Rethinking Runtime Verification on Hundreds of Cores: Challenges and Opportunities

Tayfun Elmas
Semih Okur
Serdar Tasiran

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2011-74
http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-74.html

June 13, 2011

Rethinking Runtime Verification on Hundreds of Cores: Challenges and Opportunities

Tayfun Elmas[1], Semih Okur[2], and Serdar Tasiran[2]

[1] University of California, Berkeley, USA,
elmas@eecs.berkeley.edu
[2] Koç University, İstanbul, Turkey,
{sokur,stasiran}@ku.edu.tr

Abstract. We propose a novel approach for runtime monitoring and verification on computers with a large number of computation cores. The goal of the approach is to minimize the impact of runtime verification on the performance of the application being monitored. We distinguish between two kinds of computational overhead: (i) overhead caused by instrumentation and/or logging, and (ii) parallelizable overhead due to the verification algorithm(s) analyzing the executions. So far, runtime verification algorithms have been designed to run on the same threads as the code being monitored and both (i) and (ii) contribute to the slowdown of the program being monitored. The framework we propose allows us to carry out (ii) on separate, dedicated cores and threads. As a result, the program being monitored only experiences slowdown due to (i). We conjecture that, with some inexpensive hardware support, (i) can be reduced to negligible levels. By parallelizing analyses so that they run at least as fast as the program being monitored, but on separate computational resources, one can potentially use this approach for monitoring, error detection, containment, and recovery from errors. As a demonstration of concept, we investigate runtime monitoring for concurrency bugs, in particular, data race detection. We use a few CPU threads/cores and a large number of cores on a GPU to minimize the slowdown of the application on which race detection is being run. Our early experimental results indicate that this approach has potential.

1 Introduction

Computers with tens to hundreds of CPU cores are expected to be commonplace in the near future. This has resulted in an aggressive push for parallelization. Legacy application and systems software is revised, and new software is developed explicitly to make use of a large number of cores. For software correctness and reliability, this has significant consequences. Little if any of the aggressively concurrent software is mature. Thus, it makes sense to dedicate some of the concurrent processing power and hardware resources available on chips with many cores to monitoring for concurrency errors, and reducing the impact of such monitoring on the threads being monitored. In this paper, we propose such a framework. A key feature of our framework is that decouples runtime verification algorithms from the programs being monitored in a manner that makes it possible to run the programs and analyses in parallel, thus minimizing the impact on program performance.

We propose an approach to make use of some of the computation cores and other hardware resources in a computer to monitor the programs running on other cores for concurrency errors, to contain and/or recover from these errors, if not immediately, shortly after they take place. While exploring such an approach, we had two goals:

1. to have minimal, tolerable impact on the threads being monitored, and
2. to have the monitoring algorithms work at the same speed as the program, while possibly lagging behind by a bounded amount.

The rationale behind the first goal is to enable efficient, even post-deployment use of the monitoring and bug-detection algorithms for safety-critical systems. The rationale behind the second goal is to make it possible to contain concurrency errors, notify the threads that have experienced the errors, and gracefully shut down the program or to recover from the error. One result of the second goal is to force the monitoring framework to parallelize the event logging and analysis algorithms as much as possible.

These goals motivated us to distinguish between two kinds of overhead induced by runtime monitoring and verification: (i) overhead caused by instrumentation and/or logging, and (ii) parallelizable overhead due to the verification algorithm(s) analyzing the executions. Traditionally, online runtime verification algorithms are run on the same threads as the code being monitored and both (i) and (ii) contribute to the slowdown of the program being monitored. The novelty of the framework we propose is that it allows us to carry out (ii) on separate, dedicated cores and threads. As a result, the program being monitored only experiences slowdown due to (i). By parallelizing analyses so that they run at least as fast as the program being monitored, but on separate computational resources, this approach has the potential to enable continuous monitoring, error detection, containment, and recovery from errors for systems with large number of cores.

In our approach, the application being monitored and the actual monitoring code run on separate processing units/resources. They communicate with each other using some shared memory or message passing. The instrumented application code only has the additional responsibility of communicating relevant events to the monitoring code. The monitoring and runtime analysis code can be quite complex, but runs on separate processors and is parallelized, thus, the application performance is not affected by the runtime analyses being performed. We conjecture that the performance penalty on the application being monitored due to instrumentation and communication of relevant events can be reduced to negligible levels, for example, using inexpensive hardware support such as hardware-assisted message passing [5]. The goal is for the monitoring code to run at least the same speed as the application being monitored, but lag behind by a very small delay due to event communication. (In our experiments this delay was in milliseconds.) This makes possible scenarios in which, in response to errors detected, the application is shut down gracefully, or a previous valid checkpoint is restored or the application is restarted.

As a demonstration of concept, we investigate runtime monitoring for concurrency bugs, in particular, data race detection. Since systems with hundreds of cores are not yet available as mainstream, to investigate the feasibility of our proposal, we use a few CPU threads/cores to carry out the efficient concurrent transfer of logged events from the CPU cores to a Graphics Processing Unit (GPU), and we use the GPU to run our race detection algorithm. Today's GPUs provide a highly parallel, multithreaded, computation environment with hundreds of processor cores and a higher memory bandwidth than CPUs. Thus, our framework allows us to investigate opportunities for efficiently performing various kinds of runtime analyses on highly parallel computing environments.

