2015 International Conference on Parallel Architecture and Compilation

Parallel Methods for Verifying the Consistency of Weakly-Ordered Architectures

Adam McLaughlin
Georgia Institute of Technology
adam27x@gatech.edu

Duane Merrill Michael Garland
NVIDIA
{dumerrill,mgarland}@nvidia.com

David A. Bader
Georgia Institute of Technology
bader@cc.gatech.edu

***Abstract*—Contemporary microprocessors use relaxed memory consistency models to allow for aggressive optimizations in hardware. This enhancement in performance comes at the cost of design complexity and verification effort. In particular, verifying an execution of a program against its system's memory consistency model is an NP-complete problem. Several graph-based approximations to this problem based on carefully constructed randomized test programs have been proposed in the literature; however, such approaches are sequential and execute slowly on large graphs of interest. Unfortunately, the ability to execute larger tests is tremendously important, since such tests enable one to expose bugs more quickly. Successfully executing more tests per unit time is also desirable, since it allows for one to check for a greater variety of errors in the memory subsystem by utilizing a more diverse set of tests.**

This paper improves upon existing work by introducing an algorithm that not only reduces the time complexity of the verification process, but also facilitates the development of parallel algorithms for solving these problems. We first show performance improvements from a sequential approach and gain further performance from parallel implementations in OpenMP and CUDA. For large tests of interest, our GPU implementation achieves an average application speedup of 26.36x over existing techniques in use at NVIDIA.

***Keywords*-Memory Consistency Verification; Relaxed Memory Models; Graph Algorithms; Parallel Algorithms**

I. Introduction

Modern architectures use memory reordering techniques to obtain better performance and energy efficiency. For instance, high latencies to memory can be hidden by overlapping memory accesses with computation. Allowing for the reordering of memory instructions comes at the cost of design complexity, verification effort, and programmer burden [3]. On today's shared memory multiprocessor systems these problems are exacerbated by an increasing number of cores. Although techniques such as speculative execution, shared caches, coherence mechanisms, and instruction pipelining all have well-known benefits, an improper implementation of these techniques can lead to subtle memory errors such as data corruption or illegal instruction ordering [10]. Furthermore, the use of these techniques are visible to programmers, especially those concerned with the low-level performance-sensitive details of the system [1].

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited)

Shared memory multiprocessor systems have a *memory consistency model* that is essentially a contract between hardware and software regarding the semantics of memory operations [21]. The simplest memory consistency model is the *Sequential Consistency* (SC) model. Under this model, all processors observe the same ordering of operations serviced by memory. Processors execute instructions precisely in the order specified by the program, or *program order*. A read from a particular location in memory is guaranteed to return the value of the last write to that location under the SC model. Although this model is intuitive, it restricts the use of performance optimizations commonly used by hardware and compiler designers [1].

In contrast, the ARM processors considered in this work have a significantly more relaxed memory model [5]. Weakly-ordered ARM processors allow speculative execution and reordering of a thread's reads and writes. Additionally, writes are not guaranteed to be simultaneously visible to other cores. A consequence of these relaxations is that, the order in which instructions access memory (e.g., the *memory order*) on such processors is distinct from the program order. In comparison to the SC model, many more outcomes satisfying relaxed memory models exist, which makes direct verification a challenging process.

The verification process affects a processor's time to market: verification plays an important role in discovering defects early during the design process when remediation is less costly. As such, the problem of verifying that a multiprocessor complies with its memory consistency model has seen significant attention in the literature [10], [4], [15], [2], [16]. Formal approaches attempt to exhaustively check a design using proof methodologies, but cannot scale to the size of current microprocessor designs that require millions of lines of RTL code [14], [6]. Furthermore, formal approaches tend to employ a high-level abstraction of a microarchitecture design, neglecting the details of its implementation. Unfortunately, the implementation itself is a significant source of bugs in large designs [6].

The verification of an execution against its system's memory consistency model is an NP-complete problem [4], [21]. Contemporary solutions thus trade time for accuracy, providing polynomial time approaches that are incomplete: they may miss violations of the memory model, but violations that are found are legitimate. We present our work in the context of TSOtool, a software package that employs a graph-based approach for verifying the Total Store Order (TSO) model [10]. TSOtool easily extends to other relaxed memory models, can evaluate specific processor implementations as well as generic protocols, and has been used to find subtle bugs in commercial products [10], [15].

Despite the usage of polynomial time verification algorithms, consistency verification is typically limited by both strong and weak scalability. Since these techniques are incomplete, test coverage is dictated by the number of program traces that can be evaluated. Practical scalability with respect to trace size is also important: it is desirable for instruction traces to comprise very long periods of race conditions and asynchronous behavior among parallel processors [16]. The bugs that existing tools are designed to find are deep corner cases that slip through pre-silicon verification. Longer tests put caches and supporting logic in more interesting states that are likely to trigger such bugs, if they exist. A high-performance approach additionally allows verification engineers to tailor their tests to specific issues much more rapidly given results from prior tests. Hence it is desirable to execute larger tests as well as to execute a single test as fast as possible.

This paper addresses these challenges and presents the following contributions and results:

- We improve existing iterative, graph-based approaches for memory consistency verification by diminishing how frequently data structures need to be updated. This refinement reduces the work complexity of the algorithm from $O(n^2p^2d_{max})$ to $O(n^2p)$ per iteration for a program execution graph with n vertices, p virtual processors, and maximum vertex outdegree d_{max}. We prove that this reduction of work converges to the same result that would be computed by prior techniques.
- In addition to sequential speedups over existing approaches, our approach is more amenable to parallelization because it performs batched graph updates with less frequency. We implement parallel versions of our sequential approach in both OpenMP and CUDA and for sufficiently large test instances of interest, our GPU approach can achieve over an order of magnitude speed increase over our sequential approach.
- Although our optimizations are focused on a subset of the overall consistency verification problem, for large test cases our GPU approach achieves an average application speedup of 26.36x over a modified version of TSOtool used to verify ARM-based processors that we have been experimenting with at NVIDIA.

II. Background

The goal of our application is to verify the correctness of the memory subsystem as it is being designed, which implies that we need to ensure that the processor's memory consistency model is not violated. Based on dependencies between instructions of a program that are required to be satisfied by the rules of the architecture (such as read after write hazards), we can construct a partial ordering of memory instructions, which we model as a directed graph. Given the outcome of a specific execution of our program, we can *infer* additional edges that are required to be satisfied by the rules of the consistency model (such as ensuring that a load reads the most recently written data to memory). These inferred edges densify the graph representation, creating a more complete (but not necessarily total) ordering of memory instructions. If a cycle manifests from this process then we have a contradiction in the memory order and thus the memory model was violated or is invalid.

The remainder of this section provides more detail regarding the memory consistency verification process as implemented by TSOtool [10].

