KUDA: Accelerating Dynamic Race Detection using Parallelism on a GPU

by

Ümit Can BEKAR

A Thesis Submitted to the

Graduate School of Engineering

in Partial Fulfillment of the Requirements for

the Degree of

Master of Science

in

Computer Science and Engineering

Koç University

June 24, 2013

To my family...

ABSTRACT

We propose a novel technique by introducing a coprocessor to runtime verification, ergo reducing the cost of race detection without any hardware extension to mainstream PC environment. The goal of our approach is to offload the high computational overhead of traditional race detection to hundreds of cores available at modern GPUs. Existing runtime verification frameworks have been designed to run on the same processing units as the code being monitored and (i) instrumentation and (ii) analysis costs contribute to the slowdown of the program being monitored. The framework we propose allows us to carry out (ii) on separate, dedicated cores. As a result, the program being monitored experiences slowdown due to bookkeeping of events, bottleneck is not caused by race detection. An orthogonal line of work shows that with some inexpensive hardware support, monitoring costs can be reduced to negligible levels. By parallelizing the offloaded work, our experiments show that they run as fast as the program being monitored, on separate computational resources. As a demonstration of concept, we investigate runtime monitoring for concurrency bugs, in particular, data race detection. We use a few CPU threads and a large number of cores on a GPU to minimize the slowdown of the application on which race detection is being run.

ÖZETÇE

Bu tezde sunulacak olan iş özgün bir çalışma zamanı doğrulama çerçevesidir. Yaklaşımımızdaki ana amaç ise geleneksel yarış durumu denetleyicilerindeki işletim yüklerini ayırıp, bilgisayarlarımızda bulunan donanımsal olanakları kullanarak, halihazırdaki çalışma zamanına koşut çalışan, koşutlu doğrulama yapmaktır. Bu yüzden çerçevemizi çok çekirdekli işlemcilere (CPU) ve grafik işlemcisine (GPU) sahip kişisel bilgisayarlara herhangi bir donanım eklemesi gerekmeksizin gerçekleştirdik. Çalışmamızdaki ana yenilik, koşutzamanlı bir programın güvenilirlik özelliklerinin grafik işlemcisindeki iş parçacıklarında denetlenmesinin ilk olarak öne sürülmesi ve bunun için gerekli tekniklerin ve algoritmaların tasarlanmasıdır. Daha önceki çalışmalarda bu denetlemenin tamamı merkezi işlem ünitesi üzerinde gerçekleşmekteydi, ve denetleyicinin iş parçacıklarının denetlenen program iş parçacıklarıyla aynı işlemci üzerinde koşması programın başarımını önemli ölçüde düşürmekteydi. Denetleyicilerdeki işletim yükünü ikiye ayırıyoruz: gözlemleme ve denetleme yükleri. Detaylı inceleyeceğimiz bu yazılım çerçevesi, ayırdığımız iki işletim yükünü farklı işlemcilere paylaştırmaktadır. Sonuç olarak, denetlenen koşutzamanlı programın başarımı, yalnızca gözlemleme ve bu gözle- min öteki işlemciye aktarımından kaynaklanan işletim yüklerinden dolayı etkilenir. Sunacağımız çerçevenin ön ürünü olan KUDA birimlerimizle yaptığımız

deneylerimiz, farklı işlemcideki iş parçalarında koşut zamanda yarış durumlarını (data race) denetlemektir.

ACKNOWLEDGMENTS

I would like to express my gratitude to my co-author Tayfun Elmas for his quality work with Semih Okur, which made the grounds of this thesis project. I would like to thank my thesis advisor Serdar Taşıran, all the past and current members of the Multicore Software Engineering Research Center: Burcu, Erdal, Hassan, İsmail, Ömer and Süha, and of course, my friends who were also co-located in our lab: Ayse Nur, İdil, Özge and Salih. The work reported in this thesis was supported by Microsoft® Research. I am very lucky to have my parents, Fazilet and Cemal. I am also very lucky to have my aunt and her husband Nuran and İsa for having me in their home with my dear cousins Mert and Cansu, especially when the times were harsh for me. Of course, I send love to my brothers Alp and Berk for their love back. My grandma Fatma and my other aunt Neşe and her husband Ismail, my cousin Cem and his newly wed wife Burcu with the cutest nephew, Emir; I feel sorry that I could not spend more time with them.

Finally, I thank anyone who took time to read my thesis. Especially, I must give immense gratitude to my dearest friends Betül Gül and İsmail Kuru, because if they wouldn't be a part of my journey here, I would possibly have been graduated at least a semester before, leaving this office without having so much fun in the process!

TABLE OF CONTENTS

LIST OF FIGURES

NOMENCLATURE

CLR	Common Language Runtime
CMP	Chip-level Multiprocessing
CPU	Central Processing Unit
GB	Giga bytes
GPU	Graphics Processing Unit
KB	Kilo bytes
OS	Operating System
PC	Personal Computer
RAM	Random Access Memory
$SIMD$	Single Instruction Multiple Data
SMP	Symmetric Multiprocessing
STI	Sony, Toshiba, IBM

Chapter 1

INTRODUCTION

1.1 Programming on Multicore Hardware

Writing commercial grade parallel software on top of multicore platforms today considered harder than writing sequential software on these platforms, it is mainly because achieving high performance and correctness together at the same time is notoriously harder job to accomplish on time. One of the main reason for this challenge is that, debugging parallel programs is much harder due to its nondeterministic execution behavior, resulted from different executions of a buggy program may work correctly for some time and on a specific interleaving, it may crash. This phenomenon is caused by nondeterministic behaviors of software, running on deterministic hardware. There exist two main reasons why any kind of software to behave nondeterministically: (i) input nondeterminism or (ii) thread scheduling nondeterminism that may lead to at least one data race during the execution of the program.

A second reason for increased difficulty of programming in parallel is that, when a program has a data race on today's computer systems, programmers can not rely on memory models of their systems due to the lost sequential consistency property.

The memory model of a shared-memory multicore is basically a contract between the hardware designer and the programmer of the multicore. For example, Java memory model[23] asserts that any Java program must have data race freedom in every execution in order to developers to have conventional reasoning during programming on Java platform. Sequential consistency is one of the consistency models used in the domain of concurrent programming. This property can be elaborated as a result of any execution of a program is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. It is a property was so fundamental that all the software developers (apart from the software developers who worked in then-niche area of HPC/cluster distributed programming) were considering it as a given, however since the superscalar CPU microarchitectures have been introduced, the sequential consistency[20] has became a property that has been relaxed.

Informally, a data race is a type of conflicting memory access that leads to nondeterministic behavior. A data race happens when at least two threads accesses a shared variable concurrently without synchronization, and at least one of them is a write (see Chapter 2, Section 2.2.1). A data race error[27, 26] results to memory corruption, other than lost sequential consistency. Lack of sequential consistency means that an execution that was further up in the source code could happen before the execution of the line; this situation lets us lose conventional reasoning for bugs in programs. Secondly, nature of thread level parallelism is that the threads are scheduled by the operating system non-deterministically: one execution interleaving might not happen until after millions of other possible following interleaving. These two reasons are exactly why debugging a parallel program is notoriously hard, one out of a billion execution might end up in an error and to catch that bug you have to recreate the whole interleaving up to the point of origin of data race.

1.2 Software Verification

Software verification is a field of software engineering whose goal is to assure conformance to all of the expected requirements of computer programs. Seminal papers by Robert Floyd in 1967 [14] and C.A.R. Hoare in 1969 [17] gave the field its start. In layman's terms, if we think of a program execution as a state machine, we can make assertions [39] about the state of the machine before and after the program executes.

Since the first piece of software was ever created, there existed specifications for it to be conformed, other than syntactic rules. Division by zero is the perfect example among others. For some critical programs these requirements can be much more detailed than a division by zero freedom. Requirements expected from software can be widely varied; there also exists specifications related to liveness of a program, which are orthogonal to safety specification, e.g. absence of a null dereference.