We provide a framework that instruments binaries so that the application threads log the interesting events in a central event list. The analysis threads then work off of this event list to perform possibly expensive but parallelized analyses. The separation of recording of events from their analysis led us to investigate techniques dedicated to efficient recording of events. For this purpose, in the future, inexpensive hardware support may be provided. For this, we use carefully designed non-blocking algorithms and block-based handling of the event list for efficient recording of the events in this list. In particular, we communicate the events to the GPU for processing in fixed-size segments called *frames*. We accomplish fast, highly parallelized runtime analysis on a GPU with hundreds of cores by exploring algorithms that can check each event frame independently from other frames. Our experience was that this can be done without significantly affecting the soundness of the checking. Long-enough frames allow the analyses to catch all errors that can be caught by analyzing the entire execution. Since the computational cost of the analysis threads does not affect application performance, in this highly parallel setting, one does not have to sacrifice precision in order to achieve low performance impact.

For demonstration, we adapted the well known Eraser and Goldilocks algorithms for data race checking, so that they can be parallelized to run on a large number of threads and cores on the GPU. Surprisingly, this high parallelism has a simplifying effect on the algorithm implementation. Since we have many threads/cores, the algorithm can be written to make each thread or core to perform a local and independent check (for a single memory access) without having to worry about sharing or interaction with other threads. Each thread creates only the necessary data structures for the check and discards/reuses them after the check completes; this avoids the need for memory management and sharing of complicated algorithm-specific data structures.

We implemented our proposed system in a tool called Kuda. Kuda is open source and available at http://kuda.codeplex.com. We use the Pin [6] library to instrument binaries for monitoring and the CUDA [8] library to run our analysis algorithms on the GPU. We applied Kuda to a number of multithreaded programs from the PARSEC and SPLASH-2 benchmark suites. We performed experiments using CPU and GPU implementations of the Eraser and Goldilocks algorithms. We chose Eraser to represent a cheap (although imprecise) algorithm, while Goldilocks served as a representative precise, higher-complexity algorithm. We contrasted two approaches: (i) a straightforward implementation of ERASER running on the same threads and cores as the application, and (ii) an implementation of GOLDILOCKS using our framework, where the checking threads are decoupled and run on the GPU. Overall, our early experimental results indicate that our approach is promising. Using a cheaper race detection algorithm using the traditional approach as exemplified by (i) causes about twenty times more slowdown compared to a more complex race-detection algorithm implemented in our approach!

To the best of our knowledge, ours is the first study on decoupling runtime monitoring algorithms for concurrent programs and parallelizing them so that they run at the same speed with the monitored program, which, as a result, suffers minimal performance impact. There is work on offline runtime verification algorithms, where, while the application is being run, only logging overhead is incurred. However, such studies do not focus on parallelizing runtime analyses and running them concurrently, at the same speed as the application. Thus, they do not offer the same error containment and recovery potential as our approach. Furthermore, these approaches do not pay attention to minimizing the resources used for communication between application threads and analysis threads. Note that the approach in our paper addresses a different problem from that of race-detection for code running on GPUs.

2 Challenges in runtime monitoring

The purpose of this section is to point out the key challenges one is likely to face when building a runtime verification tool for concurrency-related errors. In order to make our ideas concrete we will work out this section using a well-known data-race detection algorithm.

2.1 Example: Eraser algorithm for data-race detection

ERASER is a well-known lockset-based algorithm for detecting race conditions dynamically [10]. For simplicity of presentation, we focus on the core algorithm using only locksets without distinguishing between read and write accesses.

A *race condition* occurs if two different threads perform conflicting accesses (i.e., at least one of them is a write) on a shared (global) variable and there is no proper synchronization between these accesses. In order to detect race conditions, ERASER enforces the locking discipline that every shared variable x is protected by a common lock throughout the execution.

The Eraser algorithm maintains for each variable x, a lockset $LS(x)$ representing the set of the algorithm's guess of the locks protecting x. It also maintains for each thread t, a lockset $LH(t)$ representing the set of locks held by thread t at a given point in an execution. $LH(t)$ is updated appropriately when thread t acquires and releases a lock. The algorithm attempts to infer the actual protecting locks for each data variable x by initializing $LS(x)$ to the set of all locks in the program and then updating $LS(x)$ to be the intersection $LH(t) \cap LS(x)$ at each access to x by a thread t.

2.2 Cost of runtime monitoring on the CPU

The implementation of the Eraser algorithm requires 1) to monitor the events in an execution that the algorithm needs to keep track of, which is usually done by instrumenting the program's source or binary code, and 2) to perform some computation to update the algorithm-specific data structures, i.e., the maps LH and LS, and check some conditions, i.e., "Does $LS(x)$ become empty?". In 1), events are either immediately communicated to the algorithm by running a call-back function to perform 2), or saved to a temporary buffer to be processed later (e.g., in a linked list of events as in [3]). There are two main sources of runtime cost, which combined together contribute highly the overhead of the monitoring on the application:

The instrumentation cost, i.e., the cost of monitoring and communicating events to the algorithm for processing. In Eraser, every shared memory operation and synchronization (locking) operations has to be monitored. As the number and variety of events monitored by the algorithm increases, the frequency of interrupting the execution with callbacks to the algorithm increases and this becomes a bottleneck even though the actions of the algorithm are simple and cheap. Our experimental results in Sec. 5 show that only instrumenting the program (without performing any computation at instrumentation points) can generate 1.6X to 7.1X overhead on the uninstrumented program.