A. Constraint Graph

Let a graph $G = (V, E)$ consist of a set V of $n = |V|$ vertices and a set E of $m = |E|$ edges. A *directed edge* $(u, v) \in E$ originates from vertex u and terminates at vertex v. A *cycle* is a sequence $u_0, u_1, \ldots, u_k, u_0$ of vertices starting and ending at the same vertex (u_0) such that there exists an edge in G between each consecutive pair of vertices in the sequence. The *diameter* of a graph is the length of the longest shortest path between any pair of vertices.

Our method of consistency verification is concerned with *constraint graphs* [3], [13], partially-ordered directed graphs that model the memory semantics of a given program execution. The vertices of the graph represent dynamic processor instructions and the edges represent dependence relationships. Instructions have several key attributes:

- Instruction type (Load, Store, or Barrier)
- Address (Memory location accessed by the instruction)
- Data (Value read by loads or written by stores)
- Processor (A numerical identifier of the core that issued this instruction)

Of course, traces of programs can have other types of instructions, such as floating-point arithmetic (FADD, FSUB, FMUL, FDIV, etc.); however, these instructions do not access memory and can safely be ignored for the consistency verification process.

Edges of the graph represent *memory ordering* between dynamic instructions. That is, an edge from instruction u to instruction v signifies that instruction u accessed memory before instruction v. Note that memory order is distinct from instruction execution order because in-flight reordering is allowed by the ARM architecture. Indeed, an edge from u to v in the same processor does not imply that u preceded v in the instruction stream.

B. TSOtool Workflow

Figure 1 illustrates the consistency verification process employed by the work from Hangal *et al.* on TSOtool. The

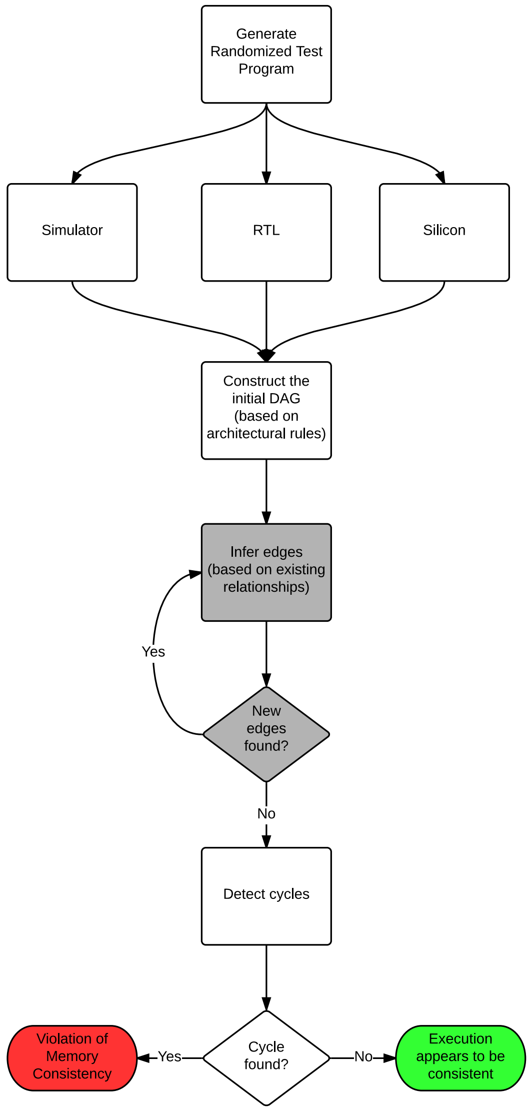

Generate Randomized Test Program
Simulator
RTL
Silicon
Construct the initial DAG (based on architectural rules)
Infer edges (based on existing relationships)
Yes
New edges found?
No
Detect cycles
Violation of Memory Consistency
Yes
Cycle found?
No
Execution appears to be consistent

Figure 1. Design flow for memory consistency verification

process begins with a randomly generated test program. Test programs can be generated using parameters such as the total number of instructions per core, the number of unique store locations, the ratio of loads to stores, and the types of memory instructions to target various subsets of the memory system. The generated test program is carefully constructed such that each store in the graph writes a *unique value* to memory [21]. The uniqueness of store values provides a trivial mapping of a load to the store that wrote its data, which simplifies the algorithms for analyzing the ensuing graph. Since the data written and read from memory is independent from the behavior of the protocol, using unique store data does not limit the diversity of test cases that can be generated.

Once the test program is generated, it is executed (on a simulator, RTL, or silicon) to obtain the actual data values observed by each load instruction. This information, combined with the rules of the underlying architectural and consistency models allows one to create the initial directed graph for analysis. This graph comprises two classes of edges:

- *Static* edges. These relationships are enforced by the architecture in the presence of data hazards. For instance, the ARM architecture specifies that operations issued prior to a memory barrier must execute before operations after the memory barrier. The architecture also prevents reordering of loads and stores in the presence of data hazards.
- *Observed* edges. These relationships are enforced by the data read by load instructions during a particular execution of the test program. For example, if a load L on processor p_0 reads the (globally unique) value x from address A and we know that store S on processor p_1 wrote the value of x to A, we can add a directed edge from S to L in the graph, because the consistency model requires that loads read data from the store that most recently wrote to memory.

Once the initial graph is constructed, the existing relationships (edges) in the graph can be used to infer additional relationships according to the consistency model. Any such new edges may lead to further relationship inferences, and edges are inferred iteratively until the graph has reached a fixed point. At this point, the graph is checked for cycles in linear time using a technique known as trimming [18]. If the graph contains one or more cycles then we are certain that the consistency model was violated. If the graph has no cycles the execution of the program *appears* to be consistent. Consistency is not guaranteed because these static and observed relationships are not complete: there are further (mutually exclusive) sets of *plausible* relationships that could be established in accordance with the memory model to provide a total ordering of memory instructions and thus a perfectly accurate verification. However, determining whether there exist any such plausible sets that do not induce dependence cycles is NP-complete [16]. For more information regarding static, observed, and inferred edges, we refer the reader to [14].

This iterative process corresponds to the gray boxes in the center of Figure 1, which represent a substantial portion of the overall consistency verification process and thus our focus for optimization and parallelization.

C. Inferred Edge Insertions

There are two types of inferred edge insertions made by TSOtool, referred to as rule 6 and 7 insertions[1] (using the TSOtool notation [10] and shown in Figures 2 and 3, respectively). Notationally, $ST[A] \rightarrow 1$ means that this instruction

[1]Rules 1-3 cover static edges and rules 4-5 cover observed edges.

