future of MTTOPs depends on building up a large software ecosystem, and doing so is extremely difficult for platforms that do not provide hardware coherence. If software must manage the caches, then software interoperability and composability becomes far more complicated. Furthermore, concerns about coherence not scaling have been addressed by some recent work that shows that coherence can scale far better than many had thought [23][11], permitting it to be used in MTTOPs with hundreds and perhaps even thousands of caches.

5. WHY REVISIT MEMORY CONSISTENCY?

We seek to compare hardware consistency models for MTTOPs. MTTOPs change the trade-offs for a memory consistency model compared with the trade-offs architects consider for CPUs, for a variety of reasons. We focus on eight particularly relevant and significant differences between MTTOPs and CPUs.

5.1. Outstanding Cache Misses per Thread → Potential Memory Level Parallelism

MTTOPs consist of simple, in-order pipelines, unlike the more sophisticated out-of-order pipelines that are common in CPUs. MTTOPs are also less likely than CPUs to benefit from prefetching into L1 caches. Thus, a thread running on an MTTOP is unlikely to have a large number of outstanding requests. In contrast, a CPU seeks to maximize ILP and memory level parallelism (MLP) per thread, and one of the keys to single-threaded performance on a CPU is issuing memory requests in parallel [10]. CPUs execute instructions out of order so as to reach memory accesses sooner and overlap them with already-outstanding misses. CPUs also prefetch extensively. Thus, a thread running on a CPU may often have many concurrent outstanding misses, whereas MTTOPs likely have one or perhaps a few outstanding misses per thread. MTTOPs overcome the limited MLP per thread by supporting vastly more threads per core, thus enabling many outstanding requests per core.

<u>Impact on consistency model:</u> Some CPU consistency models and their implementations are designed to optimize each thread's MLP. Such optimizations for per-thread MLP are likely to be less beneficial for MTTOPs.

5.2. Threads per Core → Latency Tolerance

An MTTOP core supports dozens of threads. In contrast, a typical CPU core supports only 1-8 thread contexts. Thus, an MTTOP core can—if the software offers enough threads—tolerate large memory access latencies by quickly switching from a thread waiting on memory to other threads that are ready. An MTTOP with sufficient threads to run places huge bandwidth demands on a memory system, because of all the threads it supports, but it tolerates far greater latencies than a CPU. As one indicator of this latency tolerance, many current MTTOPs (GPGPUs) have cache access latencies that are far greater than those of current CPUs.

<u>Impact on consistency model:</u> Some CPU consistency models and their implementations are designed to optimize latency. Such optimizations for latency are likely to be less beneficial for MTTOPs.

5.3. Threads per System → Synchronization and Contention for Shared Data

Typical multicore CPU chips support on the order of a few dozen threads, whereas MTTOPs support hundreds or thousands of threads. Synchronizing vastly more threads and sharing data among them is qualitatively more challenging. MTTOP programmers are likely to avoid the coarse locks and barriers that are often acceptable for CPU programmers, because of the extraordinary contention. A coarse lock or barrier in a CPU workload may incur some contention, but the contention will be less than for a MTTOP and it may even be possible for a CPU to speculatively elide a coarse lock at runtime, as was proposed by Rajwar and Goodman [26] and later implemented in Intel's Hardware Lock Elision (HLE) for the recent Haswell chip [7].

Figure 2 shows a parallel sum of a matrix where SMALL is less than one hundred and BIG is in the thousands. On a CPU, the maximum parallelism would come from parallelizing along the rows (SMALL). Figure 3 shows an equivalent implementation that parallelizes along BIG for an MTTOP. For this to work, the MTTOP must perform a parallel reduction that requires frequent synchronization to avoid data races. Although this code is somewhat simplistic and this example may seem contrived, nearly identical code is executed multiple times per iteration of a Kmeans problem ported to an MTTOP. The take-away point is that synchronization is likely to be far more frequent in MTTOP code.

<u>Impact on consistency model:</u> Some CPU consistency models and their implementations rely on the fact that lock and barrier contention is unlikely. For example, some aggressive implementations of SC allow a load to speculatively execute as soon as its address has been computed and then re-execute at Commit to check that the value did not change in the interim [8]. If contention is rare, such optimizations are useful, but contention is likely to be far more frequent in MTTOPs.

```
int matrix[SMALL][BIG]={....}
int sum[SMALL];
int global_sum=0;


int id=thread_id;//0<=id<SMALL


// Each thread does its sum

int acc=0;
for(int j=0; j<BIG; j++){
   acc+=matrix[id][j]
}
sum[id]=acc;






// Thread 0 does global sum

barrier(SMALL);



if(id==0){
   for(int i=0;i<SMALL;i++){
       global_sum += sum[i];
   }
}
```

Figure 2. Code for multithreaded CPU (e.g., 8-64 threads).
Runtime = O(BIG)

```
void barrier(int width);// synchs threads 0 to width-1
                     // barrier complexity=O(log2(width))

int matrix[SMALL][BIG]={....}
int sum[SMALL];
int global_sum=0;

int tmp[BIG/2];

int id=thread_id; // 0<=id<BIG


// More sophisticated reduction

for(int i=0; i<SMALL; i++){
  //assume BIG=2^k,ignore SIMD effects
  int thread_set=BIG/2;
  tmp[id]=matrix[i][id];

  barrier(BIG);

  while(thread_set > 1){
     if(id<thread_set){
        tmp[id] += tmp[id+thread_set];
     }

     barrier(BIG);

     thread_set = thread_set/2;
  }
  if(id==0){
     int sum_tmp = tmp[0];
     sum[i] = sum_tmp;
     global_sum += sum_tmp;
  }
}
```

Figure 3. Code for MTTOP (e.g., thousands of threads).