The analysis cost, i.e., the cost of processing the events and updating the algorithm's state accordingly. Runtime verification algorithms for concurrent programs usually maintain data structures shared among the threads participating in the algorithm, For example, the Eraser implementation maintains a lockset for each thread and and for each shared variable. Other algorithms maintain, e.g., pointers to the last accessing thread or an additional virtual-clock vector to internal locks for synchronizing accesses by different threads. Fetching and manipulating these data structures at high frequencies creates a considerable overhead and may cause big divergences in the timing behavior of threads.

For Eraser, accessing the map LS (or $LS(x)$, if the map is distributed) may require to use a common lock to avoid two threads both accessing x to manipulate $LS(x)$ simultaneously. This creates large critical sections (code segments to be executed atomically) throughout the execution and is a significant source of runtime overhead. In summary, while Eraser is one of the simplest, cheapest algorithms for race detection, its implementations can cause considerable runtime and memory overhead and this makes the race checking hard to apply at the post-deployment.

In order to reduce the runtime overhead of the checking, we distribute the responsibilities for the algorithm to worker (checker) threads separate from the application threads. Checker threads run on separate cores, and do not slow down the application being monitored. In this paper, we investigate this idea by running the runtime analyses on GPUs. Our novel approach has the side benefit of simplifying the implementation of runtime verification algorithms, which are often forced to make use of tricky data structures and optimizations when run on the same threads as the applications. When run on separate cores, simpler but parallelized implementations of these algorithms provide the required performance.

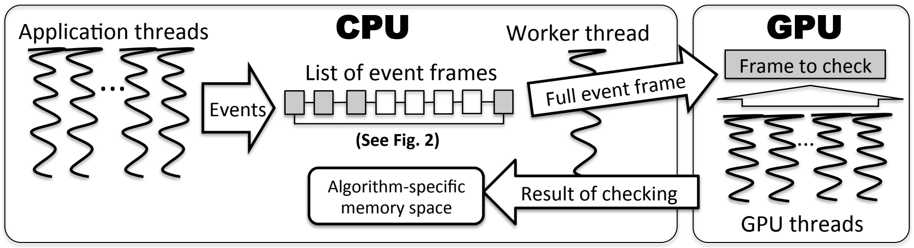


Fig. 1. Components of our runtime monitoring system.

3 Our approach I: Overall system

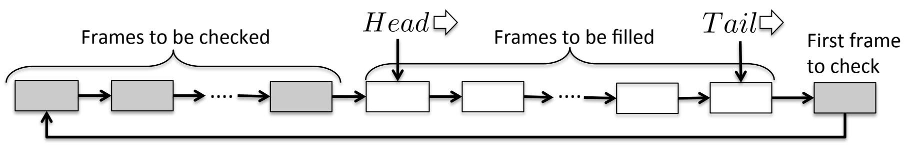
Motivated by the challenges given in the previous section, our main goal is to design a runtime verification framework that will have the minimum negative impact on the program's running time and concurrency. Our key design decision is to carry out the checking algorithm (i.e., data-race detection) on physically separate multi-processors, in our case the GPU cores. The application threads running on the CPU are only responsible for recording their events in a shared data structure and communicating events to the GPU for further processing. Fig. 1 illustrates this separation of responsibilities between the CPU and GPU threads. In this section, we present our techniques for observing an execution trace, i.e., recording events and communicating them to the GPU. The following section complements this description by giving GPU-based algorithms for data-race detection.

3.1 Observing the execution trace

Our technique is based on logging the execution as a linear sequence of events and running the analysis in a very efficient way. In order to enable efficient handling of the event log, we process the a fixed-size segment of this log, called *frame*, at a time. In our experiments we fixed this size as 1024-events and refer to it by the FrameSize constant. We treat each event frame a *unit of input* for the analysis implemented in the GPU. Each frame is checked independently from other frames and minimal information is kept between frames, e.g., racy variables to omit accesses to those variables. When a frame is completely checked, the events in it is discarded and it is reused to store later events.

While splitting the execution into independent frames may cause unsound results due to pairs of events from separate frames, our framework allows to adjust the precision to increase the chance of finding bugs. We have chosen to defer the soundness issue, since the goal of this study was to show the feasibility of highly parallel, at-speed runtime verification. Empirical evidence by other researchers indicates that this is a minor source of unsoundness: Many concurrency errors involve a small number of threads, and can be detected by focusing on a short portion of the execution [7].

