ScoRD: A Scoped Race Detector for GPUs

Aditya K. Kamath* Alvin A. George* Arkaprava Basu
Department of Computer Science and Automation
Indian Institute of Science
Bangalore, India
{adityakamath, alvingeorge, arkapravab}@iisc.ac.in

***Abstract*—GPUs have emerged as a key computing platform for an ever-growing range of applications. Unlike traditional bulk-synchronous GPU programs, many emerging GPU-accelerated applications, such as graph processing, have irregular interaction among the concurrent threads. Consequently, they need complex synchronization. To enable both high performance and adequate synchronization, GPU vendors have introduced scoped synchronization operations that allow a programmer to synchronize within a subset of concurrent threads (a.k.a., *scope*) that she deems adequate. Scoped-synchronization avoids the performance overhead of synchronization across thousands of GPU threads while ensuring correctness when used appropriately. This flexibility, however, could be a new source of incorrect synchronization where a race can occur due to insufficient scope of the synchronization operation, and not due to missing synchronization as in a typical race.**

We introduce ScoRD, a race detector that enables hardware support for efficiently detecting global memory races in a GPU program, including those that arise due to insufficient scopes of synchronization operations. We show that ScoRD can detect a variety of races with a modest performance overhead (on average, 35%). In the process of this study, we also created a benchmark suite consisting of seven applications and three categories of microbenchmarks that use scoped synchronization operations.

***Index Terms*—Graphics processing units, Parallel programming, Software debugging**

I. INTRODUCTION

Today, Graphics Processing Units (GPUs) serve as the primary computing platform for a wide range of application domains. The massive data parallelism of GPUs had initially been leveraged by highly-structured parallel tasks such as matrix multiplication, where the interactions among the concurrent threads are regular and relatively infrequent. Such regular applications could use the GPU's coarse-grain bulk synchronous model [1] of execution very well, with little need for advanced synchronization operations.

In recent times, however, a broader range of application domains such as graph processing, deep learning, weather modeling, data analytics, computer-aided-design, and computational finance have started using GPUs [2]. Many of these emerging applications often entail irregular interactions among the concurrent threads. To fulfill the synchronization needs of such applications, modern GPU programming languages and hardware have enabled semantically rich synchronization primitives such as various flavors of atomic, fence, barrier, and acquire/release operations.

*Authors contributed equally.

However, it is hard to efficiently support globally visible synchronization operations across thousands of concurrent threads in a GPU. Fortunately, global synchronization is often unnecessary in GPU programs [3]–[5]. Popular GPU programming languages – CUDA and OpenCL – expose a hierarchical programming paradigm. They group threads into threadblocks (workgroup in OpenCL); many threadblocks make up a grid. Further, the hardware typically schedules a group of 32 to 64 threads, called warp or wavefront, to execute in a SIMT (single-instruction multiple-thread) fashion. Consequently, GPU programs are often naturally written in a way that requires communication only within a subset of threads at a given level in the hierarchy.

GPU programming languages, thus, expose synchronization operations with non-global side effects. Both CUDA and OpenCL expose different *scope* qualifiers that can be used with synchronization operations. A scope identifies the subset of concurrent threads that are guaranteed to observe the effect of the synchronization. CUDA exposes three scopes – block, device and system [6]. For example, an atomic read-modify-write (RMW) operation (a.k.a., atomic) with block scope is only guaranteed to affect threads within the same threadblock as the thread executing the RMW. If the scope were device, all the threads in the kernel running on the GPU would observe the effect of the operation. An even wider scope – system – affects all threads of a program spread over the CPU and multiple GPUs. OpenCL supports similar scopes too [7].

GPU hardware resources are also arranged in a hierarchy that naturally lends itself well to scoped synchronization. For example, threads belonging to the same threadblock are scheduled on a single streaming multiprocessor (SM). Threads in an SM share an L1 cache and a scratchpad memory. Threads within a threadblock can thus communicate much faster than those in different threadblocks, executing on different SMs. Consequently, a synchronization operation with block scope executes faster than one with device or system scope.

A GPU programmer can use scoped synchronization operations to efficiently synchronize across only a subset of threads as per the semantic requirement of the program. While scoped operations add an essential capability for balancing the need for synchronization and performance, they also open up a new dimension that the programmer needs to worry about for writing *correctly synchronized* programs. We declare the scope of a synchronization operation to be *insufficient* if it does *not* encompass both the producer and the consumer that the

operation intends to synchronize. We call races that arise due to the insufficient scope of a synchronization operation as *scoped races*. For example, let us assume two threads perform an atomicExch operation on the same variable with block scope. If these threads belong to different threadblocks, then they may not observe the effect of each other's atomic operation. Consequently, a scoped race will arise.

Decades of prior research have demonstrated that writing a correct multi-threaded program is hard [8]–[10]. Consequently, there has been a plethora of software tools and hardware support for detecting races in multi-threaded CPU programs [8], [10]. Arguably, writing a correct GPU program with orders of magnitude more threads is even harder, especially if it involves irregular interactions between threads. Realizing the need for tools to detect incorrectly synchronized GPU programs, researchers have proposed software tools and hardware support in recent times [11]–[14].

Unfortunately, the current set of tools falls short in one way or the other. More importantly, they mostly ignore scoped races. Tools such as NVIDIA's CUDA-Racecheck [15], GRace [16], or GMRace [13] limit themselves to detecting races that can occur among the threads within a threadblock via the scratchpad memory, and ignore the relatively harder class of races via global memory. More recently, researchers have proposed dynamic binary instrumentation, e.g., Barracuda [12], and/or static compilation time hints, e.g., CURD [11], to detect races in global memory. However, being pure software tools, they typically incur 2×-1000× performance overhead. More importantly, they largely ignore scoped races. For example, Barracuda considers scopes in only fence operations while ignoring them for other synchronization operations such as atomics. Researchers have also recently proposed hardware support to detect races in GPU programs, but it completely ignores scoped races [14]. The shortcomings of these works are summarized in Table VIII of Section VII.

In this work, we thus propose ScoRD (Scoped Race Detector), an efficient hardware-based detector for scoped races, and any other global memory races in GPU programs. While we focus on CUDA to make discussions concrete, ScoRD's design is not limited to CUDA only. In CUDA, both atomic RMW and fence operations can be qualified with scopes, and consequently, their use can cause scoped races. ScoRD extends the well-known happens-before race detection [17] with the notion of scope to detect scoped races due to atomics and fences. Further, the CUDA guidebook suggests [18]–[20] that programmers can combine atomic RMWs and fences to constitute lock and unlock operations in the absence of acquire/release operations in current versions of CUDA*. ScoRD thus dynamically infers lock and unlock operations and extends lockset-based mechanisms [8], [10] with the notion of scope to detect races on data items protected by locks. While we focus on scoped races, ScoRD detects global memory races due to missing synchronizations too, as detection for scoped races naturally subsumes any global race detection.

ScoRD introduces small hardware state (less than 3KB) to hold information on active synchronization operations along with the logic for detecting races in the GPU. As GPUs often concurrently execute thousands of threads, tracking happens-before interactions [22] among each pair of threads is not scalable, unlike in CPU. ScoRD thus, further keeps metadata for each unit of global memory (here, at the granularity of 4 bytes) to track the identity of the last accessor of the memory location along with relevant synchronization and scope information. This metadata is kept in the global memory of the GPU. A naive approach, however, would incur significant memory overheads due to metadata (e.g., up to 2×). We observe that races typically occur among memory accesses that happen relatively close to each other in time. Further, only a small fraction of allocated memory participates in a race. We thus only keep the metadata for recently accessed memory addresses, using a direct-mapped software cache. This helps us reduce metadata overhead by 16×, to a reasonable 12.5% without greatly sacrificing the accuracy of race detection.

A challenge in thoroughly evaluating ScoRD, though, is the lack of an open-source benchmark suite with copious use of scoped synchronization operations. As newer GPUs support an ever-increasing number of concurrent threads, global synchronization is becoming costlier. Consequently, CUDA and OpenCL are continually enhancing support for scoped operations. However, open-source applications are understandably slow to catch up to the use of evolving scoped synchronization operations. We thus created the ScoR benchmark suite, comprising of seven applications and thirty-two microbenchmarks. The applications in ScoR can be configured to omit proper synchronization operations to create up to twenty-six unique races. We use ScoR to perform a thorough evaluation of ScoRD.

ScoRD incurs low performance overhead (35% on average) with no false positives. Moreover, ScoRD can be turned on only during software testing or debugging and can be turned off during production run to avoid overheads.

In summary, we make the following contributions.

- We detail ScoRD, a hardware-based scoped race detector for reporting races in GPU programs with low-performance overhead. To the best of our knowledge, ScoRD is the first of its kind.
- We create ScoR benchmark suite containing several applications and microbenchmarks that make use of scoped synchronization primitives. We open-sourced ScoR to aid future research†.

II. Background and the Baseline

To appreciate this work, some background on the GPU's execution hierarchy, its synchronization operations, and a bit about its memory consistency model would be useful.

*A recent version of PTX, i.e., PTXv6.0 introduced acquire/release instructions, but the latest CUDA specification (v10) is silent on how to use them as of yet [6], [21].