The crux of software verification is the Hoare logic, it is a formal system with a set of logical rules for reasoning rigorously about the correctness of computer programs using Hoare triples, basically a precondition, the statement and a postcondition. Hoare logic can provide rules for all the constructs of a simple imperative programming language. There exist two branches of software verification: dynamic and static verification. It is also common to have hybrid approaches[40]. Any formal software verification method can be reduced to Hoare triples.

Dynamic verification techniques are usually runtime testing methods and static verification techniques are usually source code analysis techniques. e.g. KUDA is a novel dynamic verification technique. However, dynamic verification techniques lack the ability to verify that software is error-free. Because dynamic techniques are basically functional testing, monitoring programs for errors appearing with respect to correctness algorithms, or kinds of temporal logic. This leads to Dijktra's famous quotation: "Testing can show the presence of errors, but not their absence". To tackle completeness of verification researchers have developed automated testing tools and rigorous testing tools, using systematic [24] or fuzzy approaches [16]. Both techniques can be commonly referred as model checkers. Model checkers are basically tools that monitor a target program over and over again for some time, and while doing eventually checks for all possible program states that it can represent. Such static verification techniques can be considered as a more complete approach to software verification, however it has completely different challenges and advantages over dynamic verification that will not be covered in this thesis.

1.3 Our Approach

Most significant difference of KUDA framework is that the application being monitored and the analysis program run on separate runtimes. They communicate with each other using a custom message passing interface provided by NVidia®CUDA™[29]. 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. CPU produces the events to be consumed on the GPU, so our approach is basically using GPU as a co-processor. Using co-processing for runtime verification, race detection to be exact, is our main contribution. Related work is left for another chapter.

Today's GPU architectures provide a highly parallel, multithreaded computation environment with hundreds of processor cores and a higher memory bandwidth than today's CPUs. Thus, our framework allows us to investigate opportunities for efficiently performing various kinds of runtime analyses on highly parallel computing environments.

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 [41, 42]. 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 bookkeeping of events. As a demonstration of concept, we investigate runtime monitoring for concurrency bugs, in particular, data races. Since CPUs 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 transfer of logged events from the CPU cores to a GPU, and we use the GPU to run our race detection algorithm.

1.4 Our Contributions

We first 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. For this, we use carefully designed 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 could 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 can achieve a lower performance impact while still not sacrificing from precision.

For demonstration, we adapted the well-known Eraser, Goldilocks and FastTrack algorithms for SIMD-based parallel 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 library to instrument binaries for monitoring and the CUDA library to run our analysis algorithms on the GPU. We applied KUDA to a number of multithreaded programs from the PARSEC [4] and SPLASH-2 [45] 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) implementations of GOLDILOCKS and ERASER 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 5x slowdown compared to a more complex race-detection algorithm implemented in our approach.

1.5 Organization of the Thesis

The organization of this thesis is as follows. In Chapter 2, we provide the necessary technical background information to reflect the importance and challenges of state of the art runtime verification of concurrent programs and introduce some of the significant algorithms also used as components of this thesis. In Chapters 3 and 4, we present the CPU and GPU components of Kuda project respectively, throughout these chapters our splitting framework has been contrasted with traditional race checkers. In Chapter 5, experimental results from the application of Kuda to benchmark programs are described. Finally, in Chapter 6 we introduce some future work and conclude the thesis.

Chapter 2

RUNTIME VERIFICATION

The purpose of this chapter is to point out the key challenges one is likely to face while doing research on runtime verification (see 2.1), precise race detection using vector clocks (see 2.2.1), some important runtime verification algorithms for detecting data races (see 2.3) and significant research directions one is likely to take when building a runtime verification tool for concurrency-related errors (see 2.4).

2.1 Challenges in Runtime Verification

The main challenge for implementing a runtime verification tool originates from the aim for generating the minimum runtime overhead of the monitoring on the application without sacrificing the completeness of detection. Completeness of a runtime verification tool can be analyzed by its ability for catching concurrency errors. A verification tool can be: complete (sound and imprecise), sound but imprecise, unsound but precise or unsound and imprecise. A complete runtime verification tool is able to report a violation iff it occurs during runtime, an unsound and imprecise tool generates false errors and misses some errors during runtime. Data race detection is one of the most important branches of runtime verification. We will limit this thesis to data race detection tools.

2.2 Race Detection

A race detection algorithm for concurrent programs maintains data structures shared among the threads participating in the algorithm (It is common to name the content of these data structures as meta-data). For example, the seminal Eraser [34] paper has an implementation maintaining a lockset for each thread and for each shared variable; yet there are other algorithms which maintain pointers to the last accessing thread or an additional virtual-clock vector to internal locks for synchronizing accesses by different threads. Any race detection algorithm needs to use proper synchronization to ensure consistent accesses to the algorithm-specific data structures 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. Overheads induced by runtime tools actually impact the responsiveness of the target application and also influences its concurrency characteristics. Implementing an online monitoring system for such programs requires a considerable amount of engineering effort in order to reduce the runtime overhead of the monitoring and thus to have the minimal impact on programs runtime behavior. Apart from the runtime challenges of such monitors, precisely detecting all possible data races (i.e. without any false detection) within a program's execution is an NP-Hard problem as shown in [27]. However, dynamic race detection tools check for races only during an actual (or speculative) execution of the target parallel program.

In the following section, we introduce you to Eraser race detection algorithm and traditional implementation on the CPU. Details of Goldilocks and FastTrack algorithms are available in our references [9, 12, 8, 6, 7].

2.2.1 Formal Definition of Data Races

There exists several definitions of data races. We respect the C++11 specification: The execution of a program contains a data race if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither *happens before* the other. Any such data race results in undefined behavior. Two memory accesses conflict if they access the same memory location (e.g. variable), and at least one access is a store instruction. The notion of *happening before* partial ordering of actions (or events) can be captured by the happens-before partial order on a program execution, defined in Lamport's seminal paper [19], which can be represented as a directed acyclic graph. A data race occurs when two actions are not ordered via happens-before dag -they occur *simultaneously* during execution of the program. A program is said to be data-race-free (on a particular input) if no particular execution results in a data race. However, proving data-race-freedom is an NP-Hard problem and it is beyond scope of this thesis. In this thesis we are focusing on detecting races occurring online, i.e. on-the-fly detection during an execution. Now we will provide a formal definition of detection of races via happens-before relation:

2.2.2 Segments of a thread [1]

Creation or termination of threads and communication among threads are all controlled by executing certain special sequences of instructions called *synchronization* *operations*. The synchronization operations on a given thread are executed in a definite order. These operations are used to partition the thread into parts called *segments*. A segment of a thread is a maximal sequence of instructions containing exactly one synchronization operation that ends the sequence. A synchronization operation executed on a thread T is a posting operation, if it posts some information carried by T that can be read later by T itself or by some other thread. A synchronization operation executed on a thread T is a receiving synchronization operation, if it reads the information already posted by one or more posting synchronization operations. A given synchronization operation cannot both post and receive. However, the information posted by a single posting operation can be read by more than one receiving operation, and a given receiving operation may read information posted by more than one posting operation.

Consider a thread T and label the distinct sync ops on it by A_1, A_2,..., A_n in the order of their execution. Let S_k denote the segment defined by A_k, where $1 \leq \mathrm{k} \leq \mathrm{n}$. For $2 \leq \mathrm{k} \leq \mathrm{n}$, the segment S_k consists of the sequence of instructions between A_{k1} and A_k, including A_k and excluding A_{k-1}. The first segment S_1 consists only of the instructions belonging to the sync op A_1. The segments of T are executed in the order: S_1, S_2,..., S_n. In a given segment S_k, the instructions are executed sequentially. The sync op A_k is always executed last, but the other instructions in S_k, if any, are executed in an unspecified order. No instructions of S_k are executed before the execution of A_{k-1} is complete.