Runtime = O(SMALL)*$\log_2$(BIG)*$\log_2$(BIG)

5.4. Threads per System → Opportunities for Reordering

As the number of threads in a program increases (i.e., as there are more threads to perform the same total amount of work), the number of memory operations performed by each thread decreases. Synchronization becomes an increasingly large fraction of the runtime, as we observed in the code in Figure 2 and Figure 3. Because memory operations must complete before synchronizing, in order to prevent data races, there are fewer opportunities to reorder memory operations to achieve memory level parallelism at the thread level.

Impact on consistency model: Consistency models allow memory operations to be reordered only between synchronizing events. Synchronizing events depend on the specific consistency model and include atomic read-modify-write instructions, Fences, and Barriers. With the increased frequency of synchronization in MTTOP workloads, the window in which memory operations can be reordered shrinks to only a few operations. Consistency models that permit more reordering may offer less benefit to MTTOPs.

5.5. Register Spills/Fills → RAW Dependences

In the MTTOP benchmarks we use in this paper, we observe very few register spills and fills. There are several hypotheses for this behavior, but we focus on the two most plausible. First, vectorized (SIMT) code removes many loops that would be required in CPU code and thus reduces the number of live variables. Second, MTTOP cores tend to have register files that are considerably larger than CPU register files. A core in AMD's Southern Islands GPGPU has a 256 KB register file. A CPU core with 256 64-bit registers has, by comparison, only 2 KB of registers. All other things being equal, with larger register files, the likelihood of register spills and fills decreases. All other things are not equal, though, because an MTTOP core shares its register file across many more threads. Nevertheless, MTTOP register files seem to be sized large enough to mostly avoid spills and fills.

Regardless of the cause for fewer spills and fills, this phenomenon reduces the likelihood of read-after-write (RAW) dependences through memory. In fact, in all of the MTTOP benchmarks with which we experiment in this paper, we observe

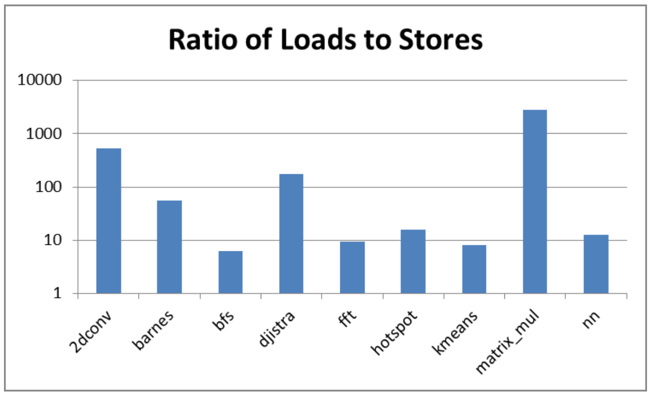

Figure 4. Ratio of loads to stores in MTTOP benchmarks

no RAW dependences through memory without a Fence between the store and subsequent load, except during recursion.

Impact on consistency model: Consistency models that enable the use of write buffers, such as x86/TSO, must consider RAW dependences through the write buffer. When a load's address matches the address of a store in the write buffer, an optimized implementation enables the load to read the value of the store. This optimization is useful for nearby RAW dependences through memory. Given that MTTOPs are less likely to have nearby RAW dependences through memory, this optimization is less likely to be useful for MTTOPs than for CPUs.

5.6. Algorithms → Ratio of Loads to Stores

The nature of MTTOP algorithms leads to a mix of dynamic instructions that tends to vary substantially from the mix observed in CPU workloads. Specifically, we focus on loads and stores. In CPU workloads, such as Parsec [4] and Lonestar [19], there are usually between 1.5-5 loads per store, with each benchmark suite having only one or two outliers with a greater ratio. In MTTOP workloads, we observe (experiments explained in Section 7) the results in Figure 4. All ratios are greater than six, and several MTTOP benchmarks have ratios above twenty, including a maximum ratio of 2700.

The intuition for why MTTOPs have much greater ratios of loads to stores derives from the types of algorithms run on MTTOPs. In Figure 5, we show code for matrix multiplication, which is an embarrassingly parallel algorithm that runs well on MTTOPs. The amount of reading (loads to arrays *A* and *B*) dwarfs the writing (stores to array *C*).