†Available at https://github.com/csl-iisc/ScoR/

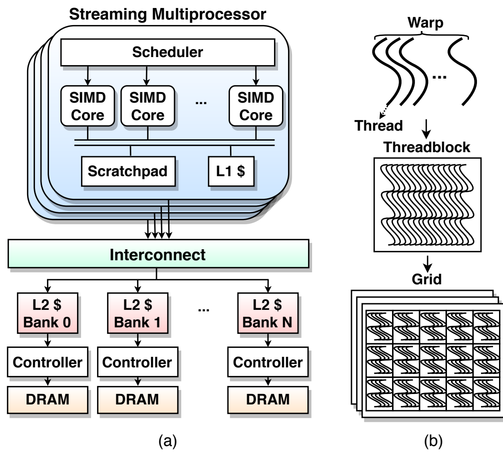

Fig. 1: Baseline GPU system architecture.

A. Execution hierarchy in a GPU

GPUs are designed for massive data-parallel processing that operates on hundreds to thousands of data elements concurrently. A GPU's hardware resources are organized in a hierarchy to keep its vast parallelism tractable.

Figure 1 (a) depicts the architecture of a typical GPU hardware. Streaming multiprocessors (SMs) are the basic computational blocks of a GPU, with typically around 8 to 64 SMs in a GPU. Each SM includes multiple single instruction, multiple data (SIMD) units, which have multiple *lanes* of execution (e.g., 16-32). A SIMD unit executes a single instruction across all lanes in parallel. The memory resources of a GPU are also arranged hierarchically. Each SM has a private L1 data cache and a scratchpad that is shared across the SIMD units within the SM. When several data elements being requested by a SIMD instruction reside in the same cache line, a hardware coalescer combines the requests into a single cache access to gain efficiency. A larger L2 cache is shared across all the SMs through an interconnect.

GPU programming languages, such as OpenCL and CUDA, expose a hierarchy of execution groups to the programmer that follows the hierarchy in the hardware (Figure 1 (b)). In CUDA parlance, a thread is akin to a CPU thread and is the smallest execution entity that runs on a single lane of a SIMD unit. A group of threads, typically 32, forms a warp, which is the smallest hardware-scheduled unit of work that executes in SIMT fashion. Several warps make up a threadblock, which is programmer visible. All threads in a threadblock are scheduled on the same SM. Finally, work on a GPU is dispatched at the granularity of a grid, which comprises of several threadblocks.

B. Synchronization in GPUs

To make the discussion on synchronization concrete, we will pivot around CUDA. However, most of these concepts are equally applicable to OpenCL too.

In the early days, GPUs were primarily used for regular bulk-synchronous compute tasks. Consequently, one of the primary and often used synchronization primitives is a barrier (e.g., __syncthreads in CUDA). A barrier ensures that threads wait until all threads in the threadblock have reached the barrier, and all global and shared memory accesses made by these threads before the barrier are visible to all threads in the block. While a barrier acts as an execution barrier across the threads in the block and also enforces ordering of memory accesses, a memory fence (e.g., __threadfence) does only the latter. Specifically, a fence ensures that all writes to all memory made by the calling thread before the fence is visible to other threads.

While fences are necessary to ensure that the intended consumer of a data item observes the latest value, it may not be sufficient alone. For example, after completing a store followed by a fence, one may expect that other threads reading from the same location would obtain the updated value. However, this may not be the case in modern GPUs, since upper-level caches (e.g., L1 caches) and buffers are not kept coherent by hardware, unlike in CPUs. Therefore, while the store may have reflected on the shared cache, a consumer thread may have a stale copy of it in its local L1 cache. The onus of avoiding such stale reads is with the programmer. Specifically, CUDA provides the *volatile* qualifier that ensures memory operations bypass non-coherent caches and intermediate buffers. These qualified memory operations are referred to as *strong* operations by NVIDIA. In fact, CUDA programming guide suggests that fences guarantee ordering only for strong operations [6].

CUDA also supports atomic read-modify-write operations of various flavors. For example, atomicExch allows a thread to read a value stored in the memory and write a new value to that address such that no other thread can interfere until the entire operation is complete. Atomic operations are often used to create locks. The atomic operations in CUDA are *relaxed* in nature; i.e., they do not enforce any ordering guarantees. For example, an atomic does not ensure that writes occurring before it would be visible to other threads. Thus, lock/unlock operations in CUDA comprise of an atomic (for updating the lock variable) and a fence (for ordering). It is worthwhile to note that atomics are inherently *strong* operations since they take effect at the shared level of cache, bypassing possibly incoherent intermediate caches.

In a recent version of PTX, v6.0, NVIDIA also added support for two more synchronization operations – acquire and release. An acquire operation makes the effect of memory operations (e.g., loads/stores) from other threads visible to operations after the acquire in the current thread. A release operation makes the effect of operations by the thread executing the release visible to other threads. Acquire and release operations are typically used for lock and unlock operations, respectively. While earlier versions lack these instructions, acquire/release can be synthesized using atomic and fence operations. NVIDIA states that an acquire pattern is equivalent to an atomic operation on the lock variable (e.g., atomicCAS) followed by a fence [20], [21], [23]. A release can be composed of a fence followed by an atomic operation (e.g., atomicExch).

Scoped synchronization: Unlike in a CPU, a GPU typically has tens of thousands of concurrent threads. Consequently, global synchronization across all threads is slow in a GPU.

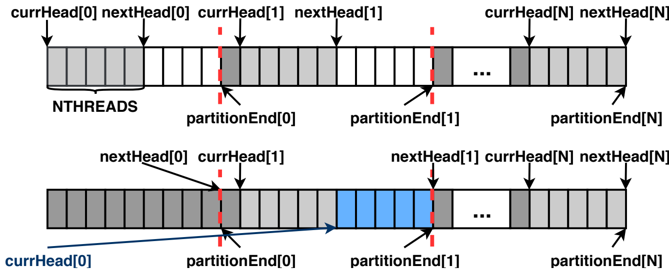

Fig. 2: Work stealing. Effect of stealing shown in blue.

Further, it is often unnecessary to synchronize across all threads in a kernel, owing to the GPU's hierarchical programming paradigm. Thus, GPU vendors have enabled the ability to synchronize across a subset of concurrent threads. For example, in CUDA, atomic and fence operations support three different *scopes* – block, device and system. An operation performed with a given scope is only guaranteed to be visible to threads that fall within the scope of that operation. For example, a fence performed with block scope is only guaranteed to affect the threads within the threadblock to which the issuing thread belongs. A device-scope operation is only guaranteed to affect all threads in a GPU for a given program. If a system has multiple GPUs and a program spans multiple GPUs, then system scope affects threads across different GPUs, as well as the CPU belonging to a given program. Like CUDA, OpenCL also supports very similar scopes for synchronization operations. In this work, we ignore the system scope without any loss of generality.

C. GPU memory consistency models

Memory consistency models define which values from memory operations are illegal and which are legal [24]. A weaker memory model allows a larger set of possible outcomes from concurrent operations, while a stricter model allows a smaller number of reordering of memory operations. Synchronization operations enable a programmer to enforce desired ordering that may not be implicitly enforced by the consistency model of the system.

Several published works on GPU memory consistency models take scoped synchronization operations into account [3]–[5]. In this work, we assume the Heterogenous-Race-Free (HRF)-relaxed-indirect memory consistency model [5], and our simulation framework enforces this memory model. At a high level, HRF-indirect allows transitivity of scopes – if a thread A is synchronized with a proper scope with thread B and later thread B synchronizes with thread C, then thread A and thread C have effectively synchronized as well. Scope-inclusion allows synchronization without requiring the scopes of operations in participating threads to be exactly the same, as long as the scopes of both the consumer and producer threads include each other. We refer readers to [3], [5] for an an in-depth discussion on GPU consistency models.

III. Scoped GPU races and ScoR benchmark suite

A race on a global memory location occurs if two threads perform conflicting accesses (i.e., at least one of them write) to the same memory location and if the accesses are not separated by *appropriate* synchronization operations [25]. In the traditional sense, e.g., in the CPU, races occur due to the complete absence of necessary synchronization operations. In GPU programs, however, a race can occur even if two conflicting accesses are separated by synchronization operation(s) but of *insufficient* scope – a.k.a. scoped races [3].

We first discuss different classes of scoped races possible in GPU programs with the help of examples. We later describe the benchmark suite we created that makes use of scoped synchronization operations and can be configured to introduce scoped races.

A. Classification of Scoped GPU races

To make the discussion concrete, we will focus on synchronization operations available in CUDA v.8.0, and PTX v.5.0. There, both atomic and fence operations can be qualified with scope. Further, these two operations can be combined to create lock/unlock operations providing mutual exclusion to code regions [18]–[20]. The use of insufficient scopes in any of these can create a scoped race. Therefore, a total of three types of scoped races are possible as follows.

Scoped race due to atomic operation: If the producer thread of a data item updates a global memory location using a scoped atomic operation, but the consumer is outside that scope, we declare a scoped race. To demonstrate how this race can creep in, while optimizing code, we explore the use of scoped atomics in a real application.