A segment on a thread knows exactly how many segments on that thread have already executed. However, it has only a partial knowledge of how many segments on a different thread have already executed. Any such knowledge that the segment has is based upon the information, if any, that its thread has already received by executing receiving synchronization operations. Since a global clock is not available, we have to base our analysis on this incomplete knowledge.

2.2.3 Happens-before Relation, Vector Clocks and Precise Race Detection

We now define happens-before, a partial order relation $\prec$ between segments. Let S denote a segment on a thread T and S$'$ a segment on a thread T$'$. We have S $\prec$ S$'$, if one of the following holds:

1. T = T$'$ and S is executed before S$'$.

2. S is a posting segment on T defined by a synchronization operation A, S$'$ immediately follows a receiving segment on T$'$ defined by a synchronization operation B, and B reads the information posted by A.

3. There exist a finite sequence of segments S_0, S_1,..., S_m in the program with S = $S_0 \prec S_1 \prec ... \prec S_{m-1} \prec S_m = S'$,

such that for 0 $\preceq$ p $\preceq$ m1, the relation $S_p \prec S_{p+1}$ holds in the sense of either Condition 1 or Condition 2. If S $\prec$ S$'$, the segments S and S$'$ must be distinct. As usual, the notation S $\preceq$ S$'$ means either S $\prec$ S$'$ or S = S$'$. It is clear that $\preceq$ is a partial order on the set of segments in the program.

If $S \prec S'$, then segment S must finish executing before segment S' starts (in the particular program execution under consideration). Even if S finishes executing before S' starts, the relation $S \prec S'$ may or may not be true. If S and S' overlap or S finishes before S' starts, then $S' \prec S$ is false. If neither $S \prec S'$, nor $S' \prec S$ is true then S and S' are said to be parallel segments.

Consider any segment S on any thread T in the program. Let T_S denote the set of all threads known to S. This set consists of T itself, and every other thread T' with a posting segment S' such that $S' \prec S$. The vector clock of S is a function $V_S : T_S \rightarrow 0,1,2,...,\infty$ defined as follows: For all $T' \in T_S$, $V_S(T')$ = [Number of posting segments S' on T' such that $S' \prec S$].

Theorem 1.

Let T and T' denote two distinct threads, S a segment on T, and S' a segment on T'. Then $S \prec S'$ if and only if $V_S(T) < V_{S'}(T)$.

Corollary 1.

A segment S on a thread T is parallel to a segment S' on a thread T', iff $V_S(T) \geq V_{S'}(T)$ and $V_{S'}(T') \geq V_S(T')$.

Theorem 2.

A segment S on a thread T is parallel to a segment S' on a thread T'. Let R_S denote the set of memory locations read and W_S the set of locations written by the segment S, and similarly for the segment S'. Then there is a data race between S and S', iff the following two conditions hold:

1. $V_S(T) \geq V_{S'}(T)$ and $V_{S'}(T') \geq V_S(T')$;

2. Either $[\mathrm{R}_S \cup \mathrm{W}_S] \cap \mathrm{W}_{S'} \neq \emptyset$, or $[\mathrm{R}_{S'} \cup \mathrm{W}_{S'}] \cap \mathrm{W}_S \neq \emptyset$.

2.3 Some Race Detection Algorithms

Dynamic race detection algorithms can be divided into two subcategories: lockset based and vector clock based. Both approaches have the ability to completely detect data races in any environment. Eraser and Goldilocks are lockset based race detection algorithms, and FastTrack is a vector clock race detection algorithm. Throughout their evolution, lockset based approaches were seen as fast, sound yet imprecise approaches, which basically generates numerous false positives; rendering them less useful for programs with many shared data. Actually, there exists a side research to prioritize the error reports of such tools, utilizing source code characteristics, i.e. static analysis. On the other hand, vector clock based approach was regarded as complete yet slow one. However FastTrack authors have engineered this approach so well that the performance of the race detection is currently known as the state of the art since the last couple of years. As in RaceTrack, there exists hybrid approaches, combining a phase of lockset based algorithm and then applying vector clocks and thereby achieving a third kind of race detection [46, 30].

Completeness of data race detection can also be tweaked by changing the granularity of race detection. However, in this thesis we are only interested with word granularity data race detection tools. Some algorithms adjust the granularity of shared variables between collection of variables (objects, arrays) to individual memory cells [44, 46]. Increasing the granularity of the checking significantly decreases (at least an order of magnitude) the computational cost of both instrumentation and analysis.

2.3.1 Eraser *algorithm*

Eraser is a well-known lockset-based algorithm for detecting race conditions dynamically [34]. Variants of Eraser in the literature use additional mechanisms such as a state machine per shared variable in order to handle special cases such as thread locality and object initialization patterns [34, 44, 46]. To detect race conditions, Eraser enforces the locking discipline that every shared variable x is protected by a common lock throughout the execution. If this assumption holds for every variable in the execution, then the execution is declared race free. The other direction of this statement is not valid: There are race free programs violating this assumption, which makes Eraser sound yet imprecise.

For simplicity of presentation, we focus on the core algorithm using generic 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. The Eraser algorithm maintains 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. If this intersection becomes empty, this means that x has not been consistently protected by the same lock, thus a race is reported. 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 callback function to perform 2), or saved to a temporary buffer to be processed later (e.g., in a linked list of events as in [8]). There are two main sources of runtime cost, which combined together contribute highly to the overhead of the monitoring on the application:

2.3.2 An Eraser *implementation*

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. 5.2 show that only instrumenting the program (without performing any computation at instrumentation points) can generate 3x overhead on the un-instrumented program.

In order to accelerate accessing to these data structures, it is a well-known technique to distribute the structures to appropriate locations in over memory [46, 8]. For example, locksets $LH(t)$ can be attached to thread t using thread-local storage pointers and $LS(o.f)$ for fields f of an object o can be attached to object o by adding an extra field. Many instrumentation framework including Pin [21] provide utilities to attach an auxiliary pointer to a thread object that can be fetched and updated throughout the execution, that pointer refers to the data structures that are used when processing the events of that thread. fetching the locksets from these maps may require expensive hash computations with expensive memory operations and nontrivial arithmetic operations. It is also possible to filter the instrumentation points by static analyses, for example, to identify thread-local variables, but this creates extra effort, e.g., code annotations as in [11], and often imprecise in the presence of complex sharing and aliasing.

For Eraser, accessing the map LS (or $LS(x)$, if the map is distributed) requires 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, their implementations can cause considerable runtime and memory overhead and this makes the race checking hard to apply at the post-deployment.

The programmer has to spend a high amount of effort to make use of nontrivial and often error-prone mechanisms. For example, a naive Eraser implementation requires a large memory space to store locksets when there are a high number of shared variables. Moreover, it has to manage the locksets in the case of dynamic and frequent allocation/deallocation of threads and variables. In order to reduce the runtime and memory cost of the algorithm, the programmer has to develop a highly optimized implementation. A clever way of reducing cost of analysis is to filter out events (memory accesses) that are known not to result in a data race with any other event. In [33] authors have developed three filters for such purpose. Another example, to reduce the number of memory allocations for locksets, locksets of deallocated variables is reused for newly allocated variables, requiring a memory pool of locksets. These and similar extra management tasks are usually nontrivial and error-prone to implement and if not implemented carefully, may create extra overhead on the core algorithm. This results in highly complicated implementations for very simple algorithms such as Eraser.

2.4 Previous Research on Race Detection

Both in FastTrack [12] and Goldilocks [8] provide a complete race detection tool for applications running on a managed environment, Java in this case. Other complete data race detection tools are available, Radish [5]. However there exists wide variety of research tools that favor performance over completeness of race detection. Such incomplete yet faster tools fall into three categories: sound but imprecise Eraser [34], Racetrack [46], and MultiRace [32]; unsound but precise Kuda [2]; both unsound and imprecise Racez [37] and DataCollider [10].

