Position Paper: Software-based Techniques for Reducing the Vulnerability of GPU Applications

[1]Si Li, [2]Vilas Sridharan, [3]Sudhanva Gurumurthi, [1]Sudhakar Yalamanchili
[1]Computer Architecture Systems Laboratory, Georgia Institute of Technology
[2]RAS Architecture, Advanced Micro Devices, Inc.
[3]AMD Research, Advanced Micro Devices, Inc.

Abstract

As highly-parallel accelerators such as graphics processing units become more important in high-performance computing, so does the need to ensure their reliable operation. In response, research has been directed at several efforts to characterize and understand the hardware vulnerability of GPU microarchitecture structures, as well as to detectng and correcting such vulnerabilities. In this position paper, we advocate a transparent, customizable, and dynamic software approach to reduce the vulnerability of GPU applications. Specifically, we propose to realize these enhancements via transparent code patching of GPU applications to increase protection of critical program regions. We outline our approach, current infrastructure, some early results, and long-term plans.

1 Introduction

Fault-detection and recovery mechanisms are integral to computing devices deployed in environments that demand high reliability (e.g., supercomputers). As GPUs increasingly are used for general-purpose computing, there is a need to incorporate reliability features to meet customer expectations. Some GPU accelerators already implement reliability mechanisms (e.g., redundancy in the memory system and bus operation). Error-correction code (ECC) enabled memory corrects transient faults in DRAM, while cyclic redundancy checks in the GDDR5 interface prevent faults from occurring during transfers across the memory bus[12]. GPUs also employ ECC to protect large on-chip memory structures [11]. However, not all devices employ these techniques and such techniques do not cover all failure models (e.g., untested corner cases or transient faults in the compute logic). Hardware-based protection mechanisms also can impose significant area, performance, and power overheads.

Our research pursues a software-based approach to providing resilience for applications running on GPUs. Software-based approaches can facilitate end-to-end resilience that allows fault detection and recovery to be tailored to the application rather than following a “one size fits all” approach that is designed for the worst case. Flexible software approaches can reduce the overall cost of implementing reliability, by reducing die area dedicated solely to reliability. Such approaches also can boost performance and energy efficiency by incorporating the right level of protection for software to achieve resilience. Therefore, we seek an approach that is customizable, extensible, and transparent to the application program to augment hardware techniques. Previous approaches [16, 17, 14, 20] demonstrated the increased reliability of software-based fault-detection techniques in CPU architectures. The growing predominance of GPUs in compute-heavy infrastructures motivates the need to pursue a similar research direction.

The specific work discussed in this paper relies on the well-known architecture vulnerability factor (AVF) concept [1] and the more recent related concept of program vulnerability factor (PVF) [19]. PVF is a component of AVF that can be controlled by program transformations including the insertion of error-detection and recovery code. We refer to such approaches that mitigate vulnerability as *Software Reliability Enhancements (SRE)* [10]. Such approaches introduce error-checking mechanisms such as checksums, redundant computations, and redundant memory operations. They are inserted into the application binary during just-in-time (JIT) compilation. Each SRE has a specific coverage and associated performance overhead. The actual decision to use SRE and the choice of specific technique (e.g., checksum), is made by a reliability-aware runtime manager responsible for launching the applications.

Our approach is to estimate the vulnerability of regions of an application kernel to transient errors to determine spe-

Presented at the Dependable GPU Computing workshop of ACM/IEEE DATE 2014 conference, Dresden, Germany, March 28, 2014

cific regions of a GPU kernel where the insertion of an appropriate SRE will be most beneficial while minimizing performance overhead. This involves determining the degree of vulnerability in a region of code as well as the type of vulnerability, to choose the most beneficial SRE. In turn, this enables a figurative "knob" to dial between performance and reliability. This knob can be controlled by the higher-level runtime manager that determines the level of acceptable fault vulnerability relative to performance impact.