Let us consider a well-known graph processing algorithm such as graph coloring, where the objective is to assign a color to each node of a graph such that no two neighboring nodes have the same color. In the implementation, each thread is tasked to assign a color to one of the vertices. The work of coloring thousands of vertices is equally partitioned among the available threadblocks at the start of the execution. Typically, the number of vertices in a partition far exceeds the number of threads in a threadblock (e.g., 256), and thus, a threadblock must iterate multiple times to color the vertices in its partition. The number of vertices colored in each iteration is equal to the number of threads in a threadblock (NTHREADS). This is pictorially depicted in the top part of Figure 2. The array named partitionEnd[] holds the index of the end of each partition in the global array of vertices. The number of entries in the array is equal to the number of threadblocks. The array currHead[] holds the starting index of the set of vertices that are being colored in the current iteration, while nextHead[] holds that for the next iteration.

The amount of work to color a vertex varies depending on the number of edges that are incident upon it. Consequently, threadblocks may take different amounts of time to color vertices in their respective partitions. To reduce overall execution time, a threadblock that finishes early can *steal work* from another

```
1  __device__ int getWork(...)
2  {
3    if(tid != 0) // Only lead thread assigns work
4      return -1;
5    // Get work from own local partition
6    currHead[blockId] =
7      atomicAdd(&nextHead[blockId],
8      NTHREADS); //device scope
9    // Work left in own partition?
10   if(currHead[blockId] < partitionEnd[blockId])
11     return currHead[blockId];
12   // Otherwise steal work
13   int victimBlock = getPartitionToStealFrom();
14   if(victimBlock == -1) // Check if successfully stole
15     return -1; //No work
16   currHead[blockId] =
17     atomicAdd(&nextHead[victimBlock],
18     NTHREADS);
19   if(currHead[blockId] <
20     partitionEnd[victimBlock])
21     return currHead[blockId];
22   return -1; //No work left
23 }
```

(a) Correct code (non-racey).

```
1  __device__ int getWork(...)
2  {
3     if(tid != 0) // Only lead thread assigns work
4      return -1;
5    // Get work from own local partition
6    currHead[blockId] =
7      atomicAdd_block(&nextHead[blockId],
8      NTHREADS); //block scope
9    // Work left in own partition?
10   if(currHead[blockId] < partitionEnd[blockId])
11     return currHead[blockId];
12   // Otherwise steal work
13   int victimBlock = getPartitionToStealFrom();
14   if(victimBlock == -1) // Check if successfully stole
15     return -1; //No work
16   currHead[blockId] =
17     atomicAdd(&nextHead[victimBlock],
18     NTHREADS);
19   if(currHead[blockId] <
20     partitionEnd[victimBlock])
21     return currHead[blockId];
22   return -1; //No work left
23 }
```

(b) Racey code due to insufficient scope.

Fig. 3: Use of scoped atomics in work stealing.

```
1  __global__ void reductionKernel(...)
2  {
3    ...
4    // Update array used within block
5    sdata[tid] = mySum + sdata[tid + 64];
6    __threadfence_block();
7    lock[tid] = 1
8    ...
9    while(lock[tid + 32] != 1);
10   __threadfence_block();
11   sdata[tid] = mySum + sdata[tid + 32];
12   ...
13   if (tid == 0) // Update global array with final sum
14     g_odata[blockIdx.x] = sdata[0];
15   __threadfence(); //device scope
16   //Scoped race if __threadfence_block used
17   ... // Wait for all blocks to finish
18   reduce(g_odata);
19 }
```

Fig. 4: Use of (non-racey) scoped fence.

```
1  __global__ void searchTree(...)
2  {
3    while(atomicCAS_block( // Acquire block-scoped lock
4      &localStack.lock, 0, 1) != 0);
5    __threadfence_block();
6    Node parent = getNode(localStack); // Remove node
7    localStack.top++;
8    localStack.work--;
9    __threadfence_block(); // Release block-scoped lock
10   atomicExch_block(&localStack.lock, 0);
11   ...
12   while(0 != atomicCAS( // Acquire global lock
13     &globalStack[id].lock, 0, 1));
14   __threadfence();
15   Node parent = getNode(globalStack[id]);
16   globalStack[id].top++;
17   globalStack[id].work--;
18   __threadfence(); // Release global lock
19   atomicExch(&globalStack[id].lock, 0);
20 }
```

Fig. 5: Use of scoped lock/unlock (acquire/release pattern).

block's partition. The bottom part of Figure 2 depicts how a threadblock (here, threadblock 0) that finished coloring vertices in its partition steals work (a set of NTHREADS vertices) from a partition belonging to threadblock 1.

Figure 3a shows pseudo-code for how a leader thread (here thread id = 0) gets the set of vertices to be assigned colors next, at the end of each iteration. The lines 6-8 shows how the leader thread updates the currHead with the present value of nextHead and atomically updates nextHead. Notice that device scope (default) is used for the atomic. The lines 10-11 checks if there are any more vertices left to color in the threadblock's original partition. If not, the leader thread would steal work from another partition. It first determines which threadblock's (victimBlock) partition to steal work from (line 13). It then performs the stealing by updating nextHead[] of victimBlock using device scope atomic operation (line 16-18). Finally, it validates the stolen set before returning.

One may incorrectly presume that the use of a block-scope atomic is sufficient when updating the nextHead[] if *no* work-stealing is performed (lines 6-8 in Figure 3b). The leader thread updates a variable used by threads within its block. This is the common case as stealing happens only under the load imbalance. However, this could lead to a subtle race if another threadblock attempts to steal from the given block's partition (i.e., the victimBlock) at the same time when the victim itself is assigning work from its own partition. The update by the leader thread of the victimBlock would not be visible to the stealing threadblock. This shows how subtle scoped races can seep into the code while performance-optimizing an application.

Scoped race due to fence: After updating a global memory location with the data item, if the producer thread executes a scoped fence where the consumer is outside that scope, a scoped race occurs. This is because the update by the producer may not be observed by its intended consumer.

Let us consider an implementation of a reduction operation that sums an array of a large number of integers to a single value (relevant pseudo-code in Figure 4). Each threadblock is responsible for calculating a partial sum of a sub-array of elements of size twice the number of threads in a block. For example, if there are 256 threads in a block, then each sub-array will be of size 512 entries. In the first step, each thread will sum one element from the first half of the sub-array with the corresponding element in the second half. After this step, it will reduce the sub-array to partial sums with 256 elements. Next, 128 threads in the block will similarly compute partial sums with 128 entries, and so on. The lines 5-6 shows how 64 threads reduce an array with 128 element to 64 elements. The fence ensures that threads in the block observe the updated partial sums before 32 threads start computing in the next step. The block scope is enough as only threads in the same threadblock consume the partial sums.

The code in lines 7-10 ensures that each thread waits for the thread in the previous step whose results it will use in the current step. Finally, once a threadblock finishes computing on its subarray, it adds the partial sum to the global array (g_odata), as shown in lines 13-14. Since other threadblocks can consume values in the global array, a device-scoped fence is needed. A block-scoped fence would lead to a scoped race (not shown).

Scoped race due to lock/unlock: If two threads attempt to update the same global memory location within their critical sections, but the scope of the lock/unlock operations do not include both the threads, a scoped race occurs. As per CUDA programming guide [18]–[20], a lock operation can be constructed by an atomic on a lock variable followed by a fence. Similarly, the unlock operation can be constructed by a fence followed by an atomic on the lock variable. The scope of the lock/unlock operation is equal to the narrowest scope of its constituents.

Let us consider the example of unbalanced tree search (Figure 5). Here, each threadblock has a local and global stack where they place child nodes that are generated from a parent node. Each thread in a threadblock removes nodes from their local stack to produce child nodes based on a simple hash function. Since this procedure involves multiple steps that must be executed atomically (lines 6-8), the stack is locked using block scope (lines 3-5, 9-10) while nodes are removed. In case the local stack is empty, threads can attempt to remove nodes from the global stack of any threadblock. Since these are shared, device-scope locking must be used (lines 12-14, 18-19). A scoped-race would occur if these atomic operations or fences used block scope (not shown).

Relation between a barrier and a scoped race: We observe that races could *also* arise due to the complete absence of a fence, e.g., if the fence is missing in Figure 4, line 6. While a block-scoped fence is sufficient in this case, a barrier could have also prevented the race, since barriers also act as block-scope fences (Section II). However, barriers additionally synchronize progress of all threads in the block and, thus, are slower.

TABLE I: Description of microbenchmarks.

Sync. type	Racey tests	Non-racey tests	Description
Fence	2	4	A write to global memory followed by a read by another thread, with or without a __threadfence in between, of varying scopes.
Atomics	4	5	Atomic and non-atomic operations on global memory using varying scopes.
Lock / unlock	12	5	Loads/stores on global memory with or without lock/unlock (acquire/release) of varying scopes. Required __threadfence may also be missing.
Total	**18**	**14**	

TABLE II: Applications in the benchmark suite.