RaceTrack [46] employs adaptive techniques for tracking properties, meta-data and granularity contexts while using a hybrid detection algorithm. The potential races that are generated in the coarse-grained approach are used for handling adaptation, which leads to refinement; secondly threadset technique has the ability to ignore race checking on safe variables. Racetrack is embedded to .NETs VM CLR where RaceTrack exploits the tools framework. Plus it also employs post-processing of potential race warnings. Experiments show that the detector does 3x slowdown for worst, but typically 2x.

ThreadSanitizer [36] is another sampling data race detection tool developed by Google. It is built on LLVM and therefor it is based on low-level compiler instrumentation. Another important tool from Google's research team is RACEZ [37]. RACEZ uses the sampled memory trace collected by the hardware Performance Monitoring Units (PMUs) for race checking via a lockset based detection algorithm. RACEZ is the first race detection tool that uses PMUs for instrumenting. Thread library functions, are wrapped so that each synchronization call updates thread local lockset information. Whereas all memory operations (includes locks, mallocs etc.) are collected by the PMUs and then this two separate bookkeeping are correlated. When an application thread I created by the wrapped call, PMU interrupts are bound to each using a file descriptor, creating PMU context. Then a system call inside a thread starts the self-monitoring process. When a PMU samples an event it creates an interrupt, when the buffer is read PMU closes and waits for a restart. Sampling period and buffers can be configured at each start. Therefor PMUs can be configured at runtime, which enables dynamic optimizations. Sampling period can be configured on the fly. Racez basically delivers a signaling mechanism to multithreaded applications for self-monitoring. Hardware seamlessly keeps track of sampled memory accesses and signals threads to collect it. Race detection is done post-mortem and both unsound and imprecise but contains stack traces. This approach leads down to a surreal 0.02x overhead on average to the target application under average sampling.

Authors of DataCollider [10] have implemented a race checker, oblivious to synchronization protocols, right into the OS kernel. It samples memory access data via random code breakpoints and captures the racing thread red-handed via data breakpoints. DataCollider race checker uses a repeated-read strategy. It reads the value once before and once after the delay. A change in value is an indication of a conflicting write, and hence a data race. DataCollider incurs very low overhead while using sampling option.

RaceFuzzer [35] is composed of two phases; first a race detection algorithm, second phase is the actual fuzzing. This work attempts to force entire reported race interleaving during testing in order to separate false positives from true race bugs. RaceFuzzer instruments Java byte code to observe various events and to control the thread scheduler. The algorithm takes set of potential races from first phase then, basically second phase actively controls a randomized thread scheduler of concurrent program based on potential data races discovered by an imprecise race detection technique. Therefore RaceFuzzer is actually a race-directed random testing based on previously known potential races. That leaves RaceFuzzer out of our focus. Experiments show that tool does 3x slowdown, 1.1x at best. But with the exception of an order of magnitude slowdown for HPC programs.

HARD [48] is the first hardware implementation of the lockset algorithm to exploit the race detection capability of this algorithm with minimal overhead. HARD efficiently stores lock sets in hardware bloom filters (it is similar to our kernel implementations, see 4.4) and converts the expensive set operations into fast bit- wise logic operations with negligible overhead.

Kuda has three different race detection algorithms available in its core. Keep in mind that analysis can be extended with other detection algorithms, not limited with race detection, like atomicity checking [47, 18, 31] and other security checks [28]. Kuda using Goldilocks race detection algorithm is an unsound but precise race detection tool, meaning the result of race detection may contain false negatives (some data races might be missed) but any error given by the tool is actually a data race. Eraser is an algorithm that is sound but imprecise, meaning it does not miss any data race yet it may alert the user even when there are no data races occurred. Kuda using Eraser race detection algorithm is both unsound and imprecise race detection tool. The reason Kuda introduces unsoundness to runtime verification is left out for the following chapters. Kuda also has a FastTrack kernel, however it is a very limited implementation compared to the other two, due to commodity GPU's current architectural limitations such as very limited shared memory.

Our separation of instrumentation and analysis allows one to focus not on optimizations but a simple implementation of the core algorithm on highly efficient cores, available on commodity GPUs. 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. 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.

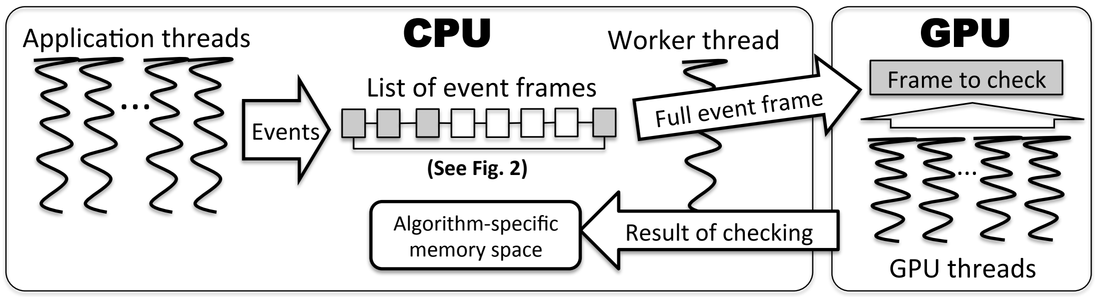

Figure 3.1: Components of our runtime monitoring system.

Chapter 3

EXTRACTION OF TARGET APPLICATION'S EXECUTION TRACE

Our main goal is to design a runtime verification framework that will have the cost of analysis for race detection to have minimum negative impact on the program's runtime behavior. Our key design decision is to carry out the checking algorithms on physically separate multi-processors, in our case the GPU cores.

3.1 Binary Instrumentation

In this section, we present our techniques for observing the target application's execution trace, i.e., recording events and communicating them to the GPU. The following chapter will complement this by introducing GPU-based algorithms for data race detection. Binary instrumentation is needed to gather the necessary data on the execution of a program. There exist several industry strength frameworks that are open to us researchers: Intel® Pin[21], Valgrind[25], DynamoRIO[38] and for managed environments there is RoadRunner [13] among others.

In order to instrument the binary and gather the execution trace, we have developed two distinct toolkits for two separate runtime environments:

3.1.1 Native binaries

For native binaries, in Kuda, we are using Intel® Pin [21]™dynamic instrumentation tool for attaching hooks to the events occurring within the target application. Pin was originally created as a tool for computer architecture analysis, but its flexible API and an active community (called "Pinheads") has created a diverse set of tools for security, emulation and parallel program analysis. Intel® Parallel Suite™is a pack of commercial tools for parallel programmers that uses Pin tool as their backbone. Such tools are commonly called as Pintools. Pintools can be used to perform program analysis on user space applications in Linux and Windows™. Thus, it requires no recompiling of source code and can support instrumenting programs that dynamically generate code. Pin provides a rich API that abstracts away the underlying instruction-set idiosyncrasies and allows context information such as register contents to be passed to the injected code as parameters. Pin automatically saves and restores the registers that are overwritten by the injected code so the application continues to work. We have chosen to work on top of Pin since an application instrumented with Pin observes the same addresses and registers for program code and data as it would were it running without Pin, it was the perfect framework for achieving data race detection. And the authors of Pin have shown that Pin is faster than other systems such native binary instrumentation framework. As for our concurrency primitives on native code, inside our Pintool, we attach hooks for pthreads[15] library calls.

3.1.2 Java byte-code:

For instrumenting Java applications, in Kuda, we are using RoadRunner [13]. It is a dynamic analysis framework designed to facilitate rapid prototyping and experimentation with dynamic analyses for concurrent Java programs. It provides a clean API for communicating an event stream to back-end analyses such as in Kuda, where each event describes some operation of interest performed by the target program, such as accessing memory, synchronizing on a lock, forking a new thread, and so on. Java events are passed to our native Kuda interface via JNI, the Java Native Interface. However, such interface turned out to be the bottleneck of our framework due to high volume of data transformations, therefore we chose to limit our demonstration only limited for native applications.