2 Rationale for Software-based Approaches

As with the CPU industry, we see the emergence of multiple GPU platforms and consequently instruction set architectures (ISAs). Our goal is to develop a set of SRE techniques that are based on the bulk synchronous parallel execution model and easily portable across multiple platforms. Accordingly we focus on code injection at the low-level ISA such as NVIDIA's Parallel Thread Execution (PTX) or the Heterogeneous System Architecture Intermediate Language (HSAIL) ISA implemented on AMD GPUs. Thus, while SRE techniques can be customized for a target GPU, our approach can provide a repertoire of target-neutral SRE program transformations.

Further, we envision developing and cataloging SRE techniques that target different fault models or different classes of effects of faults. For example, in earlier work [10] we reported on symptom-based SRE techniques for an alignment checker that targets errors in memory address operands in load/store instructions. These errors can be the result of transient bit-flips or algorithmic or logic errors. We also reported a control-flow checker that targets faults in the branch address operand of control-flow instructions and detects illegal control flows. We also want to introduce vulnerability-based SRE techniques that provide end-to-end protection as opposed to symptom or structure-based error detection in hardware-based techniques such as ECC; these solutions are limited to strict protection-domains.

Finally, we want to introduce techniques to estimate program vulnerability to maximize reliability-coverage while minimizing the cost of SRE overhead. We believe this can be acheived by using information that is available statically or at JIT time such as PVF, control-flow, and key GPU-specific variables. Ultimately, the long-term solution is to enable selective, transparent, customizable code injection to improve reliable operation of applications on GPUs. While algorithm-based fault tolerance (ABFT) techniques also can provide software-based fault tolerance for GPUs [8], such techniques target specific algorithms. Our goal is to develop more generic software approaches that provide GPU reliability. However, an interesting area of study would be a performance comparison of SRE and ABFT techniques. In fact, one question is whether SRE be used to implement ABFT techniques, thereby leveraging the advantages of both.

3 Code Injection Mechanism

Lynx [7] is a dynamic instrumentation engine for data-parallel applications on GPU architectures. Specifically, Lynx allows the creation of customized, user-defined instrumentation routines that can be applied transparently at run-time for a variety of purposes, including performance debugging and correctness checking. Lynx originally was written as part of GPU Ocelot [3], a dynamic compilation framework for executing CUDA$^{\text{TM}}$ applications on multiple backends. CUDA kernels were compiled to NVIDIA's PTX ISA and then parsed and stored in Ocelot's intermediate representation (IR) format. The Ocelot pass manager applied analysis and transformation passes over the kernel to insert instrumentation code. Lynx now exists as a stand-alone library and provides both an API to integrate with any runtime as well as a default implementation of the CUDA runtime to support the execution of CUDA applications directly. The general procedure for executing CUDA applications includes specifying the desired instrumentation (or in our case SRE code) in a C-like syntax, lowering to PTX, applying the relevant PTX transformations to inject this code, and emitting the patched kernel for execution on the device. For more information on the transformation pass framework, please refer to [6].

Our current infrastructure is being extended to support other low-level virtual ISAs such as HSAIL [9]. This is being achieved by generalizing the Ocelot IR to serve as a neutral target supporting multiple vendor IRs, thereby leveraging the significant investments so far to support a repertoire of target-neutral SRE program transformations. Advances that are more recent have combined symbolic execution with the use of AMD's intermediate language (IL) to reduce the overheads of instrumentation and extend the scope of SRE [13].

4 A Motivating Example

To locate code regions of high vulnerability, we currently focus our attention on the liveness properties of variables in the IR. The intuition is that the greater the live range of a variable, the more vulnerable it is to transient errors.

We can currently execute kernels with Ocelot's PTX emulator and study the live ranges of individual values, which correspond to unique virtual registers in a one-to-one mapping. Live ranges are measured in the number of PTX instructions between the initial production and final consumption of a value. We study the live range behavior by considering several metrics. For example, we sum the live ranges