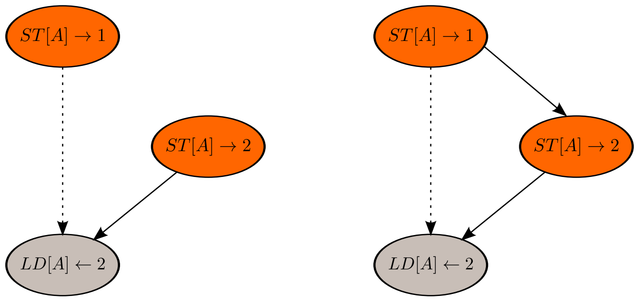


Figure 2. Example of a Rule 6 inferred edge insertion

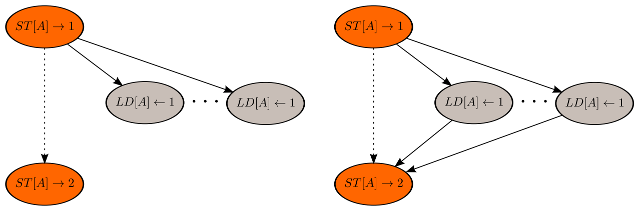


Figure 3. Example of Rule 7 inferred edge insertions

wrote the value 1 to location A. Similarly, $LD[A] \leftarrow 2$ means that this instruction read the value 2 from location A. The edges drawn with dotted lines in the Figures denote reachability, meaning that the head of the edge can reach the tail of the edge either directly or transitively. In contrast, the edges drawn with solid lines denote the stronger notion of direct neighbors in the graph.

An example of a Rule 6 insertion is shown in Figure 2. We can see from Figure 2a that the store writing the value 1 can reach a load to the same address that reads the value 2. Since the load reads a different value than the store that can reach it and since store values are unique, there must be another store that writes 2 to location A before the load occurs. Furthermore, this store must have accessed memory after the store that wrote 1 because otherwise the load would have read the value 1. Therefore, we can insert an edge from the store that wrote 1 to the store that wrote 2, as shown in Figure 2b.

Figure 3 shows an example of Rule 7 edge insertions. In this case, a series of loads read the value 1, as shown in Figure 3a. Since store values are unique, these loads must all be reading a value written by the same store. If that store can reach another store in the graph that accesses the same location in memory (and, as it must by design, writes different data), we know that the series of loads must all precede the later store since otherwise the loads would have read the value from that store instead (i.e., the value 2). Hence, we can insert edges from each of these loads to the later store, as shown in Figure 3b.

III. Sequential Methodology

This section describes several approaches for inferring edges from a constraint graph to solve the memory consistency verification problem. We give an overview of the algorithm used by NVIDIA's application of TSOtool to verify the memory consistency of ARM processors (which have a weaker memory model than TSO). Next, we explain several key performance optimizations to TSOtool [15] that we leverage for our parallel implementation.

A. Initial Algorithm

Algorithm 1: Simple Sequential Approach for Inferring Edges

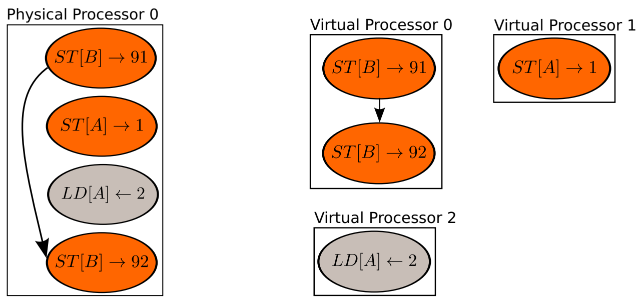
```
1 for {$S \in V \mid S.type = ST$} do
2     for {$X \in V \mid S \leq X$} do
3         if $S.location = X.location$ then
4             if $X.type = LD \wedge S.data \neq X.data$ then
5                 //Add Rule 6 edge from S to the parent
                  store of X
6             else if $X.type = ST$ then
7                 for {$L \in V \mid S.data = L.data$} do
8                     //Add Rule 7 edge from L to X
```

Algorithm 1 shows a straightforward approach for an iteration of inferring edges. This process is repeated iteratively until a fixed-point is reached. The outermost loop iterates through each store vertex S in the graph. The inner loop on Line 2 iterates through every reachable vertex from S. Here we use the notation $S \leq X$ to represent that instruction S comes before instruction X in memory order, which is equivalent to saying that X is reachable from S in the graph.

The remaining lines in Algorithm 1 logically enforce the inferred edge rules depicted by Figures 2 and 3. The loop in Line 7 essentially represents the set of loads L that read data from S. The complexity of finding these vertices can be reduced since we can explicitly map each store to its respective child loads once the initial graph is constructed, as this information is constant throughout the execution of the program. Overall, the time complexity of one iteration of Algorithm 1 is $O(n^3)$, assuming that edges can be inserted into the graph in $O(1)$ time.

B. Virtual Processors and Reverse Vector Time Clocks

Manovit and Hangal develop a more efficient algorithm that leverages transitivity via the use of *virtual processors* and *Reverse Time Vector Clocks* (RTVCs) [15]. The authors discovered that the set of instructions from each physical processor can be grouped into subsets of instructions that belong to virtual processors (vprocs) where each virtual processor is sequentially consistent (and thus have equivalent



(a) Physical (b) Virtual

Figure 4. Splitting of a physical processor in sequentially consistent virtual processors

program order and memory order). Figure 4 shows an example of how a physical processor can be split into sequentially consistent virtual processors. This process depends on the memory model being targeted; for the ARM processors considered in this study, instructions on a physical processor are grouped by their memory location and instruction type. Since instructions that access different locations in memory can be freely reordered by the hardware, they must be assigned to different virtual processors. Although the instructions belonging to vprocs 1 and 2 in Figure 4 access the same memory location A they must also belong to separate virtual processors because the view in which these instructions are executed from *other physical processors* can be out of order.

Algorithm 2: Optimized Sequential Approach for Inferring Edges

```
1  for {$S \in V \mid S.type = ST$} do
2      for {$X \in S.rtvc[P], P \in p$} do
3          while $X \neq vprocs[P].end$ do
4              if $X.type = LD \wedge S.data \neq X.data \wedge S.location = X.location$ then
5                  //Add Rule 6 edge from S to the parent store of X
6                  $update\_RTVCs()$
7                  break //Move on to next vrpoc
8              else if $X.type = ST \wedge S.location = X.location$ then
9                  for {$L \in V \mid S.data = L.data$} do
10                     //Add Rule 7 edge from L to X
11                 $update\_RTVCs()$
12                 break //Move on to next vproc
13             else
14                 $X \leftarrow X.next$
```

A consequence of this grouping is that if a vertex S has an outgoing edge to an instruction X in virtual processor p then S implicitly precedes all of the successors of X that are also in p. Thus, a given instruction only needs to inspect its outgoing edges to its earliest successors in each virtual processor [15]. This bounds the number of reachable vertices to be inspected by each store by p, the number of virtual processors, rather than n, the total number of vertices. The data structure that points from each vertex to its earliest successors in each vproc is referred to as a Reverse Time Vector Clock (RTVC) [15], named after the popular approach for partially ordering events in distributed computing [9], [12].