Fig. 2 shows our main data structure for keeping event frames: a circular linked list. At any time this list contains a fixed number of frames, where each frame is a memory buffer to store FrameSize events. As explained below, the circular linked list allows us to reuse the frames in an efficient way throughout the execution. Fig. 2



```
Algorithm RecordEvent(e)
(Executed by application threads)
   // Find the first frame to insert the event
1  frame := Head
2  index := AtomicGetAndIncrement(frame.size)
3  while (index ≥ FrameSize) {
4      if (frame = Tail) { goto line 1 } // restart
5      frame := frame.next
6      index := AtomicGetAndIncrement(frame.size)
7  }
   // Insert event to the frame at index
8  frame[index] := e
   // Shift Head, if the frame becomes full
9  if (index = FrameSize − 1) { Head := Head.next }
```

```
Algorithm CheckFrames()
(Executed by worker thread)
1  while (program is running) {
2      wait until Head ≠ Tail
3      frame := Tail
       // Check frame at GPU
4      Copy frame to GPU device memory
5      Asynch-Call GPU kernel for race checking
       // Shift Tail to reuse the frame
6      frame.size := 0
7      Tail := frame.next
8      wait until GPU kernel finishes
9      Copy result of the checking from GPU
10 }
```

Fig. 2. The cyclic linked list of event frames and related algorithms.

shows pseudo code to record events (**RecordEvent**) and to process full event frames, i.e., communicating them to the GPU for the analysis (**CheckFrames**). While the former is performed by application threads, we dedicate a separate worker thread (running on the CPU) for the latter.

The event list is managed by non-blocking algorithms with few atomic instructions; no lock is required to record an event and process an event frame. At any point in the execution, we keep two pointers to frames in our event list: *Head* and *Tail*. The part of the list between *Head* and *Tail* (both inclusively) contains the frames (shown in white color in Fig. 2) that are being filled by application threads. The rest of the list between *Tail* and *Head* contains the frames that have become full and waiting to be checked (shown in gray color). At the initial state of the event list *Head* and *Tail* points to the same frame. While frames become full, *Head* is shifted, and as the full frames are checked, *Tail* is shifted (as shown in Fig. 2). In order to prevent data races on *Head* and *Tail*, we read from and write to these variables using atomic-reference operations.

Recording events (**RecordEvent** in Fig. 2)**.** Each event frame has a field called *size*, which stores the number of events in the frame. When an application thread wants to record an event, it traverses the list starting from *Head* (lines 1-7). At each step it performs an atomic operation that reads the current *size* of the frame being visited and increments its *size* by one (lines 2 and 6). If *size* of the last visited frame before incrementing was less than FrameSize, then the thread uses that value as *index* of the frame to record the event (line 8). Otherwise, the thread tries following frames in the list in a loop (lines 3-7). If a thread reaches *Tail* while traversing the list, it restarts as this indicates that the current frame is full and subject to checking by the worker thread. In our experiments we observed that, because the checking of the frames runs at speed very close to the program, such restarts were quite rare, i.e., there is always at least one empty slot to insert an event between *Head* and *Tail*.

After adding the event to the right frame, if the current application thread finds out that the current frame is *Head* and has just become full, it shifts the *Head* pointer to the next non-empty frame in the list (lines 9).

Processing full event frames (**CheckFrames** in Fig. 2)**.** Our worker thread takes a full frame a time and sends it to the GPU for the checking (in Fig. 2 this is the rightmost frame in gray). For this, the worker thread continuously executes the loop until the program finishes (We omit the code that processes the non-empty frames after the program terminates). At each iteration of the loop, the thread first waits until *Head* and *Tail* do not point to the same frame, i.e., the list contains full frames (line 2). When the condition holds, the thread locates the frame pointed by *Tail* (line 3) and checks it at the GPU. See Sec. 4.1 for explanation of procedure (lines 4-5 and 8-9) for running the analysis on the GPU. As the analysis on the GPU runs asynchronously with the CPU, the worker thread spends the time to wait until the GPU computation terminates to mark the currently checked frame empty (line 6) and to shift *Tail* forward and make the frame available to be reused to record new events (line 7). Upon completion of the kernel call (line 8), the worker thread copies the result of the checking, i.e., racy accesses, from the GPU's memory back to the CPU's memory (line 9), in an algorithm-specific memory space. While our system reports all the errors at the end of the execution, it can be modified to report the errors as soon as it gets the response from the GPU.

4 Our approach II: Checking frames on the GPU

Having explained the CPU part of our runtime monitoring system, we present parallel algorithms for the data-race checking on the GPU cores. We first brief on GPU computing using the CUDA model and bring in some challenges in that model, which affected our system design. Then, we present our adaptation of Eraser and Goldilocks algorithms to run on parallel GPU threads.

4.1 Background on GPU computing using CUDA

The CUDA model allows programmers to write code in an extension of the C language that will be run on GPU in a highly parallel manner. The mapping of the code to physical processing units on the GPU is transparent to the programmer, and this enables one to write parallel code that can scale for devices with different parallel processing capabilities.

Each code portion to be run on GPU is written as a C function called *kernel* and can be called from C/C++ code executing on the CPU. Thus, in our framework, each analysis algorithm is written as a C function. The CPU and GPU threads operate on memory modules physically isolated from each other. As a result, we have to maintain a separate memory space on the GPU's own device memory. For this, at the beginning of the execution, we pre-allocate a memory region, as large to fit a full event frame, on the GPU's own device memory at the beginning of the execution. Additional space is also allocated to hold the intermediate results and outputs of the kernel's computation. The pointers to these memory regions are given as arguments when to the kernel call. Our worker thread (running on the CPU) must follow the following steps to run an analysis on a full event frame (line numbers below refer to the pseudo code **CheckFrames** in Fig. 2).