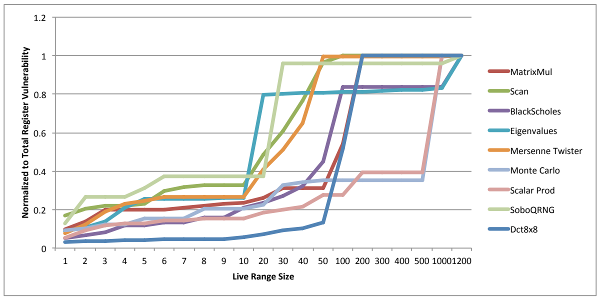

Figure 1. Cumulative Histogram of IR-level Live Range Distribution

of all variables in a thread that provides the thread block live range. Extending the sum across all the threads in a thread block provides the thread block live range. Further, computing the sum of thread block live ranges across a kernel provides the kernel live range. We can now express error coverage as a percentage of the preceding sums that are protected by the insertion of error-detection code. We also can construct live range histograms in which a bin corresponds to live range values and the count corresponds to the number of variables in all of the kernels in that application whose live range lies in the bin range. We can express error coverage and overheads as a percentage of the number of bins protected by error-detection code. Some examples are discussed in the remainder of this section.

The cumulative histogram of the live ranges across a set of benchmarks is illustrated in Figure 1. Total register vulnerability is represented by the sum of kernel live ranges across all kernels in an application. The Y-value of each live range bin captures the fraction of the total live range whose size is equal to or smaller than the live range size of the bin. The figure expresses this value as a percentage of the register vulnerability. Let V be the total number of live range instances in an application (across all kernels), and let s be the number of live range bins, the histogram then is h_i and the cumulative histogram is H_i:

$$V = \sum_{i=1}^{s} h_i \qquad H_i = \sum_{j=1}^{i} h_j$$

For example, only 13% of the total register vulnerability belongs to values whose live ranges are less than 50 instructions. This histogram captures a coarse measure of the coverage possible when implementing an error check across a subset of values based on live range size. The overhead to detect a transient bit-flip in a live range is a fixed constant of two instructions. This overhead is amortized over the duration of the live range. The larger the live range, the smaller the relative cost. From Figure 1, it is apparent that the majority of live ranges are relativelty large (>10). Thus, the overall PVF can be improved at a modest cost. Our current infrastructure can insert these checks for all variables with a specific live range and execute the modified kernel on the device. While the software engineering is demonstrable, the interesting research questions of quantifying coverage and developing control schemes for fine-grained orchestration of performance-reliability tradeoffs are in progress. The results are encouraging in establishing the opportunity to assess and manage program vulnerability to transient errors at a microarchitecture-neutral intermediate program level.

Measuring PVF over the intermediate representation has pitfalls as well as advantages. The IR representation is hardware-neutral. However, when mapped to a specific device with a specific number of registers the actual PVF will be different. We argue that the IR still provides useful information and guidance especially if there is consistent relationship between PVF computed over the IR and that computed over the target binary (this currently is a open question). Register allocation may split the live range (generate spill code) but the checks will still be valid although now any errors detected in the memory system can lead to failures. Further, we see that PVF is sensitive to the thread block scheduler because scheduling decisions can change the period of (real) time over which register values are live. Consequently they become more vulnerable to transient errors and can degrade PVF. Understanding and mitigating these effects are among the research challenges we are addressing.

5 Related Work

Yau, et al. [20] first demonstrated self-checking software over function calls, control sequence, and data integrity. Annelid [14] records the ranges of allocated memory and performs bounds-check on each memory access. Annelid operates on top of Valgrind [15], a dynamic instrumentation platform for the x86 ISA. Erez et al. [5] proposed a fault-tolerance technique using redundant execution, checkpoints at control flows that govern write-backs, and control-flow checking only at checkpoints. This technique focuses on the Merrimac architecture. Sheaffer et al. [18] proposed redundant hardware execution resources to provide resilience in the face of transient faults in computational logic. Dimitrov et al. [4] proposed three methodologies of redundant execution to achieve software reliability in GPU applications with approximately 100% overhead: duplicate kernel execution, instruction-level redundancy, and thread-level redundancy. Maruyama et al. [12] showed a low-overhead software-based ECC for GPU applications by offloading parity computation to the CPU in the form of a library. This required manual modification of the target application, whereas reliability enhancements using Lynx can be performed transparently at runtime. Chung et al. [2] proposed the concept of containment domains (CD) for resilient computing in a scalable and efficient manner. CDs are hierarchical in nature and failures in nested CD are localized and do not propagate outward. Reliability enhancements proposed in this paper can be used to detect failures when a more efficient algorithmic failure detection does not exist.