Algorithm 2 shows how the use of virtual processors and reverse time vector clocks can significantly reduce the complexity required to infer edges. The RTVC of an instruction S is denoted as $S.rtvc$ and the earliest successor of S to vproc P is denoted as $S.rtvc[P]$. The algorithm simply finds the earliest successors of each store to each vproc for which edges can be inferred, given a test program containing p vprocs. These techniques reduce the complexity of inferring edges from $O(n^3)$ to $O(n^2p^2d_{max})$, where d_{max} is the maximum degree of any vertex $v \in V$. Since $p^2d_{max} < n$ these changes have led to order of magnitude improvements in execution time over the approach outlined in Algorithm 1 [15].

Algorithm 3: Function for Updating RTVCs

```
1  $topo \leftarrow reverse(get\_topological\_sort(G))$
2  for {$U \in topo$} do
3      $U.rtvc[i] \leftarrow \infty, \forall i \in p$
4      for {$V \in U.adjacency\_list$} do
5          $W \leftarrow U.rtvc[V.vproc]$
6          if $W = \infty$ then
7              $U.rtvc[V.vproc] \leftarrow V$
8          else if $V.program\_order < W.program\_order$ then
9              $U.rtvc[V.vproc] \leftarrow V$
10         //Now check transitive edges through V
11         for {$P \in p$} do
12             if $V.rtvc[P] \neq \infty$ then
13                 if $U.rtvc[P] = \infty$ then
14                     $U.rtvc[P] \leftarrow V.rtvc[P]$
15                 else if $V.rtvc[P].program\_order < U.rtvc[P].program\_order$ then
16                     $U.rtvc[P] \leftarrow V.rtvc[P]$
```

The recomputation of RTVCs ($update\_RTVCs()$) in Lines 6 and 11 of Algorithm 2 deserves separate attention. Algorithm 3 shows the details of this function. Line 1 returns the topological sort of the input graph G in reverse order.

This operation is easily completed in linear time [7]. The reverse order of the topological sort is necessary to execute the iterations of the loop on Line 2 in the proper order. The loop on Line 4 looks at the direct neighbors of U. If a given neighbor V of U belongs to a vproc that has yet to be seen from the perspective of U, then it is the earliest successor (so far) from u to that vproc (Line 7). Otherwise, if V belongs to a vproc that U already has an entry for, the program ordering of that entry and V can be compared (Line 8) to determine which successor is earlier in memory order (recall that instructions belonging to the same vproc are ordered in memory as they are in the program). The loop on Line 11 looks transitively through the RTVC of V to update the RTVC values of U.

IV. Facilitating Parallelism

Despite the performance gains seen by the algorithm presented in the previous section, large tests of interest still take days to execute on server class machines. A natural way to elicit further performance gains is to parallelize the algorithm and run it on multi-core CPUs or GPU accelerators. However, Algorithm 2 isn't trivially parallelized. Every time an edge is inserted, the RTVCs of the head of the edge and its ancestors are updated. In general, updating RTVCs requires $O(npd_{max})$ time per edge insertion. There can be up to $O(n^2)$ edge insertions in the worst case, although the number of added edges is typically a small multiple of n. Regardless, the time spent updating RTVCs is significant and these updates are a barrier to parallelism since the iterations of the inner loops of Algorithm 2 are dependent on the RTVC values.

Algorithm 4: Parallel-friendly Approach for Inferring Edges

```
1  for $\{S \in V \mid S.type = ST\}$ do
2      for $\{X \in S.rtvc[P], P \in p\}$ do
3          while $X \neq vprocs[P].end$ do
4              if $X.type = LD \wedge S.data \neq X.data \wedge S.location = X.location$ then
5                  //Add Rule 6 edge from S to the parent store of X
6              else if $X.type = ST \wedge S.location = X.location$ then
7                  for $\{L \in V \mid S.data = L.data\}$ do
8                      //Add Rule 7 edge from L to X
9              else
10                 $X \leftarrow X.next$
11 $update\_RTVCs()$
```

To enable parallelism we propose a simple alteration to Algorithm 2: reduce the frequency of RTVC updates from once per edge insertion to once per iteration of inferring edges. Algorithm 4 shows this alteration. This change allows for the iterations of the for loops on Lines 1 and 2 to be safely executed independently in parallel at the potential cost of some unnecessary work in the form of edge insertions that provide no information with respect to the memory order of instructions. Even though the graph changes as the algorithm progresses, the work required by an iteration of Algorithm 4 only depends on the RTVC values and not the current state of the graph. When the RTVC values are updated at the end of an iteration of inferring edges, the edges found during the iteration can be inserted into the graph in one batched operation and then the new RTVC values can be derived from the updated graph.

Lazily updating the RTVCs instead of greedily updating them can result in situations where reachable vertices from each store are checked when such a check isn't strictly necessary, as is done in Algorithm 1. However, our evaluation demonstrates that updating RTVCs less frequently is well worth the cost of the extra work.

To show that this strategy maintains the correctness of the approaches outlined in the previous algorithms, it will suffice to show that our method is both sound and "as complete." Let G_i be the graph obtained after i iterations of Algorithm 1 and H_i be the graph after i iterations of Algorithm 4. Let I be the total number of iterations, noting that this value may be different for Algorithms 1 and 4. It follows that G_I and H_I are the resultant graphs after these algorithms terminate.

Theorem 1. *Soundness. All edges inserted by Algorithm 4 represent valid memory orderings of instructions.*

Proof: To show that Theorem 1 is valid, we note that even if RTVC values are "stale," they always point to a successor of S. Since Algorithm 1 iterates through *all* successors of S for inferring edges, any edges that are inserted regardless of whether or not RTVCs are consistent with the current state of the graph must be valid. Any RTVCs that are stale will be updated for the next iteration such that the earliest successors from each store to each vproc will always be checked before the algorithm terminates. Since the Algorithms 1 and 2 have been shown to only insert valid edges [15], [10] and since our algorithm only inserts edges that either of these algorithms would insert, our algorithm must also only insert valid edges. ■

Theorem 2. *Completeness parity. If G_I contains a cycle then H_I must also contain a cycle.*

Proof: To satisfy Theorem 2, we assume for the purpose of contradiction that H_I has no cycle when G_I has a cycle. This result implies that we have neglected to insert some edge e that created the cycle in G_I. However, Algorithm 4 ensures that the earliest successor from each store vertex to each vproc is checked for the application of rule 6 and

7 edges. Since it was shown that such checks provide the same information as checking all successors from each store [15], e must have been found, else it does not exist. Since e was not found, it must not exist, contradicting our initial assumption. ■

In addition to facilitating parallelism, Algorithm 4 performs less work to update RTVCs. If k edges are inserted into the graph, Algorithm 2 requires $O(k)$ RTVC updates. In contrast, Algorithm 4 only requires $O(i)$ RTVC updates for i iterations because the number of RTVC updates scales with the number of iterations rather than the number of edge insertions. Since $k = O(n^2)$ in the worst case (and is $O(n)$ in practice) whereas $i \leq 10$ for all of the test cases used for this study, Algorithm 4 requires significantly less overhead for RTVC updates than Algorithm 2. Overall, Algorithm 4 reduces the work complexity of inferring edges from $O(n^2p^2d_{max})$ to $O(n^2p)$ per iteration.

