KUDA: The Split Race Checker

U. Can Bekar UCBEKAR@KU.EDU.TR
Computer Science and Engineering, Koc University, Istanbul, Turkey

Tayfun Elmas ELMAS@EECS.BERKELEY.EDU
Electrical Engineering and Computer Science, University of California, Berkeley, USA

Semih Okur OKUR2@ILLINOIS.EDU
Computer Engineering, University of Illinois at Urbana Champaign, Urbana, USA

Serdar Tasiran STASIRAN@KU.EDU.TR
Computer Science and Engineering, Koc University, Istanbul, Turkey

Abstract

Numerous runtime analysis tools are designed to observe the concurrency bugs in parallel programs (i.e. heisenbugs), there exists commercial or open source types available. Whereas usability of these tools is hurt by their performance, slowdowns may reach several thousand times the application only run. Overhead of these tools can be divided to two tightly coupled jobs: tracing and analyzing. Traditionally, on-the-fly runtime verification algorithms have been designed to run on the same processing unit as the code being monitored and both costs contribute to the slowdown of the program being monitored. We propose a novel approach for runtime analysis framework, on commodity computers with the help of a GPU. Our approach allows us to carry out analysis work on separate, dedicated processing unit without any additional or custom hardware, in parallel. Simply put, we allow the parallel runtime analysis to run concurrently in another runtime. As a demonstration of concept, we investigate on detecting concurrency bugs, in particular, data race detection; whereas one can potentially use this framework to carry out other types of analysis like specification violations, containment and even recovery from errors. We only use an additional CPU thread, a fixed size 16 MB communication buffer and a CUDA-enabled GPU. Our experiments on the performance of the split race checker framework shows that it is on average 5 times faster than traditional race checking.

1. Introduction

We propose an approach to make use of some of the computation cores and other hardware resources in a computer to monitor the programs running on other cores for concurrency errors, to contain and/or recover from these errors, if not immediately, shortly after they take place. While exploring such an approach, we had two goals: (i) to have minimal, tolerable impact on the threads being monitored, and (ii) to have the monitoring algorithms work at the same speed as the program, while possibly lagging behind by a bounded amount. The rationale behind the first goal is to enable efficient, even post-deployment use of the monitoring and bug-detection algorithms for safety-critical systems. The rationale behind the second goal is to make it possible to contain concurrency errors, notify the threads that have experienced the errors, and gracefully shut down the program or to recover from the error. One result of the second goal is to force the monitoring framework to parallelize the event logging and analysis algorithms as much as possible.

In our approach, the application being monitored and the actual monitoring code run on separate processing units/resources. They communicate with each other using some shared memory or message passing. The instrumented application code only has the additional responsibility of communicating relevant events to the monitoring code. The monitoring and runtime analysis code can be quite complex, but runs on separate processors and is parallelized, thus, the application performance is not affected by the runtime analyses being performed. We conjecture that the performance penalty on the application being monitored due to instrumentation and communication of relevant events can be reduced to negligible levels, for example, using inexpensive hardware support such as hardware-assisted message passing (Francesco et al., 2005). The goal is for the monitoring code to run at least the same speed as the application being monitored, but lag behind by a very small delay due to event communication. (In our experiments this delay was in milliseconds.) This makes possible scenarios in which, in response to errors detected, the application has shut down gracefully, or a previous valid checkpoint is restored or the application is restarted.

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

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

For demonstration, we adapted the well known ERASER (Savage et al., 1997) and GOLDILOCKS algorithms (Elmas et al., 2007) for data race checking, so that they can be parallelized to run on a large number of threads and cores on the GPU. Surprisingly, this high parallelism has a simplifying effect on the algorithm implementation. Since we have many threads/cores, the algorithm can be written to make each thread or core to perform a local and independent check (for a single memory access) without having to worry about sharing or interaction with other threads. Each thread creates only the necessary data structures for the check and discards/reuses them after the check completes; this avoids the need for memory management and sharing of complicated algorithm-specific data structures.

We implemented our proposed system in a tool called KUDA. KUDA is open source and available at http://kuda.codeplex.com. We use the Pin (Luk et al., 2005) library to instrument binaries for monitoring and the CUDA (NVIDIA Corporation, 2011) library to run our analysis algorithms on the GPU. We applied KUDA to a number of multithreaded programs from the PARSEC and SPLASH-2 benchmark suites. We performed experiments using CPU and GPU implementations of the ERASER and GOLDILOCKS algorithms. We chose ERASER to represent a cheap (although imprecise) algorithm, while GOLDILOCKS served as a representative precise, higher-complexity algorithm. We contrasted two approaches: (i) a straightforward implementation of ERASER running on the same threads and cores as the application, and (ii) an implementation of GOLDILOCKS using our framework, where the checking threads are decoupled and run on the GPU. Overall, our early experimental results indicate that our approach is promising. Using a cheaper race detection algorithm using the traditional approach as exemplified by (i) causes about twenty times more slowdown compared to a more complex race-detection algorithm implemented in our approach!