Impact on consistency model: Several consistency models explicitly optimize for stores. TSO/x86 was developed specifically to hide the latency of stores and enable them to commit from a core without waiting for coherence permission to write to the cache. Given the relative rarity of stores in MTTOP workloads, such optimizations are likely to be far less useful for MTTOP workloads.

5.7. Intermediate Assembly Languages

A large class of MTTOPs, GPGPUs, specify an intermediate assembly language (e.g., Nvidia's PTX) instead of a hardware machine language. All CPUs provide a hardware machine language. This difference manifests in two ways. First, GPGPUs must translate at runtime from the intermediate language to the

```
int A[N][N];
int B[N][N];
int C[N][N];  // product

for(int i =0; i<N; i++){
  int tmp=0;
  for(j=0;j<N;j++){
    tmp = tmp + A[thread_id][j]*B[j][i];
  }
  C[tid][i]=tmp;
}
```

Figure 5. Matrix Multiplication Code

machine language in a process sometimes known as "finalizing." The latency of finalizing is an overhead that must be amortized over a reasonably long latency; tasks that are very short are unlikely to provide a benefit. Furthermore, the finalizer may be required to insert Fences and Barriers to ensure proper synchronization, and the complexity of determining where to insert them adds to the latency of finalizing.

The second implication of intermediate assembly languages is that the vast majority of GPGPU programmers are shielded from the hardware in a way that is unusual for CPU programmers. Although it is true that most CPU programmers rely on expert-written libraries for the code that is affected by consistency (e.g., pthreads), application programmers can write directly to the hardware. An example is the use of flag synchronization. Furthermore, HLLs like OpenCL and CUDA specify a weak memory model to the compiler. This allows the compiler to reorder instructions when producing the intermediate assembly code. From the intermediate language, the finalizer generates machine code that can reorder instructions even more. Thus, GPGPU application programmers have virtually zero interaction with the hardware consistency model (or the hardware at all).

Impact on consistency model: The two issues raised by intermediate languages have different impacts on consistency. The latency of finalizing motivates consistency models that are geared towards larger tasks. The shielding of programmers from the hardware makes programmability—as a function of the hardware consistency model—a less important issue for GPGPUs than it is for CPUs.

5.8. Threads per System → Programmability

For CPUs, the conventional wisdom is that programming for stronger consistency models, such as SC or even TSO/x86, is more intuitive and easier than programming for more relaxed consistency models. CPU programmers are likely to assume SC when first learning parallel programming, and they are likely to be surprised by having a thread observe another thread's stores out of program order.

The programming idioms on an MTTOP, however, are different. On an MTTOP, to write code in which a thread reads another thread's store, the producer and the consumer cannot be in the same SIMT thread group (i.e., executing together in a SIMT fashion, as in a GPGPU warp or wavefront), because that situation would cause deadlock. A SIMT thread group executes a single instruction at a time and thus either prevents the consumer from ever observing the store or prevents the producer from performing

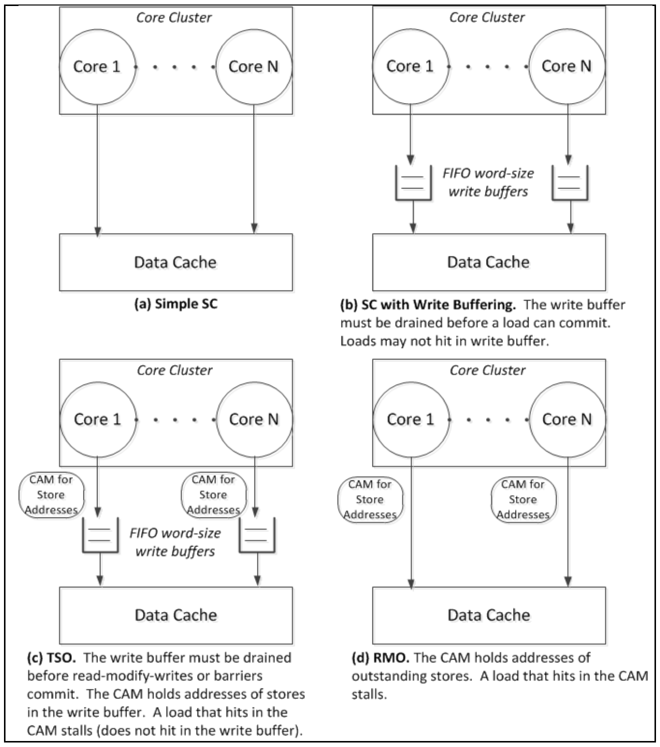
Core Cluster
Core 1
Core N
Data Cache
(a) Simple SC
Core Cluster
Core 1
Core N
FIFO word-size write buffers
Data Cache
(b) SC with Write Buffering. The write buffer must be drained before a load can commit. Loads may not hit in write buffer.
Core Cluster
Core 1
Core N
CAM for Store Addresses
CAM for Store Addresses
FIFO word-size write buffers
Data Cache
(c) TSO. The write buffer must be drained before read-modify-writes or barriers commit. The CAM holds addresses of stores in the write buffer. A load that hits in the CAM stalls (does not hit in the write buffer).
Core Cluster
Core 1
Core N
CAM for Store Addresses
CAM for Store Addresses
Data Cache
(d) RMO. The CAM holds addresses of outstanding stores. A load that hits in the CAM stalls.