V. Parallel Methodology

The approach to verifying memory consistency described in the previous section is not the first attempt to parallelize this class of algorithms. Roy *et al.* present a parallel implementation of TSOtool that targeted the Intel IA-32 and Itanium architectures [22]. Their approach cannot utilize virtual processors and the RTVC-based optimizations described in [15] because of complications arising from using specific memory types on the IA-32 and Itanium architectures.

Although these complications encouraged the design of a more general approach, the algorithms from Roy *et al.* have a few significant weaknesses. Firstly, they choose to store the graph as an adjacency matrix, requiring $O(n^2)$ space despite the fact that constraint graphs are typically quite sparse. This decision prohibited scalability to graphs larger than $10,000$ vertices in their experiments. Secondly, it is unclear if the tests performed by Roy *et al.* are scalable to large thread counts or portable to graphs with differing characteristics, such as the number of accessed memory locations or the ratio of loads to stores.

In the remainder of this section we discuss two parallel implementations of Algorithm 4: one in OpenMP for multicore CPUs and the other in CUDA for GPU accelerators.

A. OpenMP

Considering that shared-memory systems using OpenMP tend to have a limited number of threads, decomposing problems such that each thread has a sufficient amount of work is less challenging than on systems with thousands of concurrent threads, such as NVIDIA's GPUs or Intel's Xeon Phi coprocessors. Since we know in advance that the number of threads is small, we provide each thread with its own storage space for collecting newly inserted edges to reduce communication overhead. Threads are assigned to iterations of the for loop in Line 1 of Algorithm 4, which can independently traverse the RTVCs from their assigned store vertex and add edges to their local lists. Once the entire iteration is complete these thread-specific lists are trivially reduced into one global list, which is used to update the graph.

Algorithm 5: OpenMP Approach for Inferring Edges

```
1  $added\_edges \leftarrow vector(num\_threads())$
2  for $\{S \in V \mid S.type = ST\}$ do in parallel
3      for $\{X \in S.rtvc[P], P \in p\}$ do
4          while $X \neq vprocs[P].end$ do
5              if $X.type = LD \wedge S.data \neq X.data \wedge S.location = X.location$ then
6                  $added\_edges[get\_id()].insert(S \to X)$
7              else if $X.type = ST \wedge S.location = X.location$ then
8                  for $\{L \in V \mid S.data = L.data\}$ do
9                      $added\_edges[get\_id()].insert(L \to X)$
10             else
11                 $X \leftarrow X.next$
12 foreach $\{Partition\ t \in num\_threads()\}$ do
13     foreach $\{Edge\ e \in added\_edges[t]\}$ do
14         $new\_edges.insert(e)$
15 $update\_RTVCs()$
```

Algorithm 5 shows the details of this approach. The vector $added\_edges$ of length $num\_threads()$, the number of OpenMP threads, allows each thread to concurrently find edges without communication or race conditions. The for loop on Line 12 sequentially accumulates the results collected this way into one data structure so that one large update to the graph can be made instead of $num\_threads()$ (smaller) updates. Although we had the option of utilizing finer granularities of parallelism, using the coarsest level of granularity maximizes independent work among threads and minimizes the OpenMP overhead of creating and destroying threads.

B. CUDA

Our initial approach to parallelizing Algorithm 4 using CUDA involved a slightly more complicated thread decomposition than our OpenMP approach. We initially assigned thread blocks (groups of threads) to each store processed by the outermost loop on Line 1 of Algorithm 4. The threads within each block were assigned to inspect each vproc from their respective store as seen on Line 2 of Algorithm 4. This approach achieved limited processor utilization because the number of vprocs is small relative to the number of threads per block, which should be a multiple of the warp size (currently 32 threads on NVIDIA hardware). Additionally, the work done by each thread in this manner is fairly uneven. At one end of the spectrum, a thread may find that its store has a null RTVC value for the vproc that it is looking at and thus, the thread has no work to complete. In contrast, another thread may simultaneously find that the store does have an RTVC entry to this vproc but the earliest successor from the store to the vproc is much later than the initial entry. In this latter case the thread must traverse through the vproc and possibly insert edges from each load that reads from the store to this successor.

It turns out that simply taking advantage of the large amount of coarse-grained parallelism through the number of stores ($O(n)$) is a better approach. This approach more efficiently utilizes the processor because threads are constantly kept busy by processing independent store vertices rather than waiting for the critical thread in a given block to finish traversing its vproc and adding edges. Using our initial approach, if the number of vprocs modulo 32 (the warp size) is not 0 then threads will have an unequal number of vprocs to inspect. In the worst case, one thread has one more vproc than all of the others, meaning that the remaining threads in the block will all idle while the one thread with additional work inspects its additional vproc. Using a coarser approach eliminates this issue because each thread processes all vprocs from a given store. A load imbalance may still exist in the number of stores to be processed per thread; however, the small number of Streaming Multiprocessors (SMs) on the GPU implies that the same HW lanes will sequentially process many stores, making this ”off by one” load imbalance insignificant.

Algorithm 6: CUDA Approach for Inferring Edges

```
1  for $\{S \in V \mid S.type = ST\}$ do in parallel
2      for $\{X \in S.rtvc[P], P \in p\}$ do
3          while $X \neq vprocs[P].end$ do
4              if $X.type = LD \wedge S.data \neq X.data \wedge S.location = X.location$ then
5                  $t \leftarrow atomicAdd(\&edge\_ptr, 1)$
                   $new\_edges[t].insert(S \rightarrow X)$
6              else if $X.type = ST \wedge S.location = X.location$ then
7                  for $\{L \in V \mid S.data = L.data\}$ do
8                      $t \leftarrow atomicAdd(\&edge\_ptr, 1)$
                       $new\_edges[t].insert(L \rightarrow X)$
9              else
10                 $X \leftarrow X.next$
11 $update\_RTVCs()$
```

Algorithm 6 shows how we alter our parallel implementation for a GPU architecture. Having separate subarrays for each thread as was done in Algorithm 5 is no longer practical because of the large (and tunable) number of threads offered by the GPU. We instead use one array and use atomic operations (Lines 5 and 8) to ensure that threads write to unique locations in memory. In practice we calculate the number of edges to be inserted by the loop on Line 7 and perform one *atomicAdd* rather than one *atomicAdd* for each inserted edge. Since the logic of finding which edges to add is the bottleneck of this algorithm we can easily update the graph (and RTVCs) on the CPU between iterations of Algorithm 6.