1. The worker thread first copies the contents of the event frame to the pre-allocated region on the GPU device memory (line 4).
2. It calls the kernel function of an available checker algorithm (line 5). That kernel function is executed by the GPU cores in parallel and asynchronously with the CPU. When calling the kernel function, it passes as arguments the pointer to device memory region storing the current frame as well as additional values, such as the number of events in the frame. Each kernel is executed in a SIMD (single instruction, multiple data) style on multiple cores and threads. The number and hierarchical organization of the threads participating in the kernel's execution are given as special arguments to the kernel call. Each GPU thread gets a unique thread identifier among the other threads using special `threadIdx` variable (in the scope of the kernel code). This unique id allows the thread to determine parts of the event frame it should work on without interfering with other threads.
3. The worker thread uses CUDA routines to synchronize with the kernel execution for further processing (line 8). Upon completion of the kernel call, the worker thread copies the result of the checking, i.e., pairs of racy accesses in the case of data-race detection, from the GPU's device memory back to the CPU's memory (line 9) to be reported later.

While the CUDA model provides hundreds of cores available to the highly efficient analysis of event frames, it also comes with the following challenges:

- Explicitly transferring the inputs and outputs of the kernel between the CPU and GPU creates an extra communication overhead for each kernel call. Thus, we chose the frame size carefully to manage this overhead. In addition, we maintained any data that was not used in the checking separately and did not sent it to the GPU; the rest of the data relevant for the checking was encoded to fit small data structures to reduce the CPU-GPU communication cost.
- Although state-of-the-art GPU devices can have more than 1GB of memory and latest CUDA libraries offer dynamic memory allocation in the kernel, we believe that relying on limited amount of memory is essential for the efficiency of the kernel execution. Therefore, we were determined to use fixed-size representations for data structures (some of which are allocated prior to the execution) that store locksets and racy pairs of accesses detected.
- CUDA enforces the SIMD execution model rather than the typical multiple-instruction model in the CPU. This SIMD style execution of the kernel forces one to implement the algorithm in a special form of data-parallelism in order to get the maximum benefit from this model.

4.2 Data-race checking on the GPU

Given the challenges in writing kernels, we wrote parallel kernels for the Eraser and Goldilocks algorithms. Fig. 3 gives the pseudocode for these kernels. The challenge in writing the kernels is to trade the challenges given above with the large number of cores available on the GPU. In our algorithms, each thread checks a unique variable access in the given event frame, creating the data structures, i.e.,

```
Algorithm **EraserKernel**
**Input:** $Frame$ – Array of events.
**Output:** $DataRaces$ – Pairs of racy events.
    // Get thread id
1  $id$ := $threadIdx.x$
     // Fetch event
2  $e$ := $Frame[id]$
    // check if this is an access event
3  if ($IsRead(e.kind)$ or $IsWrite(e.kind)$) {
    // find out the next access to the same variable
4    $id'$ := -1
5    for ($i = id$ to $Frame.size$) {
6      $e'$ := $Frame[i]$
7      if ($IsRead(e'.kind)$ or $IsWrite(e'.kind)$) {
8        if ($e'.value = e.value$) {
9          $id'$ := $i$
10         break
11   } } }
     // if there is another access, check that access
12   if ($id' \neq$ -1) {
       // compute $LS_R$ for the first accessor
13     $LS_A$ := $LS_R$ := $\emptyset$
14     for ($i = id + 1$ to $id' - 1$) {
15       $e''$ := $Frame[i]$
16       if ($e''.tid = e.tid$) {
17         if ($IsAcquire(e''.kind)$)
18           $LS_A$ := $LS_A \cup \{e''.value\}$
19         if ($IsRelease(e''.kind)$ and $e''.value \notin LS_A$)
20           $LS_R$ := $LS_R \cup \{e''.value\}$
21     } }
       // compute $LS'_A$ for the second accessor
22     $LS'_A$ := $LS'_R$ := $\emptyset$
23     for ($i = id' - 1$ back to $id + 1$) {
24       $e''$ := $Frame[i]$
25       if ($e''.tid = e'.tid$) {
26         if ($IsRelease(e''.kind)$)
27           $LS'_R$ := $LS'_R \cup \{e''.value\}$
28         if ($IsAcquire(e''.kind)$ and $e''.value \notin LS'_R$)
29           $LS'_A$ := $LS'_A \cup \{e''.value\}$
30     } }
       // perform the check for data race
31     if ($LS_R \cap LS'_A = \emptyset$)
32       $DataRaces$ := $DataRaces \cup (e, e')$
33 } }
```

```
Algorithm **GoldilocksKernel**
**Input:** $Frame$ – Array of events.
**Output:** $DataRaces$ – Pairs of racy events.
    // Get thread id
1  $id$ := $threadIdx.x$
     // Fetch event
2  $e$ := $Frame[id]$
    // check if this is an access event
3  if ($IsRead(e.kind)$ or $IsWrite(e.kind)$) {
    // find out the next access to the same variable
4    $id'$ := -1
5    for ($i = id$ to $Frame.size$) {
6      $e'$ := $Frame[i]$
7      if ($IsRead(e'.kind)$ or $IsWrite(e'.kind)$) {
8        if ($e'.value = e.value$) {
9          $id'$ := $i$
10         break
11   } } }
     // if there is another access, check that access
12   if ($id' \neq$ -1) {
       // initialize a local lockset for e.value
13     $LS$ := $\{e.tid\}$
       // run rules of the algorithm
14     for ($i = id + 1$ to $id' - 1$) {
15       $e''$ := $Frame[i]$
         // Process operations of kind Acquire
16       if ($IsAcquire(e''.kind)$ and $e''.value \in LS$)
17         $LS$ := $LS \cup \{e''.tid\}$
         // Process operations of kind Release
18       if ($IsRelease(e''.kind)$ and $e''.tid \in LS$)
19         $LS$ := $LS \cup \{e''.value\}$
20     }
       // perform the check for data race
21     if ($e'.tid \notin LS$)
22       $DataRaces$ := $DataRaces \cup (e, e')$
23 } }
```