Benchmark	Description	Parameters	Races
Matrix Multiplication (MM)	Computes product of two large matrices. Uses scoped lock/unlock. Please refer to Figure 5 for details.	800 x 500 and 500 x 30 matrices	Scoped-lock.
Reduction (RED)	Computes reduction (sum) of a large array [26]. Uses differently scoped fences. Refer to Figure 4.	25.6M elements	Scoped-atomics and fences.
Rule 110 Cellular Automata (R110)	Computes Rule 110 of Cellular Automata over an array. Each thread updates an array location each iteration. Scope of fence used after iteration depends whether the element lies on the border of a block or not.	2.5M elements	Scoped-atomics and fences.
Graph Coloring (GCOL)	Assigns color to each graph vertex. Vertices and edges are distributed among blocks for processing [27]. Uses work stealing with scoped-atomics as seen in Figure 3.	30K vertices, 50K edges	Scoped-atomics.
Graph Connectivity (GCON)	Finds connected components of a graph. Vertices and edges are distributed among blocks for processing [28]. Uses work stealing with scoped-atomics as seen in Figure 3.	100K vertices, 150K edges	Scoped-atomics.
One-Dimensional Convolution (1DC)	Computes the convolution of a large array. Each thread does a single computation for an element, and updates memory using scoped atomics based on whether other blocks are updating the same location.	9 element filter, 1M elements	Scoped-atomics.
Unbalanced Tree Search (UTS)	Trees are constructed, using a simple hash function to decide the number of children a node has [29]. Each block has a local and global stack to keep pending nodes. Threads operate on nodes from their local stack with block scope or from any global stack with device scope.	120 trees, 9 levels, 3 avg. children ($\sim$1.2M nodes)	Scoped-atomics and lock.

B. The ScoR benchmark suite

Scoped operations are a relatively new synchronization concept. Unsurprisingly, any substantial number of open-source applications that use scoped-synchronization is yet to exist. At best, there has been a suite of microbenchmarks [30]. However, with the continued support of scoped operations in CUDA and OpenCL and slower global synchronization due to bigger GPUs, the use of scoped operations is likely to increase [4]. Many emerging GPU-accelerated applications, such as graph processing, demonstrate irregular interactions among threads that do not lend themselves well to a traditional bulk-synchronous execution [4]. The use of scoped-synchronization

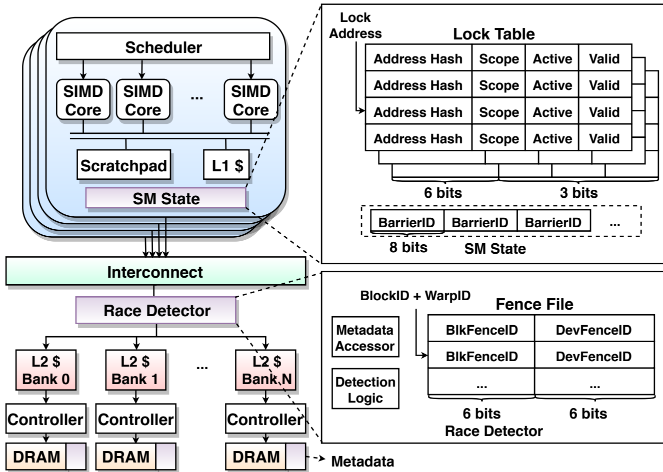

Fig. 6: ScoRD design diagram.

is essential for such applications to achieve both correctness and good performance.

We thus created a benchmark suite with seven applications and thirty-two microbenchmarks that use a variety of scoped synchronization operations, as discussed above. We call it the ScoR (Scoped Race) benchmark suite.

Table I and Table II describe the microbenchmarks and the applications, respectively. The microbenchmarks use only two threads to create both racey and non-racey conditions. These are useful for unit testing different race conditions and the accuracy of race detectors. Non-racey versions are useful to check for false positives. The applications are chosen to cover a wide variety of scoped-synchronization operations and, thus, potential scoped races. By default, each application is correctly synchronized but comes with configurable parameters that can introduce different types of scoped and non-scoped races.

IV. SCORD: A GPU SCOPED RACE DETECTOR

We design ScoRD[‡] – an accurate and efficient hardware-based GPU race detector. ScoRD detects scoped races and races in global memory due to missing synchronization operations.

ScoRD reports the instruction pointer and the data address of the memory instruction associated with the resultant race, either due to insufficient scope, or due to the absence of synchronization. It further reports whether the conflicting accesses were from the same threadblock (block-scope race) or different threadblocks (device-scope race), and the type of race, e.g., was it a race due to a missing fence/barrier or due to insufficient scope in the lock/unlock? This provides the programmer with enough context to identify bugs. ScoRD does not stop executing on detecting the first race. Instead, it attempts to complete the execution while accumulating information on detected races in a memory buffer. The user, therefore, gets information on multiple bugs in a single execution of a program.

[‡]Available at: https://github.com/csl-iisc/ScoRD/

At a high level, on a memory access (load/store or atomic), ScoRD first performs preliminary checks to find out if the access is *trivially race-free*. This captures simple yet common circumstances where races cannot exist (e.g., accesses to a location in program order) and acts as a filter to a more involved race detection. If the preliminary check fails, two types of checks are deployed to detect races. Happens-before relations [22] are checked to detect races due to insufficient scopes in atomics and fences or due to the absence of synchronizations. Further, ScoRD infers lock/unlock operations by monitoring atomics and fence pairs and uses the lockset-based algorithm [8], [10], extended to incorporate the notion of scopes, to detect races due to locking.

While lockset-based detection is restricted to detecting errors only in locking, happens-before-based detection applies to a broader range of synchronization operations. However, a race needs to manifest during execution for happens-before-based detection to detect it. In contrast, lockset-based detection can detect even potential races that may not manifest during an execution. ScoRD thus utilizes both approaches to catch different classes of scoped races as suitable.

Hardware modifications and the metadata: It is essential to first learn about the hardware changes and the required metadata to appreciate the inner workings of our race detector. Since in a typical GPU, the smallest scheduling entity is a warp, we maintain state and metadata at the warp granularity. Further, in Section VI, we discuss how ScoRD gracefully extends to designs where this may not always be true.

Figure 6 depicts the hardware modification for ScoRD. First, each SM is extended to keep track of the ID (an 8-bit counter) of the latest barrier (__syncthreads) executed by each threadblock (up to 8 per SM). SMs also keep a four-entry circular queue called the lock table for each warp (shown in the top right corner of Figure 6). The lock table is used for inferring lock (acquire pattern) and unlock (release pattern) and for tracking actively held locks by a warp. We later describe how this is used for lockset-based race detection.

ScoRD keeps metadata for each unit of global memory (by default, for every 4 bytes). The size of each metadata entry is 8-bytes long. Figure 7 shows the content of each entry. It keeps track of the threadblock ID and warp ID that last accessed the corresponding memory location. It keeps the latest device and block-scope memory fence ID executed by the warp that last wrote to the location. It further tracks the ID of the latest barrier executed by the threadblock that last accessed the location. We will shortly describe how these metadata contents are used to infer if an adequate synchronization operation has been executed between two conflicting accesses.

An entry in the metadata (Flags) also tracks the state of a memory location, e.g., if it has been modified, or has been accessed by different warps within a threadblock (BlkShared) or by different threadblocks in the kernel (DevShared). The flags track if a memory location is accessed using block or device scoped-atomic operations. In addition, the flags keep track of whether all accesses to the location since (re-) initialization have been strong operations (i.e., load/stores with

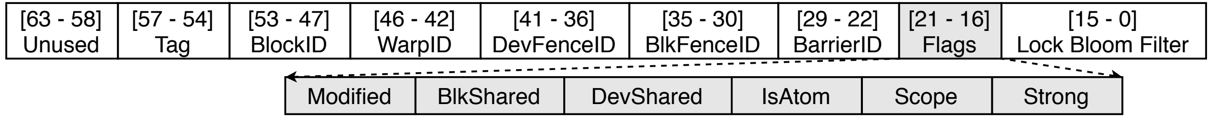

Fig. 7: ScoRD's in-memory metadata layout for one entry (8-bytes).

a volatile qualifier or atomic operations). These help identify the existence of conflicting accesses to a memory location. Last but not least, the Lock Bloom Filter keeps the summary of the locks held by the warp that last accessed the location.

The metadata corresponding to the entire device (on-board) memory of a GPU is pre-allocated in a contiguous physical memory region at boot. The Modified, BlkShared, and DevShared flags are all initialized to true for each entry. However, the size of the metadata entry (8 bytes) is double that of the granularity of tracking (4 bytes). Therefore, a naive implementation will add a 200% memory overhead for the metadata. In Section IV-B, we will describe our idea on how to reduce this overhead to 12.5% without any major impact on race detection. The Tag field in metadata entries will be used for this purpose.

Finally, the hardware race detector (right bottom of Figure 6) hangs off the interconnect that connects SMs to the L2 cache. It has three primary tasks. First, for each memory access, it loads the corresponding metadata entry. Next, ScoRD keeps a hardware fence file shared by all SMs to keep track of the IDs of the latest block-scope and device-scope fence operations executed by a given warp (6-bit counters each). The fence file is indexed by the combination of threadblock and warp ID. This helps detect if a warp has executed any fence since it last accessed a memory location (noted in metadata).