6 Future Work

In general the notion of improving program vulnerability using software reliability enhancements as described in this work has multiple possible research directions. From an engineering perspective, our vision is to extend our software infrastructure to adopt a general intermediate representation so that the open-source infrastructure can be leveraged across GPU platforms; HSAIL is a nearer-term target. From a research perspective we wish to explore multiple SRE techniques and draw initial inspiration from the body of existing work on software-based fault tolerance to improve the program and architectural vulnerability factors. The SRE techniques then could become a library to deploy selectively by a JIT compilation infrastructure to adapt to temporally varying vulnerability demands.

References

[1] A. Biswas and P. Racunas. Computing architectural vulnerability factors for address-based structures. *Proceedings of the 32nd International Symposium on Computer Architecture*, pages 532–543, 2005.

[2] J. Chung, I. Lee, M. Sullivan, and J. Ryoo. Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems. *SC*, 2012.

[3] G. Diamos and A. Kerr. Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems. *Proceedings of the 19th international conference on Parallel architectures and compilation techniques - PACT '10*, 2010.

[4] M. Dimitrov, M. Mantor, and H. Zhou. Understanding software approaches for GPGPU reliability. *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-2*, pages 94–104, 2009.

[5] M. Erez, N. Jayasena, T. Knight, and W. Dally. Fault Tolerance Techniques for the Merrimac Streaming Supercomputer. *ACM/IEEE SC 2005 Conference (SC'05)*, (c):29–29, 2005.

[6] N. Farooqui, A. Kerr, G. Diamos, S. Yalamanchili, and K. Schwan. A framework for dynamically instrumenting GPU compute applications within GPU Ocelot. *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units - GPGPU-4*, page 1, 2011.

[7] N. Farooqui, A. Kerr, G. Eisenhauer, K. Schwan, and S. Yalamanchili. Lynx: A dynamic instrumentation system for data-parallel applications on GPGPU architectures. *2012 IEEE International Symposium on Performance Analysis of Systems & Software*, pages 58–67, Apr. 2012.

[8] S. H. H-J. Wunderlich, C. Braun. Efficacy and efficiency of algorithm-based fault-tolerance on GPUs. In *On-Line Testing Symposium (IOLTS*, July 2013.

[9] G. Kyriazis. Heterogeneous system architecture: A technical review. In *AMD*, October 2012.

[10] S. Li, N. Farooqui, , and S. Yalamanchili. Software reliability enhancements for gpu applications. In *Proceedings of the Sixth Workshop on Programmability Issues for Heterogeneous Multicores*, 2013.

[11] M. Mantor. AMD Radeon$^{\text{TM}}$ HD 7970 with Graphics Core Next (GCN) Architecture. In *HotChips*, August 2012.

[12] N. Maruyama. A high-performance fault-tolerant software framework for memory on commodity gpus. *Parallel & Distributed Processing (IPDPS)*, pages 1–12, 2010.

[13] K. S. N. Farooqui and S. Yalamanchili. Efficient Instrumentation of GPGPU Programs using Information Flow Analysis and Symbolic Execution. In *Proceedings of Seventh Workshop on General-Purpose Computation on Graphics Processing Units (GPGPU-7)*, March 2014.

[14] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. *SPACE*, 2004.

[15] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, pages 89–100, 2007.

[16] N. Oh, P. P. Shirvani, E. J. Mccluskey, and L. Fellow. Control-Flow Checking by Software Signatures. 51(2):111–122, 2002.

[17] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software Implemented Fault Tolerance. *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.

[18] J. Sheaffer, D. Luebke, and K. Skadron. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 55–64, 2007.

[19] V. Sridharan and D. R. Kaeli. Eliminating microarchitectural dependency from Architectural Vulnerability. *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 117–128, Feb. 2009.

[20] S. Yau and R. Cheung. Design of self-checking software. *ACM SIGPLAN Notices*, 10(6):450–455, 1975.