3.2 Kuda ***Interface***

In our Kuda framework, target application threads running on the CPU, under a managed environment or not, are only responsible for recording their events in a shared data structure. Communication cost of these events to the GPU for further processing is a burden for a worker thread apart from the application threads. Fig. 3.1 illustrates this separation of responsibilities between the CPU and GPU threads. This was one of our goals successfully achieved. Detailed performance results are left for another chapter.

Note that, our splitting framework gives path to future realizations of highly efficient instrumentation and monitoring infrastructure independent of target application's environment and independent of targeted analysis offloaded to the GPU. There exist several important line of work, in [42] using inexpensive hardware support for acquiring and communicating the execution trace. And in [22] GPU is utilized for highly parallel garbage collection. Work in [28] can also be adopted such that the parallel security checks could be done inside a GPU. Our goal was to utilize commodity hardware environment with a modern GPU.

3.2.1 Event Bookkeeping

In KUDA, we basically linearize all events in the execution. For this, we use a monotonically incrementing global counter to assign a global order to each and every caught event. Our system requires no more synchronization to establish or observe the linearization. While observing only a partial order (of synchronization operations) would be sufficient for soundness. Instead of analyzing the entire execution, we observe an execution as a linear sequence of events, we call the trace of the execution. While multiple linearizations of the events may represent the same concurrent execution, our system observes one of these linearizations.

Our technique is based on logging the execution as a linear sequence of *event*s and running the analysis in a very efficient, parallelized way. Here, we overload the term *event* as a data structure that carries out the identifier of the executing thread, and the kind of the event (memory read/write, lock acquire/release, etc.), and a data word relevant for the kind of the event (e.g., the address of the memory accessed or the identifier of the thread created). In order to enable efficient handling of the event log, we only process 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 *frame* as 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 are discarded and it is reused to store later events. While our implementation allows enlarging or shrinking this list, our experiments show a small number of blocks are sufficient to track without any need to modify the list.

While splitting the linearized execution into chunks of independent frames may cause unsound results due to lost pairings of events from separate frames, our framework allows to adjust the FrameSize to increase the chance of finding bugs while losing some performance. 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. We overlap our frames so that the splitting of two events in consecutive frames is always a FrameSize away. 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 [24].

Fig. 3.2.1 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. 3.3 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 application threads perform the former, we dedicate a separate worker thread (running on the CPU) for the latter.

3.2.2 Recording of events (RecordEvent in Fig. 3.3)

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. 3.3) 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 grey). 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. 3.3). In order to prevent data races on *Head* and *Tail*, we read from and write to these variables by acquiring locks associated to them. These locks are implemented as a

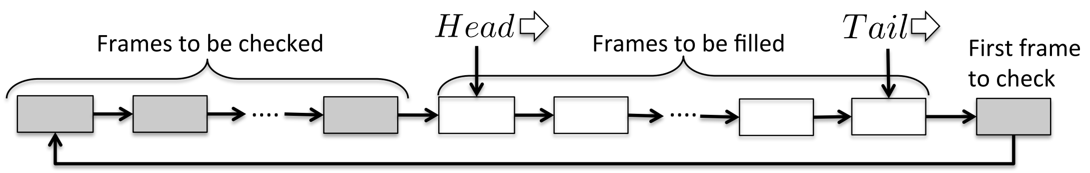

Figure 3.2: The cyclic linked list of event frames

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 ($index \geq FrameSize$) {
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 ($index = FrameSize - 1$) { $Head$ := $Head.next$ }
```

Figure 3.3: RecordEvent pseudocode

spin lock. Another approach for preventing data race to these pointers was to have atomic operations, ergo having a lock-free cyclic linked list, however our performance

```
Algorithm CheckFrames()
(Executed by worker thread)
1  while (program is running) {
2      wait until $Head \neq Tail$
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 }
```

Figure 3.4: CheckFrames pseudocode

analysis showed that such an approach has scalability and therefore low throughput issues when number of concurrently accessing threads become more than 4.

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 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 with number of benchmarks, shifting *Tail* is happens more than shifting *Head*, so that there is always at least one empty slot to insert an event between *Head* and *Tail*. We keep a global counter, called *next_index*. For each frame, we keep the minimum and maximum values for indices of events that frame will store. That is, the checking phase of reads the current value of *next_index*, say *index* and increments it by one. It then locates *index* in the event list. For this, it traverses the list starting from *Head* and going towards *Tail*. The thread while traversing the list to figure out which frame to add the event uses these bound values. Notice that, this is in fact a one-level implementation of a B-link tree. Thus, the event list could also be organized as a tree to accelerate the location of the frame to insert the event. 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).

3.2.3 Processing of frames: (CheckFrames in Fig. 3.4)

When the event list is first initialized, At the initial state of the event list *Head* and *Tail* points to the same frame. While event frames become full of events, *Head* starts to shift forward as explained above. Thus, the frames between *Tail* and *Head* become full frames to be checked. Our worker thread takes a full frame a time and sends it to the GPU for the checking (in Fig. 3.4 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).

We have the ability to apply separate *helper* threads for consuming the full frames other than the worker thread. These helper threads pick up a frame and do race checking on the CPU. This option can be enabled throughout the execution or when *Head* comes close to the *Tail*. Since our goal is to emphasize parallel race detection on GPU, we have disabled helper threads.

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.

Chapter 4

DYNAMIC RACE DETECTION ON GPUS

Having introduced the CPU part of our framework, we will now present customized algorithms for the data race detection, running on the GPU cores. In section 4.1 we first brief on GPU computing on CUDA™and then in section 4.2 using the CUDA™model and after that in section 4.3 we introduce the author to the challenges that affected our design. Then, in section 4.4 we present our adaptation of Eraser, Goldilocks and FastTrack algorithms to run on thousands of GPU threads.

4.1 Introduction to CUDA™

CUDA™ [29] is a parallel computing platform and a programming model created by NVidia®and implemented by the GPUs that they produce. A GPU provides a highly parallel, multithreaded, many-core processing environment with a hierarchical memory model isolated from the main memory. Todays GPU hardware contains hundreds of processors and a much higher memory bandwidth than CPUs. For example, the Quadro™4000 card we used contains 8 multiprocessors with 32 cores each, totally giving 256 cores running at 1.404 MHz memory clock and 950 MHz processor clock, and 2 GB of memory space with 89.6 GB/sec bandwidth. Theoretical limit of our GPU is 486.4 gigaflops at single precision. An ultra High-end CUDA™GPU available today uses the state of the art Kepler™architecture having 2688 cores, with 6 GB of memory space with 250 GB/sec bandwidth, making the theoretical limit of this ultra GPU a 3.95 teraflops at single precision.

4.2 CUDA™programming model

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 and easily scale out to multiple GPUs.

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 given steps in Fig. 3.4 to run an analysis on a full event frame.

The worker thread first copies the contents of the event frame to the pre-allocated