Figure 6. Implementations of Consistency Models

the store (because the other threads in the SIMT thread group are busy-waiting). Thus the MTTOP programmer must be able to enter synchronization loops without unsafe thread divergence.

Once the MTTOP programmer is aware of where synchronization is necessary, it is relatively simple to insert synchronization that avoids data races. Once data races are avoided, SC does not buy the programmer any benefit due to the inability to view reorderings. (Recall SC for DRF.) Furthermore, there is little likelihood of MTTOP programmers using flag synchronization—the only type of synchronization that programmers use without libraries and a type of synchronization that depends on the consistency model—for thousands of contending threads. Either there will be thousands of synchronizing variables or thousands of threads contending on a few synchronizing variables; neither scenario is attractive.

Impact on Consistency Model: The issue of programmability is likely to be a less important criterion for choosing a hardware consistency model for MTTOPs than for CPUs.

6. IMPLEMENTING MEMORY MODELS FOR MTTOPs

In this paper, we compare natural extensions of existing CPU consistency models: SC, TSO, and RMO. Because our MTTOP cores issue the same memory operations as CPUs (loads, stores, Fences), we can adopt CPU-style consistency models. We do not argue that consistency models developed for CPUs are optimal, though, and future work will explore MTTOP-specific models.

All of the memory consistency model implementations in this paper assume that there is a directory-style MOESI cache coherence protocol to keep the MTTOP data caches coherent. Recall from Section 4 that our MTTOP model has in-order cores that permit only one outstanding load per thread and thus no load-to-load reordering is possible (even if it were to be permitted by the consistency model).

6.1. Simple SC (SC_{simple})

The simplest memory model implementation we consider is a conservative implementation of SC. It has been shown previously that connecting simple in-order cores directly to a coherent memory system provides SC [24]. We illustrate this implementation, for a single core cluster, in Figure 6a. If a load or store reaches the head of a pipeline, the pipeline stalls until the load or store accesses the cache. This memory system is simple to implement because it requires handling at most one outstanding memory operation per thread. The potential problem with this memory system is performance, because blocking on every load and store could hurt performance.

6.2. SC with Write Buffering (SC_{wb})

We have also developed a more aggressive implementation of SC, illustrated in Figure 6b. We insert a FIFO write buffer between each MTTOP core and its L1 data cache, and this FIFO write buffer must be drained before a subsequent load can commit. (A load may not "hit" in the write buffer.) While waiting for the write buffer to drain, the core may prefetch, but it may not commit memory operations.

6.3. Total Store Order (TSO)

Our TSO implementation, illustrated in Figure 6c, also uses a write-only FIFO write buffer between each MTTOP core and its L1 data cache. The FIFO write buffer must be drained before every atomic instruction and Fence. The write buffer includes a CAM of the store addresses in the write buffer. If a load hits in the CAM, the load does not access the write buffer nor may it obtain a stale value from the cache. This design decision—a write-only write buffer—differs from typical TSO implementations for CPUs, in which a load may hit in the write buffer. We chose this write-only design for its simplicity and energy-efficiency; as explained in Section 5.5, the likelihood of load hits in the write buffer is much smaller for an MTTOP than a CPU. That is, if we had a read port on the write buffer, it would not be used often enough to be worth its cost. In our experiments, we tracked whether a read port on the write buffer would be used, if it existed, and we saw *zero* instances in which this occurred.

6.4. Relaxed Memory Ordering (RMO)

We implement an RMO-like consistency model that allows loads and stores to directly access the cache, with no write buffer. As shown in Figure 6d, we add a CAM that maintains the addresses of stores awaiting coherence permission to write into the cache (i.e., stores in MSHRs). If a load hits in the CAM, it stalls until the matching store updates the cache. As with our TSO implementation, we have foregone an optimization—in this case, allowing a load to hit in an MSHR for an outstanding store—because MTTOP workloads are unlikely to be able to exploit this optimization. A Fence waits for all outstanding stores to update the cache. In this implementation, this entails waiting for all entries in the CAM to be cleared.

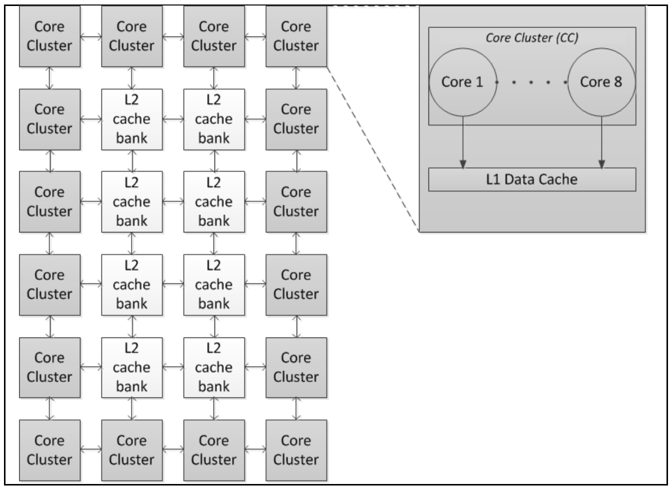

Figure 7. MTTOP System (for SC_{simple} implementation). The L1 instruction caches are omitted.