Fig. 3. Pseudocode for the kernels running the Eraser and Goldilocks algorithms.

locksets, necessary for the check locally (in its stack) and discarding them after the check completes. Due to the limited GPU memory space and the requirement to pre-allocate the memory used by the kernel, using finite-size representations for data structures in the kernel is essential. For this, we represent locksets in both the Eraser and Goldilocks kernels with bloom filters, which can represent a collection of addresses (of locks, variables, etc.) in constant-size bitsets. As the data required for a single check is of finite size and used locally and temporarily, dynamically created objects or threads do not create extra memory space or management.

Each kernel in Fig. 3 takes an event frame ($Frame$) and returns a list of racy events ($DataRaces$). We start by explaining the common portions of both kernels between lines 1-12. Note that, $Frame$ is an array of events. We organize threads in a one-dimensional thread block. Each thread obtains its id from $threadIdx.x$

(line 1), and fetches the id^{th} event from the frame (line 2). If the fetched event is not a shared variable access (line 3), the thread terminates. Otherwise, between lines 4-11, it finds the next access to the same variable (saved in $e.value$). If it finds a next access within the same frame (line 12), then id' stores the index of the access and e' stores information about that access. The thread then checks if the accesses recorded at e and e' involve in a race. The kernels differ in how this checking is done.

EraserKernel computes two sets of locksets. Between lines 13-21, the lockset LS_R is computed to find out the locks released by the first accessor thread ($e.tid$) after accessing the variable. We also use the lockset LS_A to avoid incorrectly adding a lock to LS_R that is acquired and released after the access. Between lines 22-30, the lockset LS'_A is computed to find out the locks acquired by the second accessor thread ($e'.tid$) before accessing the variable. We also use the lockset LS'_R to avoid incorrectly adding a lock to LS'_A that is acquired and released before the access. [3] At line 31, we compute the intersection $LS_R \cap LS'_A$. If this intersection is not empty, then this means that the second accessor thread acquired at least one of the locks the first thread was holding when it accessed the variable. Otherwise, then the pair (e, e') is added to the set of racy event pairs (line 32).

GoldilocksKernel uses only one lockset, LS, initialized to a singleton containing the id of the first accessor thread (line 13). The thread traverses the events (denoted e'') between e and e' and update LS. Lines 16-20 applies the standard rules for GOLDILOCKS [3]: If the current event e'' is of an acquire and the lock acquired (saved in $e''.value$) is in LS, then the thread acquiring the lock (saved in $e''.tid$) is added to LS (lines 16-17). If the current event e'' is of a release and the thread acquiring the lock (saved in $e''.tid$) is in LS, then the lock acquired (saved in $e''.value$) is added to LS (lines 18-19). In GOLDILOCKS [3], acquiring a lock, start of a thread, joining a thread, and reading from a volatile variable are all considered as *acquire* events, and releasing a lock, creating a new thread, end of a thread, and writing to a volatile variable are all considered as *release* events. Thus, the lines 16-20 can process any events in these categories. Upon completion of the traversal at line 21, the thread checks if LS contains the id of the second accessor thread. If the id is not in LS, then the pair (e, e') is added to the set of racy event pairs (line 22).

5 Implementation and experimental evaluation

We aim to evaluate two claims we referred to in Sec. 1: First, our separation of monitoring and analysis to CPU and GPU significantly reduces the overhead of the traditional approach in which both are performed on the same threads/cores. Second, our analysis code runs at a similar speed as the program and finishes soon after the program terminates. For this, we implemented our proposed system in

[3] The technical reason for this slightly complex implementation, where LS_R and LS'_A are computed in two different loops and using additional locksets (LS_A and LS'_R), is that bloom filters only support addition and lookup but not removal. Using other representations for locksets (e.g., finite-size hash tables) would make this code simpler.

Benchmark	Description	Lines	#Threads	#Events	#Frames
`PARSEC`					
`blackscholes (L)`	Black-Scholes partial differential equations	1661	9	238M	224K
`bodytrack (L)`	tracking human body with multiple cameras	7385	10	2707M	2.6M
`canneal (L)`	cache-aware simulated annealing	1793	9	468M	449K
`dedup (M)`	data stream compression	3681	25	1993M	1.9M
`fluidanimate (L)`	simulating incompressible fluid	945	9	2461M	2.3M
`raytrace (S)`	optimized ray tracing	>6K	9	332M	316K
`streamcluster (S)`	online clustering problem	2531	9	178M	166K
`swaptions (L)`	monte carlo simulation	1615	9	2731M	2.6M
`x264 (M)`	H.264/AVC video encoder	3014	64	1460M	1.4M
`SPLASH-2`					
`barnes`	Barnes-Hut for N-body problem	3507	4	3035M	2.9M
`cholesky`	blocked sparse cholesky factorization	5684	8	269M	254K
`fmm`	adaptive fast multipole for N-body problem	5434	4	1629M	1.5M
`fft`	complex 1D FFT	1462	8	577M	556K
`lu`	blocked LU decomposition	1380	8	1087M	1M
`ocean`	large-scale ocean simulation	8176	8	531M	510M
`radix`	integer radix sort	1530	8	302M	287K
`raytrace`	optimized ray tracing	11043	9	332M	316K
`water-nsquared`	water simulation w/out spatial data structure	3098	4	3120M	7.2M
`water-spatial`	water simulation with spatial data structure	3655	4	727M	701K