The race detector also houses the logic (next subsection) to detect races by consulting the access information (e.g., ID of the accessing warp, threadblock), the metadata, and the fence file. Note that even when a load instruction hits in the L1 cache, a packet is sent to the race detector for ascertaining the presence (or absence) of a race. The execution, however, can continue while race detection lags behind until the buffers between the L1 cache and the race detector overflow. This helps hide most of the latency due to extra accesses on L1 cache hits (Section V).

A. *Operation of ScoRD*

We now detail how ScoRD uses the above-mentioned hardware state and metadata to detect races. We start by describing preliminary checks, followed by checks performed for happens-before-based detection and lockset-based detection, in that order.

There are four types of instructions that are involved in either updating the hardware state/metadata and/or involved in detecting races. The fence and barrier operations update only the hardware state while memory instructions (loads/stores) and atomic operations update both the hardware state and the metadata in the memory, as well as activate race detection logic. When an SM executes a barrier, the BarrierID (counter) of the

TABLE III: Preliminary checks to ascertain if an access cannot participate in a race. md stands for the metadata entry.

Type	Condition
(a) Initialization	md.Modified && md.BlkShared && md.DevShared
(b) Program order	md.WarpID == WarpID && md.BlockID == BlockID && !md.BlkShared && !md.DevShared
(c) Barrier	BlockID == md.BlockID && BarrierID != md.BarrierID && !md.DevShared

TABLE IV: Conditions for races. fFile represents corresponding fence file entry.

Type	Condition
(a) Missing block fence	md.Modified && md.BlockID == BlockID && md.BlkFenceID == fFile.BlkFenceID && md.DevFenceID == fFile.DevFenceID
(b) Missing device fence	md.Modified && md.BlockID != BlockID && md.DevFenceID == fFile.DevFenceID
(c) Not strong access	!md.Strong OR !Strong
(d) Scoped atomic	md.IsAtom && md.Scope == BLOCK && md.BlockID != BlockID
(e) Missing common lock on load	md.Modified && intersect_locks().empty()
(f) Missing common lock on store	intersect_locks().empty()

issuing warp in the SM is incremented. When a fence operation executes, the issuing warp ID, the threadblock ID, and its scope are sent to the race detector. The race detector looks up the fence file with a combination of warp and threadblock ID and increments the block or device-scope FenceID based on the scope of the fence (Figure 6). This way, the fence file is updated with the latest fence ID executed by a warp.

The execution of a load, a store, or an atomic instruction triggers access to the metadata and a check for a race. First, the instruction type (load, store, or atomic), address, instruction's warp ID, threadblock ID, barrier ID, fence IDs (block and device scope), and the bloom filter containing the active locks (described later) are sent to the race detector. The Metadata accessor in the race detector (Figure 6) then looks up the metadata in the memory corresponding to the address of the access. Once the metadata is available, the Detection logic (Figure 6) first initiates the preliminary check as detailed next.

Preliminary race check: The objective of the preliminary check is to quickly ascertain if an access is trivially race-free. ScoRD checks three conditions for this purpose (Table III). ① It checks if the access is the first access after (re-) initialization of the memory location. This is done by checking if three flags in the metadata (Modified, BlkShared, DevShared) are all set (condition (a) in Table III). ② Next, it checks for program order across all previous accesses. This is checked by first matching the WarpID and the BlockID in the metadata with that of the

current access. It also makes sure that both BlkShared, and DevShared are unset in the metadata (condition (b)). This guarantees that the same warp performed accesses to the given memory location. ③ Finally, it checks if the previous accesses to the memory location were from the same threadblock and whether a barrier separates the previous and current access (condition (c)). If *any* of these three conditions are satisfied, then *no further race check* is performed. Otherwise, further checks are triggered (detailed shortly).

Memory access instructions also update the WarpID, BlockID, BarrierID, and FenceIDs of the metadata entry even if no race is detected. Atomic operations set the IsAtom flag and set Scope flag to block or device scope. A store or atomic instruction further sets the Modified flag. On a load, the DevShared flag is set if the BlockID in the metadata differs from the threadblock ID of the current access. Otherwise, the BlkShared flag is set if the WarpID in the metadata is different from the current access.

Detecting races due to fences or atomics: If all the conditions in the preliminary check fail, ScoRD looks for possible races due to improper use of fences or atomics as follows (first four conditions in Table IV). On a load, if the Modified flag in the metadata is unset, then no further check is required since loads alone cannot cause conflicting accesses. If the WarpID in the metadata is different from the warp ID of the current access, but the BlockID matches the threadblock ID of the current access, then it indicates the given memory location was earlier modified by a different warp from the same threadblock. Therefore, the load could constitute a conflicting access in the *block scope*. If the FenceIDs stored in the metadata match the threadfence IDs (block and device scope) of the last accessing warp stored in the fence file, then it indicates no fence was executed since the previous conflicting access. Otherwise, the FenceID in the fence file would have differed from that in the metadata. Thus ScoRD declares a *block-scope* race (condition (a) in Table IV).

If the threadblock ID of the current access is different from the BlockID stored in the metadata, then it could be a *device-scope race*. ScoRD declares a race if the DevFenceID value in the fence file matches that in the metadata (condition (b)).

As discussed in Section II-B, fences guarantee ordering only for strong operations (e.g., load/store with the volatile qualifier, atomics). Consequently, a race should be declared for conflicting accesses to a memory location that are not strong accesses, even if they are separated by a fence. To capture this, the first access with a volatile qualifier or an atomic after (re-)initialization sets the Strong bit in the metadata. Any access that is not strong unsets this bit and triggers a race (condition (c)). On a store, the race detection logic for both block and device scope is the same as above, except that the Modified flag is ignored.

For detecting atomic races, two aspects are considered. ScoRD first considers if a given memory location was previously accessed with loads/stores by checking if the isAtom bit is unset. In this case, the atomic operation is treated as if it were a store, and the checks proceed as discussed previously. If a load or store instruction finds that the previous access to the memory location was via an atomic (i.e., IsAtom flag set in the metadata), the scope of the operation was block (stored Scope flag in metadata) and BlockID is different from the threadblock ID of the current access, then a device-scope atomic race is declared.

Alternatively, if both the previous access and the current access to a memory location are atomic, then the detector looks for a device-scope race. Specifically, if ① the BlockID in the metadata is different from the threadblock ID of the current access, and ② if the scope information stored in the metadata is block, then a race is declared. Otherwise, the atomic accesses are race-free (condition (d) in Table IV).

Detecting races due to lock/unlock: As discussed in Section III, a pair of atomic and fence operations are often used in CUDA programs to implement locking. Specifically, an atomicCAS operation followed by a fence of appropriate scope is used for locking, and a fence followed by atomicExch is used for unlocking [18], [20]. Therefore, ScoRD needs to infer locks and uses a 4-entry lock table implemented as a circular buffer for this purpose. Each entry contains a hash of the address of the variable (6 bits), a bit for scope information, a valid bit, and an active bit. Whenever a warp executes an atomicCAS instruction, an entry for it is inserted in the lock table with the valid bit set, but the active bit unset. Whenever a fence instruction executes, the SM sets the active bits of lock table entries with matching or lesser scope. An active entry indicates that the warp currently holds that lock. On the execution of atomicExch, the entry with matching hash and scope has its valid bit unset indicating a release.

On execution of a load/store instruction, a summary of the active entries in the lock table is sent to the race detector. Specifically, the hash of the lock table entries and the scope is inserted into a 16-bit bloom filter, and sent along with the other information of the instruction as detailed before (e.g., warp ID, threadblock ID). The race detector then looks up the relevant metadata. If either the bloom filter content in the metadata or that of the current access is non-empty, then lockset-based race detection is triggered. There, on a load, the Modified bit in the metadata is checked. If set and the intersection (bitwise AND) of the bloom filter value stored in the metadata and that of the current access is empty, then a race due to improper locking is declared (condition (e) in Table IV). Similarly, on a store, a race is declared if the intersection of the bloom filter contents is empty (condition (f)).

Note that it is possible that multiple lock addresses hash to the same value in the bloom filter, thus incorrectly indicating a common lock. This, in turn, leads to a rare possibility that ScoRD could miss a true race (false negative). False positives are also theoretically possible in ScoRD due to the overflow of counters/IDs. For example, as the BlkFenceID is 6 bits, if exactly 64 block-scope fences are executed between conflicting accesses to a memory location, a false race will be declared due to overflow. However, such cases are practically non-existent. Increasing counter sizes would reduce the chance of false races, but would increase hardware overhead.

B. Optimization: Software cache for metadata

One of the drawbacks of the above design is the 200% memory overhead for the metadata. One option to reduce this overhead is to increase the granularity of tracking. For example, instead of tracking races at 4-bytes granularity, one can track them at 16-bytes granularity. That will reduce the overhead to 50%. However, this introduces a significant number of false positives due to the sharing of metadata.

Towards this, we observe that racey accesses to a memory location typically happen close together in time. Further, a majority of the memory locations accessed by a program do not participate in races. Even if the metadata for those locations is absent, it will not impact race detection. ScoRD, therefore, keeps metadata for recently accessed memory locations in a software cache, instead of tracking it for every memory location. Specifically, ScoRD keeps a direct-mapped software cache of metadata entries. For example, in the default configuration, ScoRD keeps one metadata entry for every 16th 4-byte segment of memory. Each entry is augmented with a 4-bit tag to uniquely identify metadata, given an address.