```
my_kernel<<<numBlocks, threadsPerBlock, ...
        amountOfSharedMemoryPerBlock, numStreams>>>(A, B, C);
```

Figure 4.1: A generic kernel launch

region on the GPU device memory, called the host memory. Then it calls the kernel function of an available checker algorithm. The kernel function is executed by the GPU cores in parallel and asynchronously with the application threads running on the CPU, only the worker thread will wait for the kernel to respond back the analysis results. When calling the kernel function, it passes as arguments the pointer to device memory region storing the current frame, number of thread blocks, number of threads per a thread block, the number of events in the frame, the amount of shared memory per thread block and number of streams (see 4.2). Each kernel is executed with a SIMD style on multiple cores and threads. The CUDA™model allows one to execute the kernel on a virtual organization of threads, independent of the number of cores on the device. This enables scalability and transparency without modifying the code. In order to engineer the organization of threads, there is a tool called Occupancy Calculator. The CUDA Occupancy Calculator allows you to compute the multiprocessor occupancy of a GPU by a given CUDA™kernel. The multiprocessor occupancy is the ratio of active threads to the maximum number of threads supported on a single multiprocessor of the GPU. Each multiprocessor on the device has a set of registers and shared memory available for use by CUDA™program threads. These are shared resources that are allocated among the thread blocks executing on a multiprocessor. The CUDA™compiler attempts to minimize register usage to maximize the number of thread blocks that can be active in the machine simultaneously. However, in some special cases like in Kuda, having a maximum number of active threads per multiprocessor is not the optimal solution. In Kuda, a single thread has to access the memory in a very high frequency, which means having more registers allocated for each thread. The reasoning behind this phenomenon is explained thoroughly in [43]. A single sentence explanation would be that the characteristic of any race detection algorithm actually favors lower arithmetic and memory latency, over higher parallelism.

In Fig. 3.4, since the GPU kernel executes asynchronously with the CPU, the worker thread can spend the time until the kernel call terminates to mark the currently checked frame empty (line 6) and makes the frame available to be reused to recording new events (line 7). The worker thread uses CUDA™routines to synchronize with the kernel execution for further processing. 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 to be reported later.

4.3 Software Engineering Challenges on the GPU

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

4.3.1 Isolated execution

Each GPU thread gets a unique thread identifier among the other threads. This unique id allows the thread to determine parts of the event frame it should work on without interfering with other threads. Explicitly transferring the inputs and outputs of the kernel between the CPU and GPU creates an extra communication overhead for each kernel call (newest high-end HP motherboards have a customized communication bus in between capable of multiple GPUs just to tackle this problem). 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. In order to reduce the communication overhead, we encode each event in two 32-bit words, storing the event kind, the identifier of the thread generating the event, and a memory address (of variable read/written, or of mutex locked/unlocked) relevant to the kind of the event. Any other data not used in the checking is maintained separately and not sent to the GPU. For example, for each event, we keep the address of the instruction that generated the event, and use it for debugging errors in which event involves in, relating the event back to the source code location that generated that event.

4.3.2 Limited device memory space

Although state-of-the-art GPU devices can have more than 2GB of memory and latest CUDA$^{\text{TM}}$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. Dynamic memory management will cause considerable overhead on the performance of the checking. Even more importantly, in today's commodity PC setups there exists a single GPU that has only 256MB of device memory. Having a 200MB of a frame would cripple the OS processes, which are working to keep up with the display. This again brings some limits on size of data structures kernels use. 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. We have experienced some minor glitches on the screen especially during the operation of KUDA. It should be noted here that in order to debug our KUDA kernel functions, we had to have a separate GPU for Xserver to run on. CUDA-gdb currently does not allow their user to attach hooks to a running kernel, without having to stop another kernel's execution.

4.3.3 Enforcing SIMD model

CUDA™enforces the SIMD execution model rather than the typical multiple-instruction model in the CPU. In the GPU, threads are divided into multiple groups (called *warp* in CUDA™) and warp of threads are co-scheduled on a streaming multi-processor and execute the same instruction in a given clock cycle. SIMD style execution of the kernel forces the developer to implement her algorithm in a special form of data-parallelism in order to get benefit of hardware parallelism. For example, since the GPU is design to run the same instructions on all threads simultaneously, control flow instructions (e.g., `if`, `while`) affect the instruction throughput by causing diverging paths to be serialized. Thus in [29] it is highly recommended that different execution paths in the kernel be avoided and the branches in the code be implemented by distributing data to different threads and using thread and frame identifiers to access different memory regions or do different computations.

4.3.4 Limited shared memory and limited shared memory functionality

Although it is not common to have nontrivial data structures such as a hash table inside the shared memory, we had to go through with it for our FastTrack implementation. CUDA^{TM}shared memory is said to be dynamically allocatable, however it is not true, especially in the sense of how traditional software engineers expect from a shared memory. When launching a new kernel you can only set the amount of shared memory that a thread block should have, and basically that's it. You cannot dynamically allocate a data structure, a thread block has a shared pointer across its threads, to the top of the allocated shared memory. It would be more fitting to call it a shared register file. The challenge that it brings to the CUDA^{TM}developers is that in order to allocate and work with the data structures, you should build your kernel for the worst case. Meaning without having the capability to dynamically adjust, allocate and free, your data structure nodes, the maximum amount of nodes that you could allocate inside the shared memory is your closest limiting factor. Since a hash table is the building block of vector clocks, a FastTrack running on a GPU can only be a very limited version; the target application you are monitoring for data races can only have a couple of threads running parallel and the FrameSize parameter should be chosen carefully so that the vector clocks could never exceed the available, tiny shared memory consisting of just a couple of KBs.

4.3.5 Memory and arithmetic latencies

A very common computation pattern for a GPU kernel is to access the device memory once (or twice) for input and some small amount of computation on top of a small matrix of pixel data. But, for race detection, each kernel thread has to access all of the events beyond their corresponding initial event (see section 4.4). This memory access pattern is very unusual for a kernel and puts a lot of stress on the memory controllers. We have experienced serious memory latencies for fetching the events inside a frame and decided to bring down the frame to the thread blocks by utilizing shared memory. This has greatly improved the overall performance of our kernels. Secondly, regarding arithmetic latency, a thread has to compare an event to forth-coming events at each iteration of a for loop. If the algorithm falls into the path that it should modify the meta-data accordingly it stresses the arithmetic capabilities of the GPU. Since threads in the same warp have to work with the same instruction the arithmetic latency becomes worse than a regular CUDA$^{\text{TM}}$kernel, plus it adds workload imbalance due to branching and nonuniform memory accesses. A possible counter measure for this behavior is creating artificial thread level and instruction level parallelism to hide the latency (by checking races for separate events inside the same loop); but a successful implementation for any input is very tricky and could have resulted in worse performance than the original. We decided to avoid an even more complex kernel code.

4.4 Data Race Detection on the GPU

Given the challenges in writing kernels, we wrote parallel kernels for the Eraser, Goldilocks and FastTrack algorithms. Fig. 4.4 and Fig. 4.4 gives the pseudocode for two of these kernels, FastTrack kernel could not be used for real applications so we defer its explanation to our open source code, available online. The implementations of the kernels are also available at `http://kuda.codeplex.com`.

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. 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 bounded sized 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.

Bloom filters can represent a set of words in a constant space, i.e., fixed-size bitsets.

Bloom filters allow insertions and lookups but not removal of existing elements. In fact, our algorithms do not require removal from locksets, but only additions and lookups. While being efficient, bloom filters make us sacrifice soundness. It might response positive to a lookup query for items, even when it is not contained in the set. As a result, our algorithm can incorrectly decide that a lock is in the lockset of a variable and thus miss data races, hurting the soundness of our checking. Nevertheless, we are already sacrificing soundness because of analyzing execution in independent frames, and the bloom filter does not affect precision of our analysis, which is our primary goal.

Each kernel in Fig. 4.4 and in Fig. 4.4 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. Both kernels transfer their portion of the frame from device memory to its shared memories (not included in the pseudocode). 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. [1] 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). Initializes a thread-local lockset as an empty set at line 13. LS stores the locks that are released by the first accessor thread ($e.tid$).

The thread traverses the events (denoted e'') between e and e' and update LS (lines 17-18. If the second accessor thread ($e'.tid$) acquire one of the locks released by the first accessor thread, the flag $racy$ is set to $false$ and the traversal ends (lines 19-22). Upon completion of the traversal at line 24, the thread checks the $racy$ flag. sIf $racy = false$ 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. If $racy = true$, then the pair (e, e') is added to the set of racy event pairs (line 25).

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 [8]: 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 [8], 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).

Algorithm **EraserKernel** — **Input:** *Frame* – Array of events. **Output:** *DataRaces* – Pairs of racy events.

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

Figure 4.2: Kernel code for Eraser

Algorithm **GoldilocksKernel** — **Input:** *Frame* – Array of events. **Output:** *DataRaces* – Pairs of racy events.