6.5. Graphics Compatibility

Because the dominant type of MTTOP today is GPGPUs, we must consider the impact of the MTTOP memory consistency model on graphics. It is not necessary for graphics memory accesses to be ordered as in the CPU-style models described above. We expect that an MTTOP that supports both GPGPU computing and graphics would provide different memory models for these two purposes. For graphics, we would want stores to bypass the L1 cache, because the core would never read these values. For loads, which generally stream into the core to be read only once, there is little point in caching the loaded values.

7. EXPERIMENTAL COMPARISON OF MTTOP MEMORY MODELS

In this section, we experimentally compare the consistency model implementations presented in the previous section.

7.1. Simulation Methodology

We simulate our generalized MTTOP model, described in Section 4, using a modified version of the gem5 full-system simulator [5]. The MTTOP configuration is illustrated in Figure 7 (omitting the L1 instruction caches), and its parameters are listed in Table 1. The figure pertains directly to the SC_{simple} implementation; other consistency model implementations would have the features, such as write buffers, shown in Figure 6.

7.2. Benchmarks

We consider a wide range of MTTOP benchmarks, listed in Table 2, some of which we wrote by hand and a subset of the Rodinia GPGPU benchmark suite [9] that we ported. Porting was necessary because our MTTOP model has cache coherent shared memory and Rodinia was written for GPUs without these features. All benchmarks were written in C/C++ and compiled directly to our MTTOP's Alpha-like hardware ISA. Because the Alpha consistency model is very close to that of RMO, Alpha code runs correctly on systems that support RMO. Furthermore, because SC and TSO are stronger consistency models than RMO, code that runs correctly on RMO will also run correctly on systems that support SC and TSO. Put another way, SC and TSO implementations are valid implementations of RMO.

Table 1. MTTOP Configuration

Parameter	Value
core clusters	16 core clusters; each core cluster has 8 cores
core	in-order, Alpha-like ISA, 64 thread contexts
interconnection network	2D torus
clock frequency	1GHz
L1I cache (shared by cluster)	perfect, 1-cycle hit
L1D cache (shared by cluster)	16KB, 4-way, 20-cycle hit
L2 cache (shared by all clusters)	256KB, 8 banks, 8-way, 50-cycle hit
consistency model-specific features	
write buffer (SC_{wb} and TSO)	perfect, instant access
CAM for store address matching	perfect, instant access

7.3. Performance Results

We present performance results, in terms of speedup, in Figure 8. The speedups are with respect to the performance of SC_{simple}. Although there appears to be some variation across the consistency models, the absolute differences are quite small and likely within the "noise" (i.e., not statistically significant given the natural variation in runtime). A few benchmarks, such as kmeans, incur some performance penalty for more relaxed models. These performance penalties, which are also extremely small, are likely due to the latency of frequent Fences for synchronization.

There are two MTTOP configuration choices that could potentially impact these results: L1 data cache size and L1 data cache hit latency. In CPUs, the choices for these two parameters usually have fairly significant impacts on performance, and we wanted to discover whether MTTOPs are as sensitive to these parameters. For our MTTOP, we swept the values of both L1 data cache size and hit latency over a wide range that includes the nominal values in Table 1. The MTTOP *speedup* results for other cache sizes and latencies were virtually indistinguishable from the results in Figure 8, even though the absolute runtimes differed, and thus there is no need to show graphs for either experiment. The lack of sensitivity is striking for architects accustomed to CPU performance studies, but they are more intuitive after considering the differences between CPUs and MTTOPs that we discussed in Section 5.

The takeaway point from these results is that the choice of

Table 2. Benchmarks

Benchmark	Description
Handwritten	
barnes-hut	N-body simulation
matrix mult	matrix multiplication
dijkstra	all-pairs shortest path
2D convolution	2D matrix-to-matrix convolution
fft	fast Fourier transform
Ported from Rodinia [9]	
nn	nearest neighbor
hotspot	processor temperature simulation [32]
kmeans	K-means clustering
bfs	breadth-first search

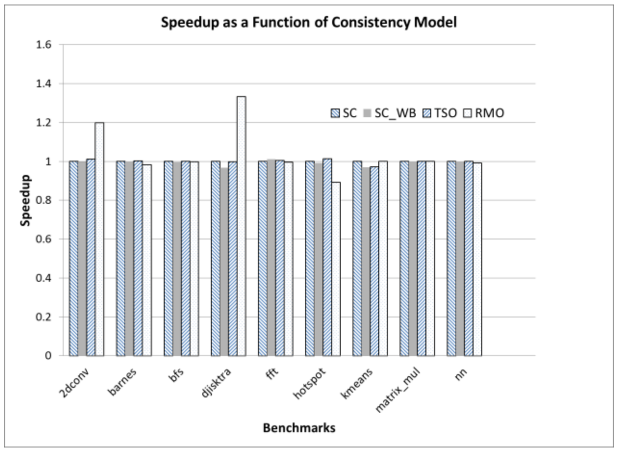

Figure 8. Performance Comparison Across Memory Models

hardware consistency model has little impact on MTTOP performance, despite its importance for CPUs. Thus, we should choose the consistency model based on other issues, particularly implementation complexity and energy-efficiency.