The metadata is looked up by indexing into the metadata region with the physical address of the memory access. Note that a contiguous physical memory region is set aside for the metadata at boot, and thus, the offset into the region is easily calculated by dividing the address of the memory access with the size of the metadata region. When the Metadata accessor inside the race detector looks up the metadata entry, it also checks the tag. On a mismatch, that metadata is not used for race detection and, thus, a race can potentially be missed (false negative). ScoRD overwrites the metadata entry with the information of the latest access.

In the Evaluation, we will show that this optimization very rarely introduces false negatives (less than 3% of cases). The reason is that the mere aliasing of metadata entries does not trigger a false negative. To trigger a false negative, the same metadata entry should correspond to at least two memory (data) locations that are accessed concurrently, and at least one of those memory locations should be part of a race. This is not a common occurrence given that only a small fraction of an application's memory potentially participates in a race. Further, since metadata is allocated in memory, it is possible to configure ScoRD during boot to not leverage metadata caching, if memory overhead is not a concern.

C. Overheads due to hardware state and metadata

The BarrierID per threadblock needs 64 bits for each SM. A lock table is 36 bits (9×4) long, and there are 32 of them per SM, one per warp. These add to 152 bytes per SM.

Each entry in the memory fence file is 12 bits long (two 6-bit counters, one each for block scope and device scope). There is one entry for each warp in each SM, and thus, the state overhead is 720 bytes. In total, the hardware state overhead adds up to about 2.9KB. Finally, in the default configuration, the metadata overhead is 12.5% of device memory size.

TABLE V: Default hardware configurations.

Number of SMs	15	Threads / warp	32
Max. threads / block	1024	Registers / SM	32768
Threadblocks / SM	8	Max. warps / SM	32
Private L1 cache	16 KB, 4-way, 128B blocks, global write-evict, local write-back	Shared L2 cache	1.5 MB, 8-way, 128B blocks, write-back
GDDR5 timing	$t_{RRD}=6, t_{RCD}=12, t_{RAS}=28, t_{RP}=12, t_{RC}=40, t_{CL}=12$	Memory channels	12

V. Evaluation

ScoRD is simulated using GPGPU-Sim [31]. The hardware parameters in the default configuration are listed in Table V. The benchmarks and microbenchmarks were compiled using CUDA 8.0 and use PTX 5.0. Further details of benchmarks are given in Table II. Matrix Multiplication, Rule 110, Reduction, and 1D Convolution use randomly generated input. Graph Connectivity and Graph Coloring uses input generated through GTgraph [32] The tool generates realistic graphs using the R-MAT algorithm [33].

A. Results

We evaluate ScoRD against following key questions. ① Is ScoRD able to detect races in the applications and microbenchmarks? ② What are the performance implications of ScoRD? ③ What is ScoRD's impact on the number of DRAM accesses and what is the efficacy of software caching of metadata? ④ What are the key sources of overhead of ScoRD? ⑤ Finally, how sensitive is ScoRD to L2 cache size?

Table VI shows the number of unique races in each application and how many of those were reported by ScoRD. In total, we find that out of 44 races, the base design (Section IV), which does not employ software caching of metadata, correctly captures all races. ScoRD with caching of metadata catches all but one (i.e., one false negative) that it misses due to aliasing in the direct-mapped cache of the metadata. In short, ScoRD is very accurate in reporting a large number of different races.

Figure 8 shows the execution cycles normalized to *no* race detection. There are two bars for each application. The second one represents ScoRD, while the first one is the base design without software caching of metadata.

TABLE VI: Number of races caught by different configurations.

Workload	Races present	Base design w/o metadata caching	ScoRD
MM	4	4	4
RED	2	2	2
R110	2	2	1
GCOL	6	6	6
GCON	5	5	5
1DC	1	1	1
UTS	6	6	6
Microbenchmarks	18	18	18
Total	**44**	**44**	**43**

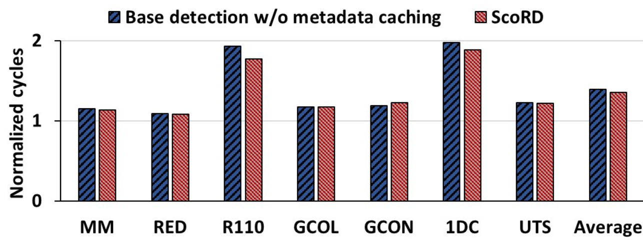

Fig. 8: Performance of ScoRD. Execution cycles normalized to no-race detection.

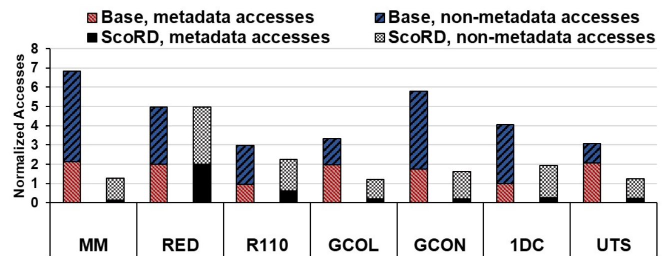

Fig. 9: Normalized number of accesses to DRAM.

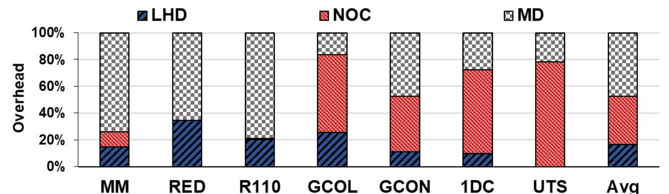

Fig. 10: Performance overhead breakdowns.

We find that the performance overhead of ScoRD across seven applications is only about 35%. This is significantly smaller than most existing GPU race detectors. The application 1DC, however, suffers more significant performance degradation (about 88%). We find that 1DC generates many network packets due to frequent atomic operations. Even small perturbation in network congestion due to additional metadata transfer for ScoRD impacts the performance of the application in a non-negligible way. In contrast, graph algorithms demonstrate irregular memory access patterns and, consequently, are memory-bound. The overheads of metadata accesses and race detection thus show less relative impact on application performance.

We further observe that software caching of metadata in ScoRD also helps its performance. Since software caching keeps only $1/16^{th}$ of the metadata entries, the number of unique DRAM accesses to metadata reduces substantially, and this aids the performance of ScoRD.

Figure 9 shows the number of DRAM accesses (L2 cache misses) normalized to *no* race detection (lower is better). Each application has two stacked bars. As before, the first one is without metadata caching, and the second one is ScoRD. The total height of the bars is normalized to the number of DRAM accesses under no-race detection. Each bar also shows what fraction of DRAM accesses are due to metadata and non-metadata access. Non-metadata accesses include normal data accesses and writebacks that occur in the course of the program's execution. We observe that due to metadata accesses, the total number of accesses may increase substantially. Metadata entries also contend with normal data for L2 capacity, thus increasing the normal data accesses as well. However, with metadata caching, on average, we access only $1/16^{\text{th}}$ of unique metadata entries. This reduces both DRAM accesses due to metadata and also the contention in the L2 cache. This is the key reason behind the improvement of performance with the optimization that maintains a software cache of metadata.

The key motivation behind the software caching of metadata is to reduce metadata overhead (from 200% to 12.5%). Previous work has proposed increasing the granularity of tracking to reduce metadata overhead [14]. However, this approach introduces false positives since races could be declared due to the sharing of metadata entries. Table VII shows the number of false positives as the granularity of metadata is increased from 4 bytes (Base detector without metadata caching) to 8 and 16 bytes. We observe that false positives increase substantially with increasing granularity, especially for graph algorithms. ScoRD, instead, takes an entirely different approach to reducing metadata overhead by leveraging temporal locality in races. This way, ScoRD does not introduce false positives, while still decreasing memory overheads to just 12.5%.

We further break down the sources of overhead introduced by ScoRD. There are three primary sources of the overhead as follows. ① Stalling the execution on L1 cache hits while waiting for the race detector (LHD), ② congestion in the on-chip network due additional information (e.g., WarpID, BlockID) sent on the packets (NOC), and ③ accesses, and writebacks to the metadata (MD). In three sets of separate experiments, we turned off timing simulations for each of these and measured performance uplifts to estimate their *relative* contribution to total overhead.

Figure 10 illustrates how the different sources of overhead impact the performance of each benchmark. On average, we find that relative contributions are 16.5%, 36.2%, and 47.3% for LHD, NOC, and MD, respectively. In general, applications with well-coalesced accesses (e.g., RED, R110) generate fewer packets on the network and thus, experienced negligible

TABLE VII: False positives with varying metadata granularity.