Table 1. Description of our benchmarks, and number of events and frames generated at a typical run. For the PARSEC benchmarks the input size is given in parantheses (`(S)`:**simsmall**, `(M)`:**simmedium**, `(L)`:**simlarge**), and for the SPLASH-2 benchmarks we used the default inputs except that some values are taken from Table 1 of [1].

a prototype tool called KUDA and applied KUDA on a collection of multithreaded benchmarks. KUDA consists of two parts:

1. A dynamic library containing the core functionality including the routines for recording events, managing event frames, and running the race detection kernels on the GPU. We use the CUDA 4.0 library [8] to write and call kernels for analyzing frames and to manage the GPU resources (e.g., transferring data to/from the GPU device memory). While our experiments are performed using the global memory, our system can use constant and texture memory. The fact that event frames are only read by the kernel enables us to make use of the constant and texture memory, which are cached for fast read-only access.
2. A Pin [6] tool to dynamically instrument x86 binaries in order to callback the routines in our dynamic library on certain events (shared memory read/write, thread creation/join, and inter-thread synchronization). Our Pin tool supports multithreaded programs written using the `pthreads` library (for thread creation and join, and synchronization primitives including mutex and readers/writer locks).

5.1 Experiments

Benchmarks. We applied our tool KUDA on a collection of multithreaded programs from PARSEC [2] and SPLASH-2 [11] benchmark suites. The names, brief descriptions, sizes (lines of code) of these programs are listed in Table 1. The table also gives, for each benchmark, the number of threads, events, and frames generated in a representative execution of the benchmark. Notice that, in a typical execution, our benchmarks generate a few hundreds of millions of events and hundreds of thousands of frames, each of which is checked on the GPU.

Benchmark	Uninstr.	Only Instrumented		Eraser on CPU		Only with Events		Goldilocks on GPU	
	Runtime	Runtime	Slowdown	Runtime	Slowdown	Runtime	Slowdown	Runtime	Slowdown
`PARSEC`									
`blackscholes`	1.31	2.83	2.1X	136.04	101X	20.89	14.7X	29.27	21.1X
`bodytrack`	4.11	10.93	2.6X	1044.48	251X	305.11	72.5X	317.58	75.6X
`canneal`	8.85	14.81	1.6X	431.04	47X	67.5	6.9X	71.66	7.4X
`dedup`	2.25	7.06	3.1X	972.12	429.9X	202.94	88X	233.56	101.6X
`fluidanimate`	3.29	8.46	2.5X	1024.52	308X	281.27	83.9X	295.24	88.1X
`raytrace`	14.43	29.35	2X	>30min	>123.7X	98.24	5.7X	105.37	6.2X
`streamcluster`	7.27	22.72	3.1X	244	31.4X	419.21	55.5X	434.94	57.7X
`swaptions`	2.61	8.01	3X	1150.97	437X	312.26	117.5X	318.57	119.9X
`x264`	1.09	7.84	7.1X	710.12	645.2X	172.2	151.7X	176.66	155.8X
`SPLASH-2`									
`barnes`	3.08	7.61	4X	1542	499.1X	348.12	111.5X	362.12	116.1X
`cholesky`	0.94	3.04	3.2X	205.34	216.2X	32.61	32.4X	33.39	33.2X
`fmm`	1.85	5.53	2.9X	2697	1455.8X	169.45	89.6X	186.21	98.6X
`fft`	1.47	3.2	2.1X	329.21	222.7X	68.42	45.3X	72.52	48.1X
`lu`	0.44	2.63	5.9X	477	742.2X	123.6	190X	131.21	201.9X
`ocean`	0.86	3.47	4X	262	301.6X	57.21	63.4X	61.21	68.1X
`radix`	1.07	2.18	2X	136.62	126.6X	34.45	31.1X	36.53	33.1X
`raytrace`	14.6	30	2X	>30min	>122.2X	117.4	6.9X	120.46	7.2X
`water-nsquared`	4.61	16.18	3.5X	3274	707.6X	851.35	182.1X	894.12	191.4X
`water-spatial`	0.63	2.91	4.6X	308	485.2X	74.67	114.9X	85.14	131.5X

Table 2. Results from our experiments. Running times are given in seconds.

Hardware. We performed our experiments on a HP xw9300 Workstation running Ubuntu Linux 10.10 32-bit kernel. Our machine has two (single-core) AMD Opteron processors with 2600 MHz clock frequency, 128 KB L1 cache, 1 MB L2 cache, and 8 GB memory (400 MHz). We used a GeForce GTX 465 GPU card with Fermi chipset. Our card provides 352 cores (11 processors with 32 cores each) with 1.21GHz clock rate, 1.23 GB of memory space with 1.4 GB/sec host-to-device memory bandwidth and 71.3 GB/sec in-device memory bandwidth.