7.4. Implementation Complexity and Energy-Efficiency

Because the studied consistency models all offer comparable performance, we focus on implementation complexity and energy-efficiency. On these issues, SC_{simple} appears to be a clear winner. It has the simplest hardware—just a single MSHR per thread context—and we would thus expect it to use the least energy. SC also offers some programmability advantages, although we have argued earlier that programmability at the hardware level is less critical for MTTOPs than for CPUs. The programmability advantage benefits the writers of compilers, finalizers, and drivers, who are more expert than typical programmers yet still likely to appreciate SC.

7.5. Implication of More Sophisticated Cores

We have assumed simple, in-order MTTOP cores. If deeper and more aggressive pipelines emerge, then a consistency model that enables loads to be reordered may be desirable. Loads are more critical to performance and thus a weaker model like RMO may be desirable. Implementing RMO requires a CAM to compare load addresses to outstanding stores in MSHRs, but the cost of the CAM—or a Bloom filter for the same purpose—may be worthwhile for systems with more aggressive cores.

To test the impact of more sophisticated MTTOP cores, we modified our MTTOP core model to enable load reordering. The RMO results in Figure 8 are for such a core. (The other consistency models disallow load reordering.) When a load issues to the memory system, the core does not stall and may continue to issue subsequent (in program order) loads. If an older load misses and a younger load hits, these loads appear out of order to the memory system. Our experimental results show that, for the majority of our benchmarks, allowing load reordering has little impact on performance. When a load issues to the memory system, it is often the case that a subsequent instruction that depends on that load follows close behind it and prevents subsequent loads from issuing. Effectively, there are few outstanding loads at any given time. However, for the 2D convolution benchmark, we observe a 20% performance improvement. This benchmark, more than the others, has a lot of independent loads for small amounts of data, because it has a small mask and a large matrix.

The impact of load reordering is a function of the compiler, and it is possible that a compiler could be tuned to take greater advantage of load reordering than what we observe in our experiments.

8. FINALIZER PROGRAMMABILITY

We have argued that the programmability criterion is less important for choosing a MTTOP hardware consistency model than for choosing a CPU consistency model. This argument rests largely on how current GPGPU programmers are shielded from the hardware ISA; application programmers write in HLLs and can see only as far down as the intermediate language.

This argument applies to the vast majority of GPGPU programmers, but it omits one small yet important class: people who write the finalizers that translate from the intermediate language to the GPU’s native hardware language. The finalizer has a difficult job in the process of running a GPGPU program. It must allocate physical registers and generate machine code without syntactic knowledge of the original source code. On CPUs, many synchronization libraries rely heavily on inline assembly code, yet GPGPUs have no such luxury. Many intermediate language instructions may have a simple one-to-one mapping to hardware instructions, but some intermediate instructions have synchronization implications (e.g., Fence, Barrier, atomic read-modify-write). It is likely that the intermediate instructions use heavyweight mechanisms to make sure that all stores are visible. If these heavyweight mechanisms thrash data out of the cache, the cache may be rendered useless in code with synchronization.

A strong or at least explicit memory model enables the finalizer writer to formalize what the hardware does in a way that can be used to facilitate optimizations. At the very least, a hardware consistency model makes caches with coherence amenable to code with synchronization. Without a well-specified hardware consistency model, the finalizer must understand the details of the hardware implementation. The finalizer is thus likely to be overly conservative and make worst-case assumptions about the hardware’s behavior. With an explicit hardware memory model, the hardware designers can enforce that model as aggressively or conservatively as chosen, and the finalizer can ignore hardware implementation details. Trying to reason about all of the possible concurrency issues in a GPGPU implementation, with its vast amounts of possibly concurrency, is a challenge that we would like to avoid.

9. CAVEATS AND LIMITATIONS

The analysis we have performed in this paper necessarily makes several assumptions, and our conclusions are at least somewhat dependent on these assumptions. The two primary types of assumptions pertain to our system model and workloads, because it is not feasible to explore all possible system models or workloads.

9.1. System Model

Our results depend on the MTTOP model we described in Section 4. We believe this MTTOP model is representative of future MTTOPs, yet we are aware that perfectly predicting the future is unlikely. We now discuss the implications on our results and conclusions of some possible variations in the MTTOP model.