VI. Results

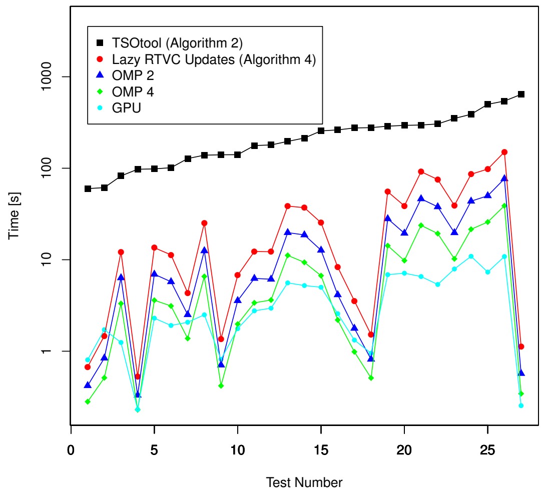
A. Experimental Setup

All experiments were run on a system with an Intel Core i7-2600K CPU and an NVIDIA GeForce GTX Titan GPU. The Intel Core i7-2600K has four cores, each of which run at 3.4 GHz, an 8 MB cache, and 16 GB of DRAM. The NVIDIA GeForce GTX Titan has a base clock that runs at 837 MHz, 6 GB of GDDR5 memory, a peak theoretical memory bandwidth of 288.4 GB/s, and is a compute capability 3.5 (“Kepler”) GPU.

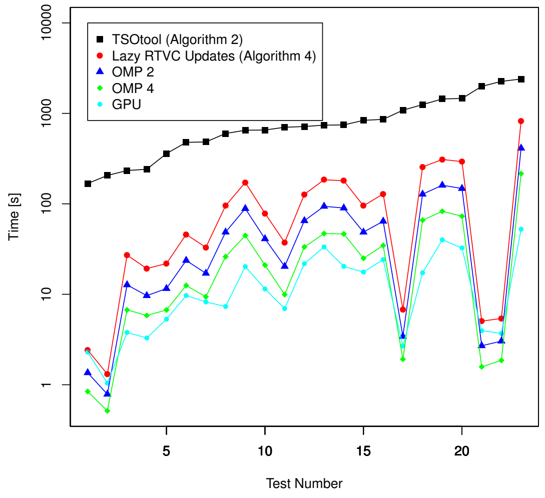
Sequential and OpenMP code was written in C++ and compiled with version 4.5.3 of the g++ compiler. CUDA code was compiled with the nvcc compiler and the CUDA 6.0 toolkit. We compare our approaches to an adaptation of TSOtool used to verify ARM processors. The system we use for testing contains four ARM Cortex-A57 cores. The Cortex-A57 microarchitecture implements the ARMv8-A 64-bit instruction set and has an out-of-order superscalar pipeline. The graphs we use for experimentation represent real traces used to find bugs in the implementation of the memory model (or bugs in the memory model itself). The graphs span sizes ranging from $n = 2^{18}$ to $n = 2^{22}$ vertices, with each vertex representing an instruction from one of the four processor cores. Each core issues the same number of instructions. The precise number of edges that each graph initially contains varies, but is fairly close to n. Hence, these graphs represent a particularly sparse, high-diameter, and low-degree network structure. We test a number of graphs of each size. These graphs vary in their proportion of load, store, and barrier instructions; number of virtual processors; and number of instruction dependencies.

B. Experimental Results

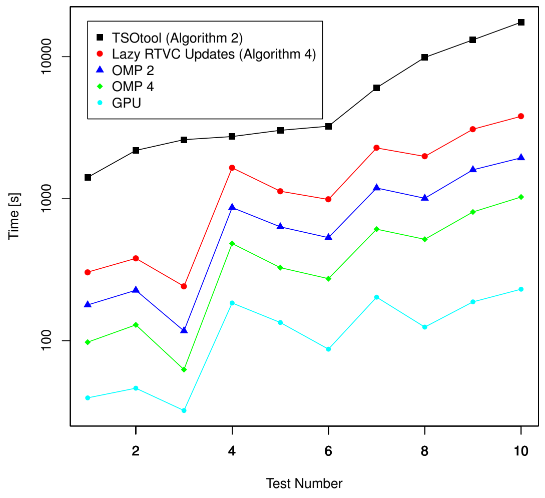
Figure 5 shows the performance characteristics of TSOtool and our improvements over this baseline. The data shown in these figures represent only the time spent adding edges to the graph; however, we will show that this time represents a vast majority of the overall execution time, and hence was our focus for optimization. Figure 5a shows tests that all have 128K instructions per core, or a total of $n = 2^{19}$ instructions across all four cores. Similarly, Figures 5b (and



(a) 128K Instructions per Core



(b) 256K Instructions per Core



(c) 512K Instructions per Core

Figure 5. Performance results for various test sizes

Table I
SPEEDUP OVER TSOTOOL FOR OUR SEQUENTIAL AND PARALLEL IMPLEMENTATIONS OF INFERRING EDGES

Inst. per Core	64K	128K	256K	512K	1M
Num. of tests	27	27	23	10	2
Alg. 4	15.09x	16.41x	14.51x	4.01x	3.08x
OMP 2	29.31x	31.49x	27.98x	7.52x	5.70x
OMP 4	53.45x	57.34x	51.68x	14.19x	10.39x
GPU	57.90x	76.98x	72.32x	42.90x	45.16x

5c) show tests that all have 256K (512K) instructions per core, or a total of $n = 2^{20}$ ($n = 2^{21}$) instructions. The tests are independent, but are sorted from fastest to slowest (for the TSOtool baseline) for convenience. We compare four of our approaches to the TSOtool baseline:

1) A sequential approach that minimizes updates to RTVCs (Algorithm 4)
2) An OpenMP implementation using 2 threads (OMP 2, Algorithm 5)
3) An OpenMP implementation using 4 threads (OMP 4, Algorithm 5)
4) A GPU implementation (Algorithm 6)

Note that by inspection of the (logarithmic) y-axis of Figures 5b and 5c one can see that for a graph that is just twice as large, experiments can take significantly longer to run. It is evident from Figure 5 that TSOtool spends excessive time updating RTVCs. Our alternative sequential method that lazily, rather than eagerly, updates RTVCs (as shown in Algorithm 4) shows substantial improvements over this baseline. Furthermore, since our algorithm facilitates parallelism, we attain additional performance improvements by using OpenMP and CUDA. In the more extreme cases, our GPU implementation is orders of magnitude faster than TSOtool.

It is interesting to note that although the results for TSOtool have been plotted in order of increasing execution time, our corresponding implementations do not necessarily share this behavior. For instance, the second slowest TSOtool test in Figure 5b executes much faster than the slowest TSOtool test for our implementations. This peculiarity is explained by the fact that this particular test has a larger portion of store instructions (79%) than the other tests of this size. A larger number of store instructions leads to more executions of the outer for loop of Algorithm 2 in comparison to other tests which also leads to a relatively greater number of calls to $update\_RTVCs()$. Since our approach in Algorithm 4 improves upon the previous approach in Algorithm 2 precisely by calling $update\_RTVCs()$ less frequently it makes sense that our approach would perform especially well for this particular test.