2. Our approach I: Overall system

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

2.1 Observing the execution trace

Our technique is based on logging the execution as a linear sequence of events and running the analysis in a very efficient way. In order to enable efficient handling of the event log, we 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 event frame a *unit of input* for the analysis implemented in the GPU. Each frame is checked independently from other frames and minimal information is kept between frames, e.g., racy variables to omit accesses to those variables. When a frame is completely checked, the events in it is discarded and it is reused to store later events.

While splitting the execution into independent frames may cause unsound results due to pairs of events from separate frames, our framework allows to adjust the precision to increase the chance of finding bugs. We have chosen to defer the soundness issue, since the goal of this study was to show the feasibility of highly parallel, at-speed runtime verification. Empirical evidence by other researchers indicates that this is a minor source of unsoundness: Many concurrency errors involve a small number of threads, and can be detected by focusing on a short portion of the execution (Musuvathi et al., 2008).

3. Our approach II: Checking frames on the GPU

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

3.1 Background on GPU computing using CUDA

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

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

1. The worker thread first copies the contents of the event frame to the pre-allocated region on the GPU device memory.
2. It calls the kernel function of an available checker algorithm. That kernel function is executed by the GPU cores in parallel and asynchronously with the CPU. When calling the kernel function, it passes as arguments the pointer to device memory region storing the current frame as well as additional values, such as the number of events in the frame. Each kernel is executed in a SIMD (single instruction, multiple data) style on multiple cores and threads.
3. 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.

Given the challenges in writing kernels, we wrote parallel kernels for the ERASER and GOLDILOCKS algorithms. 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. The implementations of the kernels is also available at http://kuda.codeplex.com.

4. Experimental evaluation

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

1. A dynamic library containing the core functionality including the routines for recording events, managing event frames, and running the race detection kernels on the GPU. We use the CUDA 4.0 library (NVIDIA Corporation, 2011) 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 (Luk et al., 2005) 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 multi-threaded programs written using the pthreads library (for thread creation and join, and synchronization primitives including mutex and readers/writer locks).

4.1 Benchmarks

We applied our tool KUDA on a collection of multi-threaded programs from PARSEC (Bienia et al., 2008) and SPLASH-2 (Woo et al., 1995) benchmark suites. Due to space restrictions our experiment results table is not presented in this paper. But note that, in a typical execution, our benchmarks generate a few hundreds of millions of events and hundreds of thousands of frames, each of which is checked on the GPU.

4.2 Results

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$^+$ (Pozniansky & Schuster, 2007) and FASTTRACK (Flanagan & Freund, 2009) (available in our code base).

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.

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 ($\mathrm{bodytrack}$) and 14.7 (fmm).

References

Bienia, C., Kumar, S., Singh, J. P., & Li, K. (2008). *The parsec benchmark suite: Characterization and architectural implications* (Technical Report TR-811-08). Princeton University.

Elmas, T., Qadeer, S., & Tasiran, S. (2007). Goldilocks: a race and transaction-aware java runtime. *PLDI* (pp. 245–255). San Diego, California, USA.

Flanagan, C., & Freund, S. N. (2009). Fasttrack: efficient and precise dynamic race detection. *PLDI* (pp. 121–133). Dublin, Ireland.

Francesco, P., Antonio, P., & Marchal, P. (2005). Flexible hardware/software support for message passing on a distributed shared memory architecture. *DATE* (pp. 736–741). Washington, DC, USA.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., & Hazelwood, K. (2005). Pin: building customized program analysis tools with dynamic instrumentation. *PLDI* (pp. 190–200). Chicago, IL, USA.

Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., & Neamtiu, I. (2008). Finding and reproducing heisenbugs in concurrent programs. *OSDI* (pp. 267–280). San Diego, California.

NVIDIA Corporation (2011). *Nvidia cuda programming guide v4.0*. NVIDIA Corporation.

Pozniansky, E., & Schuster, A. (2007). Multirace: Efficient on-the-fly data race detection in multithreaded c++ programs. *Concurr. Comput. : Pract. Exper.*, *19*, 327–340.

Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., & Anderson, T. (1997). Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, *15*, 391–411.

Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., & Gupta, A. (1995). The splash-2 programs: characterization and methodological considerations. *ISCA* (pp. 24–36). S. Margherita Ligure, Italy.