- Register file size: Our register file is relatively large. A smaller register file could lead to more register spills/fills and thus to more frequent RAW dependences through memory.
- Scratchpad memory: Our MTTOP has no scratchpad memory. Including scratchpads is likely to make the choice of consistency model even less important, because scratchpads would reduce the number of accesses to shared memory. However, it is theoretically possible that the performance of this lesser number of accesses to shared memory would be more critical.
- SIMT width: If our MTTOP cores had a wider SIMT width, then there would likely be more divergence between threads, and such divergence could increase the impact of memory system performance.
- Write-through caches: We have assumed write-back caches, yet current GPUs support write-through caching (which is preferable for graphics). It is possible that write-through caching will persist for many future MTTOPs, although the energy benefits of write-back seem compelling. If write-through caching persists, then the latency of stores becomes more important and consistency models that take store latency off the critical path may be more attractive. However, given the relative rarity of stores, even write-through caching may be insufficient to motivate a weaker model than SC.
- Non-write-atomic memory systems: We have assumed memory systems that provide write atomicity [3]. However, future systems may not provide write atomicity, and we would have to adjust our memory consistency model specifications accordingly. It is unclear if or how such a memory system would impact the performance results or conclusions.

9.2. Workloads

Our results also depend on the workloads. We have developed and ported workloads that we believe are representative of future MTTOP workloads but, as with expected system models, it is difficult to predict the future. We now consider the impact of different workloads.

- CPU-like workloads: We have assumed workloads that have regular behaviors and that are particularly well-suited to MTTOPs. If more CPU-like workloads are ported to MTTOPs, these workloads may resemble CPU workloads in that they have a smaller load-to-store ratio and/or fewer available threads to run in parallel.
- Hierarchical threading: We have assumed a programming model with a flat thread organization, but today's GPGPU programming paradigms provide a hierarchy of threads. For example, threads may be grouped into warps, and warps may be grouped into thread blocks. With hierarchical thread grouping, we may wish to consider consistency models that are aware of this hierarchy (e.g., consistency models that provide different ordering guarantees within a warp than across warps).

10. RELATED WORK

There has been some prior work in exploring memory models for emerging chip types.

Fung et al. [12] develop a transactional memory (TM) [21] model and a hardware implementation of TM for GPUs. The transactional semantics of the memory system are a form of memory model. It is an open question whether TM is appropriate for GPUs. Regardless, our work complements their work by addressing non-TM operations and by illuminating the behaviors of MTTOP workloads running on MTTOP memory systems.

Rigel [16] presents a memory model for many-core accelerators (e.g., 1000-core accelerators). Rigel provides a single shared memory with two classes of operations: local and global. Local operations access a core's cluster cache, whereas global operations bypass the cluster cache and directly access a global cache. Inter-cluster coherence is maintained with a software coherence protocol. The memory system provides Fence instructions for ordering purposes, but there is no explicit consideration of ordering rules across loads and stores (i.e., nothing analogous to current CPU memory consistency models).

Cohesion [17] provides a scalable memory system for many-core chips. The key insight is to provide software cache coherence whenever possible and use hardware cache coherence only when required for performance. Cohesion can dynamically change whether a region of memory uses software or hardware for coherence. This work is somewhat orthogonal to our work, in that it addresses the implementation of cache coherence in chips with a very large number of cores, whereas we consider the consistency model of such chips and assume hardware coherence.

Other recent work has explored how to extend the scalability of hardware cache coherence to hundreds and even thousands of cores. This work is important and related in that we assume cache coherence in our MTTOP system model. Ferdman et al. [11] developed the Cuckoo Directory scheme that enables a directory protocol's storage requirements to scale gracefully. Martin et al. [23] show that existing coherence protocol techniques can be composed to create protocols that scale to hundreds of cores.

11. CONCLUSIONS

In this paper, we have re-visited the issue of hardware memory consistency models in the context of massively-threaded throughput-oriented processors. The conventional wisdom is that such processors should benefit from weak consistency models and that the choice of consistency model should have a major impact due to the extraordinary concurrency that is enabled by MTTOPs. Thus, perhaps surprisingly, our results show that sequential consistency achieves performance comparable to weaker consistency models on a variety of MTTOP benchmarks. It is obviously premature to definitively conclude that MTTOPs should implement SC or another relatively strong consistency model, but our analysis and results suggest that the conventional wisdom favoring weak models may be misguided. If strong consistency models, such as TSO, are indeed viable for MTTOPs, it may facilitate integration of MTTOPs with CPU cores that implement strong consistency models.

12. ACKNOWLEDGMENTS

We thank Brad Beckmann and Steve Reinhardt for helpful discussions about this work. We thank Mark Hill, Milo Martin, and Mark Oskin for their valuable help in improving our presentation of this work. This material is based on work supported by AMD and by the National Science Foundation under grant CCF-1216695.

13. REFERENCES

[1] S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial," *IEEE Computer*, vol. 29, no. 12, pp. 66–76, Dec. 1996.

[2] S. V. Adve and M. D. Hill, "Weak Ordering - A New Definition," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990, pp. 2–14.

[3] Arvind and J.-W. Maessen, "Memory Model = Instruction Reordering + Store Atomicity," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, 2006, pp. 29–40.

[4] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC Benchmark Suite: Characterization and Architectural Implications," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2008.