Table I shows the geometric mean speedup of our various approaches over TSOtool for each test size. Note that the number of tests decreases with test size due to industrial time

Table II
PARALLEL SPEEDUPS OVER ALGORITHM 4

Inst. per Core **Num. of tests**	64K 27	128K 27	256K 23	512K 10	1M 2
OMP 2	1.94x	1.92x	1.93x	1.88x	1.85x
OMP 4	3.54x	3.49x	3.56x	3.54x	3.37x
GPU	3.84x	4.69x	4.98x	10.70x	14.66x

Table III
APPLICATION SPEEDUP OVER TSOTOOL FOR OUR SEQUENTIAL AND PARALLEL IMPLEMENTATIONS

Inst. per Core **Num. of tests**	64K 27	128K 27	256K 23	512K 10	1M 2
Alg. 4	5.64x	5.31x	6.30x	3.68x	3.05x
OMP 2	7.62x	7.12x	9.05x	6.41x	5.58x
OMP 4	9.43x	8.90x	12.13x	10.81x	9.97x
GPU	10.79x	10.76x	15.47x	24.55x	37.64x

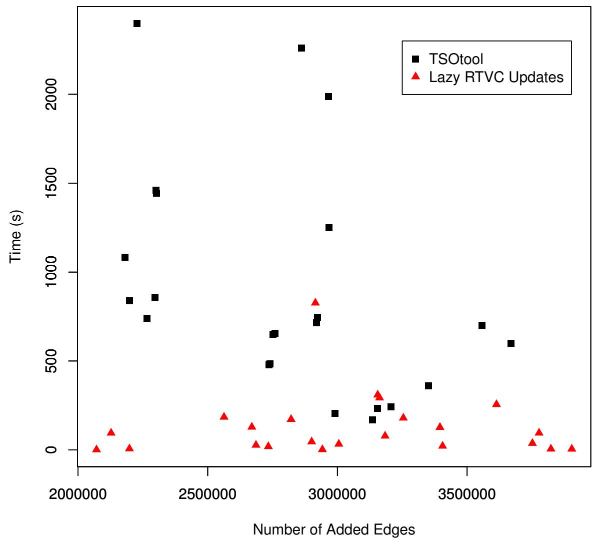


Figure 6. Scatter plot of edges added and performance

constraints when using TSOtool, providing motivation for our efforts. We can see that our reduction in the number of RTVC updates gives us at least a 3x speedup over TSOtool sequentially. Furthermore, since our methodology facilitates parallelism, we see the additional benefit of parallelism, as shown in Table II. Table II shows the precise performance gains for inferring edges in parallel. The speedups for parallel methods in Table I show total speedup over the TSOtool baseline, which includes the speedup of simply using the more efficient sequential algorithm as well as parallel performance benefits. Table II extracts the parallel speedups over our more efficient sequential approach (Algorithm 4) to convey the benefits of parallelization alone. We can see that the OpenMP implementations approximately achieve 1.9x and 3.5x speedups using 2 threads and 4 threads, respectively, regardless of problem size. Our GPU implementation consistently does better than the OpenMP implementation; however, it doesn't perform substantially better than the OpenMP implementation until the problem size is sufficiently large. Compared to our sequential approach, the GPU approach achieves more than an order of magnitude speedup for graphs with 512K instructions per core or greater.

Recall from Figure 1 that inferring edges is just one portion of the overall design flow for the memory consistency verification problem. Thus, we need to show that our efforts in improving the performance of inferring edges also improves overall application performance, else our efforts were not properly focused. Table III shows the overall application speedup of our sequential and parallel approaches over TSOtool. These speedups quantify the additional throughput one can achieve in terms of the number of tests run by using our approaches. For instance, one can run approximately 37 times as many tests with 1M instructions per core using our GPU implementation compared to what is done today in the same amount of time using TSOtool. This increase in throughput is important because it allows for greater coverage in testing. Running additional tests allows one to check for a greater variety of errors in the memory subsystem.

These speedups are also substantial in terms of absolute time and performance. One of the larger tests we experimented with required over nine hours of total application execution time using TSOtool. Using our GPU approach, we were able to finish the same test in under ten minutes. As a metric of absolute performance, we measured the GPU memory throughput of our larger tests to average 28.19 GB/s and reach a peak of 35.15 GB/s, showing that our implementation, although simplistic, efficiently utilizes the processor. Considering the extreme sparsity and irregular structure of the program execution graphs that we tested, achieving a significant percentage of the peak memory bandwidth of the processor is challenging [11].

Figure 6 provides some additional performance insights. We compare both sequential methods of updating RTVCs, eager (TSOtool, Algorithm 2) and lazy (Algorithm 4), in terms of performance and the number of inferred edges. A consequence of using Algorithm 4 and infrequently updating RTVCs is that stale RTVC values can lead to unnecessarily inferred edges. However, in some cases we can see that using Algorithm 4 actually results in *fewer* inferred edges. This result can occur when a store's RTVC to a certain vproc incrementally moves toward the beginning of the vproc during an iteration of Algorithm 2. In contrast, Algorithm 4 will skip the intermediate locations of the RTVC, thus neglecting to infer any edges at those intermediate locations.

Table IV
METADATA REGARDING THE TWELVE LARGEST TEST CASES

$\mathbf{n = \lvert V \rvert}$	$\mathbf{m = \lvert E \rvert}$	**TSOtool Inferred**	**Iter.**	**ST/LD/BAR (%)**
2,097,963	3,799,254	4,487,224	5	76/24/0
2,098,219	3,686,624	4,411,887	4	79/21/0
1,977,832	4,453,340	5,179,108	5	46/53/1
2,097,741	3,875,831	4,635,852	7	77/23/0
1,936,321	5,109,990	5,236,671	5	44/54/2
2,098,321	2,491,062	4,257,077	6	80/20/0
2,097,809	4,321,793	4,404,753	7	78/21/1
1,871,831	3,660,617	4,861,044	6	44/54/2
2,097,809	4,434,120	4,418,555	5	80/20/0
2,004,180	4,354,887	5,530,123	6	45/54/1
4,195,405	6,934,725	9,338,902	7	76/23/1
4,194,961	7,960,567	8,963,281	6	78/22/0

Although it was shown in Section IV that inferring additional edges or neglecting to infer these intermediate edges does not invalidate the program's output, it is reasonable to be concerned about the performance implications of the unnecessary work of inferring additional edges. Figure 6 shows that the amount of execution time for tests run using Algorithm 4 is independent of the number of edges inferred. In fact, the (Pearson product-moment) correlation coefficient between these two vectors is just 0.007, supporting that the data are largely unrelated. Surprisingly, for tests run using Algorithm 2 we actually see a slight *inverse* correlation between execution time and the number of inferred edges: -0.423. It is clear that other characteristics, such as the size of the graph, the frequency of dependencies between instructions, and the distribution of instruction types have a more profound impact on performance. For our largest tests we saw up to 36% additional edges inserted by Algorithm 4 compared to that of Algorithm 2; nevertheless, our speedups still justify the redundant work.