```
$id$ := $threadIdx.x$ // Get thread id
$e$ := $Frame[id]$ // Fetch event
if ($IsRead(e.kind)$ or $IsWrite(e.kind)$) { // check if this is an access event
  $id'$:=$nil$
  for $i = id$ to $Frame.size$ do
    $e'$:=$Frame[i]$
    if $IsRead(e'.kind)$ or $IsWrite(e'.kind)$
      if $e'.value = e.value$
        $id'$:=$i$
        break
    end if
  end for
  if $id' \neq nil$ // if there is another access, check that access
  $id$ := $threadIdx.x$ // Get thread id
  $e$ := $Frame[id]$ // Fetch event
  if ($IsRead(e.kind)$ or $IsWrite(e.kind)$) { // check if this is an access event
  $id'$ := -1 // find out the next access to the same variable
  for ($i = id$ to $Frame.size$) {
    $e'$ := $Frame[i]$
    if ($IsRead(e'.kind)$ or $IsWrite(e'.kind)$) {
      if ($e'.value = e.value$) {
        $id'$ := $i$
        break
  } } }
  if ($id' \neq$ -1) // if there is another access, check that access{
    $LS$ := $\{e.tid\}$ // initialize a local lockset for e.value
    for ($i = id + 1$ to $id' - 1$) // run rules of the algorithm{
      $e''$ := $Frame[i]$
      if ($IsAcquire(e''.kind)$ and $e''.value \in LS$) $LS$ := $LS \cup \{e''.tid\}$// Process operations of kind Acquire
      if ($IsRelease(e''.kind)$ and $e''.tid \in LS$) $LS$ := $LS \cup \{e''.value\}$ // Process operations of kind Release
    }
    if ($e'.tid \notin LS$) // perform the check for data race
      $DataRaces$ := $DataRaces \cup (e, e')$
} }
```

Figure 4.3: Kernel code for Goldilocks

Chapter 5

EXPERIMENTAL SETUP

In this chapter, we describe our efforts to experimentally evaluate our runtime verification approach. We aim to evaluate two claims we referred to in our introduction:

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.

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 a prototype tool called KUDA and applied KUDA on a collection of multithreaded benchmarks. KUDA is open source and available at `http://kuda.codeplex.com`.

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$^{\text{TM}}$4.0 library [29] 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 [21] 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 [15] (for thread creation and join, and synchronization primitives including mutex and readers/writer locks). Our tool instruments pthread calls at the image level; i.e., inserts callbacks at the beginning and/or end of relevant pthread functions when the pthreads library is loaded. Other instructions for memory accesses and function calls and returns are performed at the trace level (inserts callbacks right before executing a single entry multiple exit code block).

5.1 Experiments

We applied our tool Kuda on a collection of multithreaded programs from PARSEC [4] and SPLASH-2 [45] benchmark suites. 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. 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.

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. 3.2.1 with 2048 frames. Thus, our system requires only 2048 frames * 1024 events (each frame) * 8 bytes (each event) = 16 MB of memory space to store the events for the CPU. We run 400 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 5.3 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

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
`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 5.1: 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 [3].

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

Warp size	32
Memory bandwidth	102.6 GB/sec

GPU device features

Device memory to store event block	Shared
# events in each block (unit of work)	1024
# blocks sent to GPU	128
# GPU threads per event block	400
# event blocks in memory	32

Configuration parameters

Table 5.2: GPU device information and parameters that give the best results.

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 and 7.1X.

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$^+$ [32] and FastTrack [12] (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.

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 5.3: Results from our experiments. Running times are given in seconds.

5.3 Analysis

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. Note that, the slowdown when running such a simple algorithm starts from 31.4X. 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.

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. For this paper, we considered all the relevant events in the execution. We believe that static pre-analysis of the program for identifying race-free variables/accesses can reduce our overhead significantly down to tolerable limits as in [8].

Lastly, 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.

5.4 Unlisted experiments

Apart from the experiments presented in this thesis, we have also implemented an interface for KUDA to work on Java bytecode using ROADRUNNER; experimented on Java Grande Benchmarks. We have done intensive optimizations, both on the CPU and GPU runtimes of Kuda. In the CPU, using PIN, we have experimented on different buffering mechanisms, event filtering mechanisms, scalability improvements and utilization of extra threads for concurrent in-place race detection to achieve a maximum throughput of the event production and therefor race detection. In the GPU, using CUDA API, we have also experimented on customised race detection algorithms such as FastTrack, utilized shared memory for fast access to race detection metadata and reusing of input frame without sacrificing more registers, experimented on different device memory access options such as texture memory and finally another line of experiments was done on utilizing concurrent streams for asynchronous kernel execution.

It is also important to note that the experiment setup has changed after the results have been published; a new HP®z600 workstation was used for new experiments. Although the results of those experiments were showing a similar payoffs (5x speed up over tradition race detection), raw data collected on that machine was showing very high overheads for both traditional and parallelised race detection. Two reasons for this bad behaviour is understood. Firstly, the dual CPU setup has resulted in cross-CPU serialization, and secondly availability of enough hardware threads for the benchmarks to run in highly parallel manner resulted in a higher slowdowns on our monitoring mechanisms (i.e. both lockset and vector clock race detection implementations on CPU and our event buffering mechanism).

Chapter 6

CONCLUSION

In this thesis, we present parallelized runtime verification algorithms and their implementations for GPGPU's, a byte-resolution binary instrumentation tool for x86 programs with pthreads and a cyclic linked list with two implementations, a lock-free and a spin-lock implemented versions with the ability to be accessed and modified concurrently both from the head and the tail, and a couple of traditional runtime verification algorithm implementations for comparison.

Although our implementation is optimized for the current experimental setup, the architectural and hardware limitations such as GPGPU memory, arithmetic and CPU-GPU communication latencies and available register counts limits will definitely ease up over time. There seems to be a lack of thrust-worthy software libraries for a framework such as this on multicore CPU's, however new draft of C++11 concurrency definitely addresses this issue with the promised fine-grained nonblocking access. We think that our novel idea of parallelized runtime verification definitely is more scalable than the state-of-the-art race detection tools, among other runtime verification tools. There exists three potential bottlenecks: (i) Bookkeeping of records to the buffer, (ii) Difference between the throughputs of two concurrent runtimes (CPU and GPU) could result a type of starvation for the record threads since the buffer is statically allocated, and (iii) throughput of the Cuda kernels a.k.a. the GPU runtime. It might be hard to digest that these three bottlenecks are orthogonal to each other. During our research progress, we have encountered all of these bottlenecks until we settled down for (i) being the dominant bottleneck in all of our experiments for our benchmarks.

Future directions of this project could be reimplementing the CPU component of our framework with C++11 concurrency libraries, when it is available to further optimise the overall runtime of our framework. It is also possible for auto-tuning GPGPU runtime for available GPU's in the host computer. When today's cutting edge Cuda architecture is available for an average user, further optimization of our Cuda kernels would be possible. Increased shared memory availability for GPU threads would also enable FastTrack implementation. Increased register count would definitely further increase the throughput of any Cuda kernel without any intensive work. Adaptive frame size for checking races would be a hard task to optimize for any type of benchmark, and since production of events varies over runtime of any target process, resulting framework is unlikely to ultimately benefit from such technique since there are no known correlation between the fast produced events and distance between actual data races. Binary instrumentation could flag frames as being "hot" regions for increasing frame size; such hot regions are heuristically detectable via separate concurrent analysis of the target application threads, e.g. frequency of context switches. A multi-GPU implementation could be done with ease, this would theoretically enable us to double the frame size. A better encoding of trace of events would be beneficial for significantly reducing the memory cost, however such method should keep in mind the available GPU arithmetic instructions and aligned types.