[5] N. Binkert et al., "The Gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, Aug. 2011.

[6] H.-J. Boehm and S. V. Adve, "Foundations of the C++ Concurrency Memory Model," in *Proceedings of the Conference on Programming Language Design and Implementation*, 2008.

[7] P. Bright, "Transactional Memory Going Mainstream with Intel Haswell," *Ars Technica*, Feb-2012. [Online]. Available: http://arstechnica.com/business/news/2012/02/transactional-memory-going-mainstream-with-intel-haswell.ars.

[8] H. W. Cain and M. H. Lipasti, "Memory Ordering: A Value-Based Approach," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[9] S. Che et al., "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *IEEE International Symposium on Workload Characterization*, 2009, pp. 44 – 54.

[10] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, 2004.

[11] M. Ferdman, P. Lotfi-Kamran, K. Balet, and B. Falsafi, "Cuckoo Directory: A Scalable Directory for Many-Core Systems," in *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 169–180.

[12] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware Transactional Memory for GPU Architectures," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, 2011, pp. 296–307.

[13] C. Gniady, B. Falsafi, and T. N. Vijaykumar, "Is SC + ILP = RC?," in *Proceedings of the 26th Annual International Symposium on Computer Architecture*, 1999, pp. 162–171.

[14] B. A. Hechtman and D. J. Sorin, "Evaluating Cache Coherent Shared Virtual Memory for Heterogeneous Multicore Chips," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, 2013.

[15] M. D. Hill, "Multiprocessors Should Support Simple Memory Consistency Models," *IEEE Computer*, vol. 31, no. 8, pp. 28–34, Aug. 1998.

[16] J. H. Kelm et al., "Rigel: An Architecture and Scalable Programming Interface for a 1000-Core Accelerator," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009, pp. 140–151.

[17] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel, "Cohesion: a Hybrid Memory Model for Accelerators," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, 2010, vol. 38, pp. 429–440.

[18] R. Krashinsky et al., "The Vector-Thread Architecture," in *31st International Symposium on Computer Architecture*, 2004, pp. 52–63.

[19] M. Kulkarni, M. Burtscher, C. Cascaval, and K. Pingali, "Lonestar: A Suite of Parallel Irregular Programs," in *IEEE International Symposium on Performance Analysis of Systems and Software*, 2009, pp. 65 –76.

[20] L. Lamport, "How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, vol. C–28, no. 9, pp. 690–691, Sep. 1979.

[21] J. Larus and R. Rajwar, *Transactional Memory*. Morgan & Claypool Publishers, 2007.

[22] J. Manson, W. Pugh, and S. V. Adve, "The Java Memory Model," in *Proceedings of the 32nd Symposium on Principles of Programming Languages*, 2005.

[23] M. M. K. Martin, M. D. Hill, and D. J. Sorin, "Why On-chip Cache Coherence is Here to Stay," *Communications of the ACM*, vol. 55, no. 7, pp. 78–89, Jul. 2012.

[24] A. Meixner and D. J. Sorin, "Dynamic Verification of Sequential Consistency," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 482–493.

[25] S. Owens, S. Sarkar, and P. Sewell, "A Better x86 Memory Model: x86-TSO," in *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, 2009.

[26] R. Rajwar and J. R. Goodman, "Speculative Lock Elision: Enabling Highly Concurrent Multithreaded Execution," in *Proceedings of the 34rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2001.

[27] P. Ranganathan, V. S. Pai, and S. V. Adve, "Using Speculative Retirement and Larger Instruction Windows to Narrow the Performance Gap between Memory Consistency Models," in *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures*, 1997, pp. 199–210.

[28] L. Seiler and et al., "Larrabee: A Many-Core x86 Architecture for Visual Computing," in *Proceedings of ACM SIGGRAPH*, 2008.

[29] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen, "x86-TSO: A Rigorous and Usable Programmer's Model for x86 Multiprocessors," *Communications of the ACM*, Jul. 2010.

[30] I. Singh, A. Shriraram, W. W. L. Fung, M. O'Connor, and T. M. Aamodt, "Cache Coherence for GPU Architectures," in *Proceedings of the 19th IEEE International Symposium*

on High-Performance Computer Architecture, 2013, pp. 578–590.

[31] R. L. Sites, Ed., *Alpha Architecture Reference Manual*. Digital Press, 1992.

[32] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware Microarchitecture," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 2–13.

[33] D. J. Sorin, M. D. Hill, and D. A. Wood, *A Primer on Memory Consistency and Cache Coherence*, Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2011.

[34] S. Steele, "ARM GPUs: Now and in the Future." http://www.arm.com/files/event/8_Steve_Steele_ARM_GPUs_Now_and_in_the_Future.pdf, Jun-2011.

[35] V. Tipparaju and L. Howes, "HSA for the Common Man." http://devgurus.amd.com/servlet/JiveServlet/download/1282191-1737/HC-4741_FINAL.pptx, 2012.

[36] D. L. Weaver and T. Germond, Eds., *SPARC Architecture Manual (Version 9)*. PTR Prentice Hall, 1994.