Configuration parameters. For the experiments, we chose the following parameters that gave the best results in terms of runtime and memory overhead. We selected the event frame size (FrameSize) to be **1024** events. We initialize the cyclic list in Fig. 2 with **2048** frames. Thus, our system requires only
2048 frames * 1024 events (each frame) * 8 bytes (each event) = **16 MBytes** of memory space to store the events for the CPU. We run **128** GPU threads over each event frame. In order to get the maximum benefit from the GPU device's concurrent computing functionality, we collect and send to the GPU **128** consecutive event frames at a time. In this way we aim to utilize the high parallelism on the GPU to analyze multiple frames simultaneously.

5.2 Results

Table 2 gives the runtime measurements for several configurations we run for each benchmark. The column "Uninstr." lists the running time of the benchmark without any instrumentation. For other columns, we report on both the running time of the program and the slowdown in the execution over the uninstrumented runtime. The running times for all columns are given in *seconds*. When computing the slowdown for the columns "Eraser on CPU", "Only with Events" and "Goldilocks on GPU", we subtract the instrumentation cost (i.e., "Only Instrumented" - "Uninstr.") from the runtime before dividing it to the running time of "Uninstr".

The column “Only Instrumented” gives the results for the experiments when the benchmarks were loaded with Pin and relevant instructions are instrumented, but no action was taken at the instrumentation points except for calling an empty function (simply no-op). The results indicate that the instrumentation even without executing any extra code incurs overhead that ranges between 1.6X (`canneal`) and 7.1X (`x264`).

In order to compare the runtime cost of our approach and the traditional approach in which the race detection runs on the same cores as the application, we implemented the Eraser, and two vector clock-based algorithms DJIT$^+$ [9] and FastTrack [4] (available in our code base). For these algorithms, we used the same Pin instrumentation, but applied the algorithm’s rules on the application threads immediately when a relevant event occurs. Our implementations are not perfectly optimized as in the original implementations, but still provide a rough estimate for the overhead of checking on the CPU.

We observed that the overhead of the DJIT$^+$ and FastTrack implementations on the CPU are much higher than Eraser. Thus, the Eraser algorithm provides lower bounds for the runtime and slowdowns for these algorithms and in Table 2 we only list the results for Eraser under the column “Eraser on CPU”. Notice that, the slowdown when running such a simple algorithm starts from 31.4X (`streamcluster`) and can reach 1455.8X (`fmm`). Overall, running Eraser on the same cores as the application incurs a very high overhead and a few hundreds of times slowdown.

We give the results for our system under two columns. The column “Only with Events” gives the results when the race checking on the GPU is disabled, but the application threads are still recording their events. The column “Goldilocks on GPU” gives the results when the race checking on the GPU is enabled. While our system contains GPU kernels for both the Eraser and Goldilocks algorithms, we observed that the overhead when using Eraser gives only slightly lower overhead. Goldilocks is a precise race-detection algorithm, and is the most expensive and complex one of the algorithms we investigated.

In both columns “Only with Events” and “Goldilocks on GPU”, we consider the runtime of the execution after both the program and the analysis of the event frames terminated. In fact, we observed that the analysis terminates shortly after the program terminates. The difference ranges between 1-3 milliseconds (on average 2.5 milliseconds). In addition, we observed that our system does not need to allocate new event frames; it simply reuses the initially allocated 2048 frames. This result, together with the small difference between the execution times of the program and analysis, indicates that the analysis runs at speed very close to the program, following the program behind only in milliseconds.

Our results clearly indicate that performing the checking on separate cores in a highly parallelized way dramatically reduces the overhead of the runtime verification. The ratio of the slowdown of the race checking on the CPU to that of the race checking on the GPU is between 3.3 (`bodytrack`) and 14.7 (`fmm`). Only for `streamcluster` the CPU-based implementation beats our system and gives less slowdown. Moreover, for `raytrace` benchmarks in both PARSEC and SPLASH-2, the execution took more than our specified upper time limit, 30 minutes; thus when we also consider these benchmarks, the ratio of the slowdown of the CPU-based race checking to that of the GPU-based checking reaches at least 17 and 20 times, respectively.

In addition, the very small difference between the slowdowns in “Only with Events” and “Goldilocks on GPU” shows that the overhead of monitoring and recording events and managing the list of event frames highly dominates the overall overhead of our system. The ratio of the overall slowdown to that of only managing the events goes only up to 1.4 (e.g., `blackscholes`). While the overhead of recording events is still high (e.g., for post-deployment purposes), this small difference between enabling and disabling on-GPU checking gives a promising evidence that the parallel processing events on the GPU gives negligible overhead.

References

1. C. Bienia, S. Kumar, and Kai Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *IISWC*, pages 47 –56, 2008.
2. Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. Technical Report TR-811-08, Princeton University, January 2008.
3. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI*, pages 245–255, 2007.
4. Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.
5. Poletti Francesco, Poggiali Antonio, and Paul Marchal. Flexible hardware/software support for message passing on a distributed shared memory architecture. In *DATE*, pages 736–741, Washington, DC, USA, 2005.
6. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.
7. Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.
8. NVIDIA Corporation. *NVIDIA CUDA Programming Guide v4.0*. NVIDIA Corporation, 2011.
9. Eli Pozniansky and Assaf Schuster. Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, 2007.
10. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.
11. Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.