Tracking granularity	4-byte	8-byte	16-byte	ScoRD
Metadata overhead	200%	100%	50%	12.5%
MM	0	1	3	0
RED	0	1	1	0
R110	0	2	2	0
GCOL	0	27	29	0
GCON	0	28	28	0
1DC	0	1	2	0
UTS	0	12	23	0

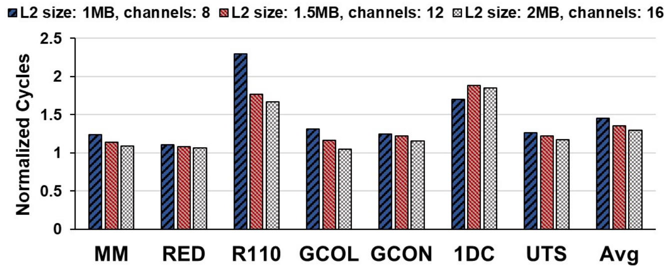

Fig. 11: Sensitivity to memory resources.

impact from on-chip congestion. Most of its overhead is due to metadata accesses. In contrast, graph applications generate many packets and congest the network as their irregular memory access pattern leaves little opportunity for coalescing, causing a much higher impact from on-chip congestion. We note that UTS experienced no impact of LHD. This application accesses its data structures in the global memory, e.g., shared stacks, exclusively using volatile operations. Since volatile operations bypass the L1 cache, no L1 hits are encountered even when the detection is turned off.

B. Sensitivity study

Figure 11 shows ScoRD's performance sensitivity to L2 cache size and memory bandwidth. There are three bars for each application. The middle bar in the cluster represents the default configuration (Table V). The left-most bar represents a lower L2 capacity and DRAM bandwidth, and the rightmost bar represents more L2 capacity and bandwidth than the default. The heights of each bar are normalized to the number of execution cycles under *no* race detection for a given configuration.

The overhead of ScoRD increases with a more constrained memory subsystem (except for 1DC). This is expected; a smaller L2 cache size and/or less memory bandwidth increases contention among the accesses to metadata and that to the normal data. However, for 1DC, less resources degraded the execution under no-race detection *relatively* more than when ScoRD was in execution. Thus, we observe slightly less relative slowdown with lower memory resources.

VI. Discussions

Detecting races in presence of Independent Thread Scheduling (ITS): The SIMT execution at warp granularity is fundamental to GPU's performance advantage in executing massively data-parallel code. However, with the Volta architecture onward, NVIDIA introduced ITS [34]. In ITS, when a warp experiences branch divergence, instructions from both paths are executed alternately by their respective threads. This may lead to a new type of race if different threads within a warp access common data during divergent execution.

While ScoRD does not support ITS yet, it can detect this new type of race with minor modifications. First, the BarrierIDs for warps are split into a 7-bit BarrierID, and a hasDiverged bit that tracks whether a warp has diverged. ScoRD would then extend the WarpID in the metadata with 5 extra bits (currently unused) to store the ThreadID of accessors. The detection logic then changes minimally to consider the ThreadID if hasDiverged is set. Otherwise, it uses the WarpID as usual.

TABLE VIII: Comparison of support by GPU race detectors.

Detector	Fences	Locks	Scoped fences	Scoped atomics	Low perf. overhead (<3X)
LDetector					✓
HaccRG	✓	✓			✓
Barracuda	✓	✓	✓		
CURD	✓	✓	✓		
ScoRD	✓	✓	✓	✓	✓

Support for acquire/release: With the release of PTX 6.0, two new scoped synchronization operations – acquire and release – have been added in NVIDIA GPUs [21].

While ScoRD does not support explicit acquire and release instructions, it supports acquire and release patterns in the case of lockset detection of critical sections. Extending it to support explicit acquire and release instructions is not difficult.

A global counter is maintained in the race detector, incremented for every release operation. Similar to the fence file, a release file is introduced, which keeps the global release counter value of the last release by a warp. Stores update the value of the release counter in the metadata of accessed data. A bit is used to track if the data was used in a release operation. During acquire, this bit is checked for a previous release, in which case the details of block ID and warp ID are sent to the acquiring warp. SMs store the details of synchronized warps. Race detection would then follow a similar procedure to fence race detection, using the release file instead.

VII. Related work

GPUs traditionally used cache flushes and invalidations to implement global synchronization. However, this becomes prohibitively expensive for large workloads. Besides, often global synchronizations are unnecessary. Newer GPUs, thus, support scoped synchronization that provides synchronization at levels imitating the GPU's execution hierarchy.

The traditional sequential consistency for data-race-free memory model [35] falls short for scopes. Researchers have thus proposed heterogeneous-race-free models that take scopes into consideration [3], [5]. Remote-scope promotion [4], [36] proposes to enable the dynamic promotion of synchronization scopes if the scope encompassing the producer and the consumer of a data item is not known statically. Researchers have also proposed an alternative that does not require scopes [37], which instead uses the DeNovo [38] coherence protocol. However, current commercial GPUs support scopes [23], [39].

Several prior works have explored race detection in GPUs on a limited scale. Boyer et al. [40] proposed race detection by running GPU kernels on emulators, but this incurs heavy slowdowns. GRace [16] and GMRace [13] propose race detectors that use static analysis and dynamic checking. Besides these, NVIDIA released Racecheck [15], a runtime tool to detect races. However, these detectors restrict themselves to shared memory and ignore the more challenging task of global memory race detection.

LDetector [41] detects races by taking snapshots and comparing changes in values. However, races caused by stores that do not change the values stored are not detected. It also ignores fences and atomics. SMT solving [42]–[45] has also been proposed to find races, but significant false positives may occur. Furthermore, none of these consider scoped races.

HAccRG [14] uses hardware support to detect races in global memory and bears similarities to our design. However, they too ignore scoped races. Besides, HAccRG incurs a memory overhead of 150%, which makes their solution less practical.

Barracuda [12] uses binary instrumentation to detect races in GPU programs. CURD [11] builds on this, optimizing the common case where synchronization occurs through barriers while relying on Barracuda for atomics and fences. While scoped fences are supported in Barracuda, it ignores scoped atomics. Further, being implemented purely in software, they observe slowdowns as high as 1000x for Barracuda, and 25x for CURD. Table VIII summarizes the differences between ScoRD and a few closely related detectors. As depicted, none of the previous race detectors support the detection of all types of scoped races while achieving low-performance overheads.

CPU race detection has been extensively explored. While many race detectors have been proposed [8]–[10], [46]–[54], these cannot be directly applied to GPU owing to its very different architecture and programming. Importantly, CPUs lack scoped synchronization.

VIII. Conclusion

As more GPU applications use scoped synchronization, it becomes important to detect potential scoped races. To the best of our knowledge, ScoRD is the first hardware-based race detector that can detect scoped races in GPUs. In addition, we have created seven applications and 32 microbenchmarks that utilize scoped synchronization to aid further research in this domain. ScoRD can detect a large range of races, with a 35% performance overhead, and a 12.5% memory overhead.

IX. Acknowledgement

We thank anonymous reviewers of ISCA'20 for their constructive criticism of this work. We thank Mark D. Hill, Shweta Pandey, Ajay Nayak, and Neha Jawalkar for their feedback on the drafts of this article. This work is partially supported by the startup grant provided by the Indian Institute of Science and a research grant from Microsoft Research India.

References

[1] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, Aug. 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[2] NVIDIA, "Gpu-accelerated applications," 2016, http://images.nvidia.com/content/tesla/pdf/Apps-Catalog-March-2016.pdf.

[3] D. R. Hower, B. A. Hechtman, B. M. Beckmann, B. R. Gaster, M. D. Hill, S. K. Reinhardt, and D. A. Wood, "Heterogeneous-race-free memory models," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 427–440. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541981

[4] M. S. Orr, S. Che, A. Yilmazer, B. M. Beckmann, M. D. Hill, and D. A. Wood, "Synchronization using remote-scope promotion," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 73–86. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694350

[5] B. R. Gaster, D. Hower, and L. Howes, "Hrf-relaxed: Adapting hrf to the complexities of industrial heterogeneous memory models," *ACM Trans. Archit. Code Optim.*, vol. 12, no. 1, pp. 7:1–7:26, Apr. 2015. [Online]. Available: http://doi.acm.org/10.1145/2701618

[6] "Cuda c++ programming guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/, NVIDIA, accessed: 2019-11-15.

[7] K. Group, "OpenCL," 2014, https://www.khronos.org/opencl/.

[8] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, Nov. 1997. [Online]. Available: http://doi.acm.org/10.1145/265924.265927

[9] K. Serebryany and T. Iskhodzhanov, "Threadsanitizer: Data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, ser. WBIA '09. New York, NY, USA: ACM, 2009, pp. 62–71. [Online]. Available: http://doi.acm.org/10.1145/1791194.1791203

[10] P. Zhou, R. Teodorescu, and Y. Zhou, "Hard: Hardware-assisted lockset-based race detection," in *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, ser. HPCA '07. IEEE, Feb 2007, pp. 121–132. [Online]. Available: https://ieeexplore.ieee.org/document/4147654

[11] Y. Peng, V. Grover, and J. Devietti, "Curd: A dynamic cuda race detector," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: ACM, 2018, pp. 390–403. [Online]. Available: http://doi.acm.org/10.1145/3192366.3192368

[12] A. Eizenberg, Y. Peng, T. Pigli, W. Mansky, and J. Devietti, "Barracuda: Binary-level analysis of runtime races in cuda programs," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: ACM, 2017, pp. 126–140. [Online]. Available: http://doi.acm.org/10.1145/3062341.3062342

[13] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "Gmrace: Detecting data races in gpu programs via a low-overhead scheme," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 1, pp. 104–115, Jan. 2014. [Online]. Available: https://doi.org/10.1109/TPDS.2013.44

[14] A. Holey, V. Mekkat, and A. Zhai, "Haccrg: Hardware-accelerated data race detection in gpus," in *Proceedings of the 2013 42Nd International Conference on Parallel Processing*, ser. ICPP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 60–69. [Online]. Available: https://doi.org/10.1109/ICPP.2013.15

[15] "Racecheck tool," https://docs.nvidia.com/cuda/cuda-memcheck/index.html#racecheck-tool, NVIDIA, accessed: 2019-11-19.