Our work has the benefit of the both worlds of dynamic race detection tools, it checks for races on-the-fly however the analysis threads run on another computing unit which makes it comparable to post-mortem race checkers. Another benefit of our work is a try to reproduce a commercially available race detection tool such as Intel's ThreadChecker, KUDA has a byte-level resolution and has always-on feature both of which are a very harsh specification for any runtime tool. A secondary benefit of split framework is that we utilize the GPU especially when -most probably- it is not really utilized, this also has a side benefit of decreasing power consumption with respect to traditional race detection on CPU.

We hope that our split race checker framework would be a common future of tomorrow's state-of-the-art race checkers.

BIBLIOGRAPHY

[1] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, PADTAD '06, pages 69–78, New York, NY, USA, 2006. ACM.

[2] Ü. Can Bekar, Tayfun Elmas, Semih Okur, and Serdar Taşıran. KUDA: GPU Hızlandırılmış Ayrık Yarış Durumu Denetleyici (In Turkish). 3. Ulusal Yüksek Başarımlı Hesaplama Konferansı, BASARIM, Ankara, Turkey, 2012.

[3] C. Bienia, S. Kumar, and K. Li. Parsec vs. splash-2: A quantitative comparison of two multithreaded benchmark suites on chip-multiprocessors. In *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pages 47–56, 2008.

[4] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, pages 72–81, New York, NY, USA, 2008. ACM.

[5] Joseph Devietti, Benjamin P. Wood, Karin Strauss, Luis Ceze, Dan Grossman, and Shaz Qadeer. Radish: always-on sound and complete ra detection in software

and hardware. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 201–212, Washington, DC, USA, 2012. IEEE Computer Society.

[6] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: Efficiently computing the happens-before relation using locksets. Technical report, Microsoft Research, 2006.

[7] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: efficiently computing the happens-before relation using locksets. In *Proceedings of the First combined international conference on Formal Approaches to Software Testing and Runtime Verification*, FATES'06/RV'06, pages 193–208, Berlin, Heidelberg, 2006. Springer-Verlag.

[8] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware java runtime. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 245–255, New York, NY, USA, 2007. ACM.

[9] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race-aware java runtime. *Commun. ACM*, 53(11):85–92, November 2010.

[10] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX*

conference on Operating systems design and implementation, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[11] Cormac Flanagan and Stephen N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 219–232, New York, NY, USA, 2000. ACM.

[12] Cormac Flanagan and Stephen N. Freund. Fasttrack: efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[13] Cormac Flanagan and Stephen N. Freund. The roadrunner dynamic analysis framework for concurrent programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '10, pages 1–8, New York, NY, USA, 2010. ACM.

[14] RobertW. Floyd. Assigning meanings to programs. In Timothy R. Colburn, James H. Fetzer, and Terry L. Rankin, editors, *Program Verification*, volume 14 of *Studies in Cognitive Systems*, pages 65–81. Springer Netherlands, 1993.

[15] E. W. Giering, Frank Mueller, and T. P. Baker. Implementing ada 9x features using posix threads: design issues. In *Proceedings of the conference on TRI-Ada '93*, TRI-Ada '93, pages 214–228, New York, NY, USA, 1993. ACM.

[16] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: whitebox fuzzing for security testing. *Commun. ACM*, 55(3):40–44, March 2012.

[17] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[18] Guoliang Jin, Linhai Song, Wei Zhang, Shan Lu, and Ben Liblit. Automated atomicity-violation fixing. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 389–400, New York, NY, USA, 2011. ACM.

[19] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[20] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess program. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[21] 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 *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[22] Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanović, Anthony D. Joseph, and John Kubiatowicz. Gpus as an opportunity for offloading garbage

collection. In *Proceedings of the 2012 international symposium on Memory Management*, ISMM '12, pages 25–36, New York, NY, USA, 2012. ACM.

[23] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 378–391, New York, NY, USA, 2005. ACM.

[24] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[25] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[26] Robert H. B. Netzer and Barton P. Miller. Improving the accuracy of data race detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '91, pages 133–144, New York, NY, USA, 1991. ACM.

[27] Robert H. B. Netzer and Barton P. Miller. What are race conditions?: Some

issues and formalizations. *ACM Lett. Program. Lang. Syst.*, 1(1):74–88, March 1992.

[28] Edmund B. Nightingale, Daniel Peek, Peter M. Chen, and Jason Flinn. Parallelizing security checks on commodity hardware. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, pages 308–318, New York, NY, USA, 2008. ACM.

[29] NVIDIA Corporation. *NVIDIA CUDA Programming Guide v4.0.* NVIDIA Corporation, 2011.

[30] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM.

[31] Soyeon Park, Shan Lu, and Yuanyuan Zhou. Ctrigger: exposing atomicity violation bugs from their hiding places. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 25–36, New York, NY, USA, 2009. ACM.

[32] Eli Pozniansky and Assaf Schuster. Multirace: efficient on-the-fly data race detection in multithreaded c++ programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):327–340, March 2007.

[33] Paul Sack, Brian E. Bliss, Zhiqiang Ma, Paul Petersen, and Josep Torrellas. Accurate and efficient filtering for the intel thread checker race detector. In *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, pages 34–41, New York, NY, USA, 2006. ACM.

[34] 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, November 1997.

[35] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM.

[36] Konstantin Serebryany, Alexander Potapenko, Timur Iskhodzhanov, and Dmitriy Vyukov. Dynamic race detection with llvm compiler. In *Proceedings of the Second international conference on Runtime verification*, RV'11, pages 110–114, Berlin, Heidelberg, 2012. Springer-Verlag.

[37] Tianwei Sheng, Neil Vachharajani, Stephane Eranian, Robert Hundt, Wenguang Chen, and Weimin Zheng. Racez: a lightweight and non-invasive race detection tool for production applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 401–410, New York, NY, USA, 2011. ACM.

[38] Gregory T. Sullivan, Derek L. Bruening, Iris Baron, Timothy Garnett, and

Saman Amarasinghe. Dynamic native optimization of interpreters. pages 50–57, 2003.

[39] Richard N. Taylor. Assertions in programming languages. *SIGPLAN Not.*, 15(1):105–114, January 1980.

[40] Richard N. Taylor. Analysis of concurrent software by cooperative application of static and dynamic techniques. In *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, pages 127–137, New York, NY, USA, 1984. Elsevier North-Holland, Inc.

[41] Mohit Tiwari, Shashidhar Mysore, and Timothy Sherwood. Quantifying the potential of program analysis peripherals. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 53–63, Washington, DC, USA, 2009. IEEE Computer Society.

[42] Evangelos Vlachos, Michelle L. Goodstein, Michael A. Kozuch, Shimin Chen, Babak Falsafi, Phillip B. Gibbons, and Todd C. Mowry. Paralog: enabling and accelerating online parallel monitoring of multithreaded applications. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 271–284, New York, NY, USA, 2010. ACM.

[43] V. Volkov. Better performance at lower occupancy. In *GPU Technology Conference*, 2010.

[44] Christoph von Praun and Thomas R. Gross. Object race detection. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 70–82, New York, NY, USA, 2001. ACM.

[45] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, ISCA '95, pages 24–36, New York, NY, USA, 1995. ACM.

[46] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 221–234, New York, NY, USA, 2005. ACM.

[47] Wei Zhang, Chong Sun, and Shan Lu. Conmem: detecting severe concurrency bugs through an effect-oriented approach. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 179–192, New York, NY, USA, 2010. ACM.

[48] Pin Zhou, Radu Teodorescu, and Yuanyuan Zhou. Hard: Hardware-assisted lockset-based race detection. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 121–132, Washington, DC, USA, 2007. IEEE Computer Society.

VITA

Ümit Can Bekar was born in Adapazarı (Sakarya), Turkey on August 18, 1988. He received his B.Sc. degree in Electronics Engineering from Sabancı University, Istanbul, in 2009. He worked as a full-time embedded systems engineer in Techneon Mutsis while continuing his M.Sc. degree in Electronics Engineering at Boğaziçi University from 2009 to 2010. From February 2011 to June 2013, he worked as a teaching and research assistant at Koç University Research Center for Multicore Software Engineering. Microsoft Research supported his KUDA project. He has published his paper to a national conference, an international workshop and a national workshop. He co-founded the ACM student chapter during his studies at Koç University. He served as a committee member to third Computer Science Student Workshop CSW'12 and workshop chair to the CSW'13. His personal website is available at `www.canbekar.com`.