Table IV shows additional information regarding our twelve largest test cases. The first two columns present the number of vertices and edges for each test case, respectively, before any edges are inferred. It is evident that the initial graphs are substantially sparse, as $m < 3n$. The third column shows the number of edges that are inferred using the algorithm from TSOtool. We place this data side by side with the number of iterations (shown in the fourth column) because these two columns contrast the number of calls to $update\_RTVCs()$ made by TSOtool and our approaches. We can see that using TSOtool, $O(n)$ calls to $update\_RTVCs()$ are made for these test cases, and these excessive calls will only become increasingly detrimental to performance as the graphs tested continue to grow. On the contrary, our approach requires at most seven calls to $update\_RTVCs()$. The speedup achieved by our approach isn't directly proportional to this reduction in the number of updates because each iteration of the algorithm requires searching through the vprocs of each store in the graph, regardless of the number of updates that occur. Although the number of iterations tends to grow with the size of the graph, the rate at which the number of iterations grows is tremendously small. Hence, the number of RTVC updates that we perform scales very well with the size of the graph. Finally, the fifth column of Table IV breaks down the percentage of store, load, and barrier instructions found within each test. Note that tests with the same initial graph and proportion of ST/LD/BAR instructions can still vary by quite a bit as the number of distinct memory locations and virtual processors may differ.

VII. CONCLUSIONS

This paper discusses several parallel methodologies for verifying the memory consistency of architectures with relaxed memory models. We provide an alternative approach to using reverse time vector clocks that chooses to update this data structure after every iteration of inferring edges rather than after every edge insertion as was done previously. This approach reduces the work complexity of inferring edges, which we have shown to be the dominating factor in terms of performance for the entire verification process. Additionally, this approach simplifies the parallelization of consistency verification as a direct parallelization of our new approach requires significantly less communication than a direct parallelization of the previous approach. For a set of 89 tests in use at NVIDIA, we achieved geometric mean speedups of 12.74x, 44.95x, and 64.28x over the best existing approach for inferring edges for our sequential, OpenMP, and GPU implementations respectively. For the twelve largest test cases, our GPU implementation was able to achieve an average application speedup of 26.36x, reducing execution time from over nine hours to under ten minutes in one instance.

A number of insights regarding the computation of parallel graph algorithms have appeared in recent literature. Frameworks such as Ligra [23] and Galois [20] alleviate the difficulty of programming graph algorithms on shared memory architectures without sacrificing performance. Heavily optimized GPU implementations of specific algorithms have also been developed, for algorithms such as Breadth-First Search [19], Single-Source Shortest Paths [8], and Betweenness Centrality [17]. This collection of work tends to focus on graph algorithms that are traversal-based; it remains unclear if these insights can be directly applied to non traversal-based algorithms such as the algorithms discussed in this paper. We consider a more general approach to the design of shared memory parallel graph algorithms to be an intriguing area of future work.

ACKNOWLEDGMENT

This research was developed, in part, with funding from the Defense Advanced Research Projects Agency (DARPA).

The views, opinions, and/or findings contained in this article are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government. Special thanks to Shankar Govindaraju and Tom Hart for clarifications on their ARM-based implementation of TSOtool.

REFERENCES

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

[2] A. Arvind and J.-W. Maessen, "Memory Model = Instruction Reordering + Store Atomicity," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[3] H. W. Cain, M. H. Lipasti, and R. Nair, "Constraint Graph Analysis of Multithreaded Programs," in *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2003.

[4] Y. Chen, Y. Lv, W. Hu, T. Chen, H. Shen, P. Wang, and H. Pan, "Fast Complete Memory Consistency Verification," in *IEEE 15th International Symposium on High Performance Computer Architecture (HPCA)*, 2009.

[5] N. Chong and S. Ishtiaq, "Reasoning About the ARM Weakly Consistent Memory Model," in *Proceedings of the ACM SIGPLAN Workshop on Memory Systems Performance and Correctness (MSPC): Held in Conjunction with (ASPLOS)*, 2008.

[6] A. E. Condon, M. D. Hill, M. Plakal, and D. J. Sorin, "Using Lamport Clocks to Reason About Relaxed Memory Models," in *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture (HPCA)*, 1999, pp. 270–278.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein *et al.*, *Introduction to Algorithms*. MIT press Cambridge, 2001, vol. 2.

[8] A. Davidson, S. Baxter, M. Garland, and J. Owens, "Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths," in *Proceedings of the 28th International Parallel and Distributed Processing Symposium (IPDPS)*, 2014, pp. 349–359.

[9] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering," in *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, 1988, pp. 56–66.

[10] S. Hangal, D. Vahia, C. Manovit, and J.-Y. J. Lu, "TSOtool: A Program for Verifying Memory Systems Using the Memory Consistency Model," in *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[11] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA Graph Algorithms at Maximum Warp," in *Proceedings of the ACM Symposium on Principles and Practice of Parallel Programming*, 2011, pp. 267–276.

[12] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.

[13] A. Landin, E. Hagersten, and S. Haridi, "Race-free Interconnection Networks and Multiprocessor Consistency," in *Proceedings of the 18th Annual International Symposium on Computer Architecture (ISCA)*, 1991, pp. 106–115.

[14] C. Manovit, "Testing Memory Consistency of Shared-memory Multiprocessors," Ph.D. dissertation, Stanford, CA, USA, 2006.

[15] C. Manovit and S. Hangal, "Efficient Algorithms for Verifying Memory Consistency," in *Proceedings of the 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2005.

[16] ——, "Completely Verifying Memory Consistency of Test Program Executions," in *Proceedings of the Twelfth International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2006, pp. 166–175.

[17] A. McLaughlin and D. A. Bader, "Scalable and High Performance Betweenness Centrality on the GPU," in *Proceedings of the 26th ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis (SC)*, 2014.

[18] W. Mclendon III, B. Hendrickson, S. J. Plimpton, and L. Rauchwerger, "Finding Strongly Connected Components in Distributed Graphs," *Journal of Parallel and Distributed Computing*, vol. 65, no. 8, pp. 901–910, 2005.

[19] D. Merrill, M. Garland, and A. Grimshaw, "Scalable gpu graph traversal," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2012, pp. 117–128.

[20] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The Tao of Parallelism in Algorithms," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2011, pp. 12–25.

[21] S. Qadeer, "Verifying Sequential Consistency on Shared-Memory Multiprocessors by Model Checking," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 14, no. 8, pp. 730–741, 2003.

[22] A. Roy, S. Zeisset, C. J. Fleckenstein, and J. C. Huang, "Fast and Generalized Polynomial Time Memory Consistency Verification," in *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*. Springer-Verlag, 2006.

[23] J. Shun and G. E. Blelloch, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2013, pp. 135–146.