[16] M. Zheng, V. T. Ravi, F. Qin, and G. Agrawal, "Grace: A low-overhead mechanism for detecting data races in gpu programs," in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1941553.1941574

[17] A. Dinning and E. Schonberg, "An empirical comparison of monitoring algorithms for access anomaly detection," in *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '90. New York, NY, USA: Association for Computing Machinery, 1990, p. 1–10. [Online]. Available: https://doi.org/10.1145/99163.99165

[18] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*, 1st ed. Addison-Wesley Professional, 2010.

[19] "Cuda by example - errata page," https://developer.nvidia.com/cuda-example-errata-page, NVIDIA, accessed: 2020-05-01.

[20] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson, "Gpu concurrency: Weak behaviours and programming assumptions," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 577–591. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694391

[21] "Parallel thread execution isa version 6.5," https://docs.nvidia.com/cuda/parallel-thread-execution/, NVIDIA, accessed: 2019-11-20.

[22] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, p. 558–565, Jul. 1978. [Online]. Available: https://doi.org/10.1145/359545.359563

[23] D. Lustig, S. Sahasrabuddhe, and O. Giroux, “A formal analysis of the nvidia ptx memory consistency model,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA: ACM, 2019, pp. 257–270. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304043

[24] V. Nagarajan, D. J. Sorin, M. D. Hill, D. A. Wood, N. Enright Jerger, and M. Martonosi, *A Primer on Memory Consistency and Cache Coherence: Second Edition*, 2nd ed. Morgan and Claypool Publishers, 2020.

[25] R. H. B. Netzer and B. P. Miller, “What are race conditions?: Some issues and formalizations,” *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, Mar. 1992. [Online]. Available: http://doi.acm.org/10.1145/130616.130623

[26] “Cuda samples,” https://docs.nvidia.com/cuda/cuda-samples/index.html#threadfencereduction, NVIDIA, accessed: 2019-11-20.

[27] M. Deveci, E. G. Boman, K. D. Devine, and S. Rajamanickam, “Parallel graph coloring for manycore architectures,” in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, ser. IPDPS ’16. IEEE, May 2016, pp. 892–901.

[28] M. Sutton, T. Ben-Nun, and A. Barak, “Optimizing parallel graph connectivity computation via subgraph sampling,” in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, ser. IPDPS ’18, May 2018, pp. 12–21.

[29] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, “Uts: An unbalanced tree search benchmark,” in *Languages and Compilers for Parallel Computing*, G. Almási, C. Caşcaval, and P. Wu, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 235–250.

[30] M. D. Sinclair, J. Alsop, and S. V. Adve, “HeteroSync: A Benchmark Suite for Fine-Grained Synchronization on Tightly Coupled GPUs,” in *IEEE International Symposium on Workload Characterization*, ser. IISWC, October 2017.

[31] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, “Analyzing cuda workloads using a detailed gpu simulator,” in *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS ’09. IEEE, 2009, pp. 163–174.

[32] D. A. Bader and K. Madduri, “Gtgraph: A synthetic graph generator suite,” *Atlanta, GA, February*, 2006.

[33] D. Chakrabarti, Y. Zhan, and C. Faloutsos, “R-mat: A recursive model for graph mining,” in *Proceedings of the 2004 SIAM International Conference on Data Mining*. SIAM, 2004, pp. 442–446.

[34] “Inside volta: The world’s most advanced data center gpu,” https://devblogs.nvidia.com/inside-volta/, NVIDIA, accessed: 2019-11-20.

[35] S. V. Adve and M. D. Hill, “Weak ordering-a new definition,” in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, ser. ISCA ’90. New York, NY, USA: ACM, 1990, pp. 2–14. [Online]. Available: http://doi.acm.org/10.1145/325164.325100

[36] J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, “Remote-scope promotion: Clarified, rectified, and verified,” in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA 2015. New York, NY, USA: ACM, 2015, pp. 731–747. [Online]. Available: http://doi.acm.org/10.1145/2814270.2814283

[37] M. D. Sinclair, J. Alsop, and S. V. Adve, “Efficient gpu synchronization without scopes: Saying no to complex consistency models,” in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 647–659. [Online]. Available: http://doi.acm.org/10.1145/2830772.2830821

[38] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. V. Adve, V. S. Adve, N. P. Carter, and C.-T. Chou, “Denovo: Rethinking the memory hierarchy for disciplined parallelism,” in *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT ’11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 155–166. [Online]. Available: https://doi.org/10.1109/PACT.2011.21

[39] B. R. Gaster, “Hsa memory model,” in *2013 IEEE Hot Chips 25 Symposium (HCS)*, ser. HCS ’13. IEEE, Aug 2013, pp. 1–42.

[40] M. Boyer, K. Skadron, and W. Weimer, “Automated dynamic analysis of cuda programs,” in *2008 Workshop on Software Tools for MultiCore Systems*, 2008.

[41] P. Li, C. Ding, X. Hu, and T. Soyata, “Ldetector: A low overhead race detector for gpu programs,” in *5th Workshop on Determinism and Correctness in Parallel Programming (WODET2014)*, 2014.

[42] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson, “Gpuverify: A verifier for gpu kernels,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA ’12. New York, NY, USA: ACM, 2012, pp. 113–132. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384625

[43] E. Bardsley, A. Betts, N. Chong, P. Collingbourne, P. Deligiannis, A. F. Donaldson, J. Ketema, D. Liew, and S. Qadeer, “Engineering a static verification tool for gpu kernels,” in *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 226–242. [Online]. Available: https://doi.org/10.1007/978-3-319-08867-9_15

[44] E. Bardsley and A. F. Donaldson, “Warps and atomics: Beyond barrier synchronization in the verification of gpu kernels,” in *Proceedings of the 6th International Symposium on NASA Formal Methods - Volume 8430*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 230–245. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-06200-6_18

[45] A. Betts, N. Chong, A. F. Donaldson, J. Ketema, S. Qadeer, P. Thomson, and J. Wickerson, “The design and implementation of a verification technique for gpu kernels,” *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 3, pp. 10:1–10:49, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2743017

[46] M. A. Bender, J. T. Fineman, S. Gilbert, and C. E. Leiserson, “On-the-fly maintenance of series-parallel relationships in fork-join multithreaded programs,” in *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’04. New York, NY, USA: ACM, 2004, pp. 133–144. [Online]. Available: http://doi.acm.org/10.1145/1007912.1007933

[47] M. D. Bond, K. E. Coons, and K. S. McKinley, “Pacer: Proportional detection of data races,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10. New York, NY, USA: ACM, 2010, pp. 255–268. [Online]. Available: http://doi.acm.org/10.1145/1806596.1806626

[48] D. Dimitrov, M. Vechev, and V. Sarkar, “Race detection in two dimensions,” in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA ’15. New York, NY, USA: ACM, 2015, pp. 101–110. [Online]. Available: http://doi.acm.org/10.1145/2755573.2755601

[49] A. Dinning and E. Schonberg, “Detecting access anomalies in programs with critical sections,” in *Proceedings of the 1991 ACM/ONR Workshop on Parallel and Distributed Debugging*, ser. PADD ’91. New York, NY, USA: ACM, 1991, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/122759.122767

[50] L. Effinger-Dean, B. Lucia, L. Ceze, D. Grossman, and H.-J. Boehm, “Ifrit: Interference-free regions for dynamic data-race detection,” in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA ’12. New York, NY, USA: ACM, 2012, pp. 467–484. [Online]. Available: http://doi.acm.org/10.1145/2384616.2384650

[51] T. Elmas, S. Qadeer, and S. Tasiran, “Goldilocks: A race and transaction-aware java runtime,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07. New York, NY, USA: ACM, 2007, pp. 245–255. [Online]. Available: http://doi.acm.org/10.1145/1250734.1250762

[52] C. Flanagan and S. N. Freund, “Fasttrack: Efficient and precise dynamic race detection,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’09. New York, NY, USA: ACM, 2009, pp. 121–133. [Online]. Available: http://doi.acm.org/10.1145/1542476.1542490

[53] C. Lidbury and A. F. Donaldson, “Dynamic race detection for c++11,” in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, ser. POPL 2017. New York, NY, USA: ACM, 2017, pp. 443–457. [Online]. Available: http://doi.acm.org/10.1145/3009837.3009857

[54] R. O’Callahan and J.-D. Choi, “Hybrid dynamic data race detection,” in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’03. New York, NY, USA: ACM, 2003, pp. 167–178. [Online]. Available: http://doi.acm.org/10.1145/781498.781528