Software-based Dynamic Reliability Management for GPU Applications

Si Li*, Vilas Sridharan†, Sudhanva Gurumurthi‡ and Sudhakar Yalamanchili*
*Computer Architecture Systems Laboratory, Georgia Institute of Technology
†RAS Architecture, Advanced Micro Devices, Inc
‡AMD Research, Advanced Micro Devices, Inc

Abstract—**In this paper we advocate a framework for dynamic reliability management (DRM) for GPU applications based on the idea of plug-n-play software-based reliability enhancement (SRE). The approach entails first assessing the vulnerability of GPU kernels to soft errors in program visible structures. This assessment is performed on a low level intermediate program representation rather than the application source. Second, this assessment guides selective injection of code implementing SRE techniques to protect the most vulnerable data. Code injection occurs transparently at runtime using a just-in-time (JIT) compiler. Thus, reliability enhancement is *selective*, *transparent*, *on-demand*, and *customizable*. We argue this flexible, automated software-based DRM framework can provide an important, cost-effective approach to scaling reliability of large systems. We present the results of a proof of concept implementation on NVIDIA GPUs demonstrating the ability to traverse a range of performance reliability tradeoffs.**

I. Introduction

This paper is concerned with the challenges facing design of reliable high performance computing systems. Transistors in future technology nodes will become more vulnerable to transient bit-flips while the number of cores and devices continues to grow [1, 2]. The result is an increasingly large cross section of data on-chip that are subject to transient errors. The cost of conventional techniques for reliability improvement such as checkpoint and roll back scales with the growth of systems by consuming an increasing fraction of the available compute time [3]. There also exists a rich repertoire of application-neutral error detection and recovery techniques that can be performed in hardware or software [4–7]. However, approaches baked into the hardware, software stack, or algorithm pay a fixed performance overhead independent of the pattern of failures or characteristics of the applications that might otherwise be exploited to improve coverage and/or overhead [8, 9]. We believe it is important to understand and control the trade offs between reliability and performance impact.

Modern and future large systems will be heterogeneous in their architecture and technology [10]. This suggests the 'one size fits all' approaches to dynamic reliability management (DRM) will be limited at best while flexibility in adapting to architectures and applications is desirable. We are interested in enhancing the reliability of GPUs for data intensive applications. Modern languages such as CUDA and OpenCL has made these accelerators accessible to a wider range of data intensive applications beyond graphics but they also present challenging resilience problems.

In this paper we advocate for an approach to dynamic reliability management (DRM) using software-based reliability enhancements (SRE) for GPU applications. We present a framework for managing the vulnerability of programs (GPU kernels) to soft errors in program visible structures. We consider errors in both sequential elements (register files) and combinational elements (ALUs). The central idea is to dynamically assess program vulnerability at a low level virtual instruction set architecture (ISA) rather than the application source followed by a selection of SRE techniques to protect the most vulnerable program structures. A just-in-time (JIT) compilation environment then creates an executable of the original code augmented with the software implementations of the selected SRE techniques. We envision this framework employed offline to construct robust applications, or online where instrumentation can guide selective, transparent code injection across program phases. This paper presents a proof of concept of the major elements of this framework.

We believe there is a major benefit to a DRM runtime that can choose from a repertoire of SRE techniques based on their coverage and performance overhead characteristics. Since the analysis and code injecton is performed at a low level ISA using a JIT compiler, reliability enhancement is *selective*, *transparent*, *on-demand*, and *customizable*. This enables on-demand improvements in resilience without a fixed perpetual overhead and without modification of the application source. We argue that such software based DRM techniques provide an important, cost-effective approach for improving the reliability of a large class of applications.

Towards the preceding goals, this paper seeks to make the following contributions:

- An approach for the characterization of the program vulnerability of GPU compute kernels.
- A code transformation mechanism for GPUs that operates at the virtual ISA level.
- A DRM framework to analyze and improve the vulnerability of GPU kernels. In particular, this approach encompasses the ability to trade performance overhead for reduced vulnerability and potentially increase reliability in a cost-effective manner.
- A proof of concept implementation of this framework with code injection of error detection techniques to illus-

trate the characteristics of the framework.

This paper describes the computation of program vulnerability for GPU kernels, the framework for selection and injection of code to improve reliability, and the results of our experimental evaluation of a prototype implementation that exercises this framework with error detection techniques.

II. Machine-Independent Reliability Analysis

With new CPU and GPU microarchitectures being produced every couple of years, the reliability characteristics of the underlying hardware changes frequently. Therefore we are motivated to seek solutions that are cost-effective and forward portable across microarchitecture generations. Moreover we note that while the hardware implementations evolve at such a pace, changes to the instruction set architecture (ISA) are much slower. For example, both AMD and NVIDIA rely on virtual ISAs that are translated at runtime to the native ISA, shielding software investments from generational machine ISA and microarchitecture specific changes. In particular, CUDA was released 7 years ago and while it has been updated often, the underlying Parallel Thread Execution (PTX) virtual ISA has mostly been extended. In contrast, the microarchitecture has seen major changes [2, 11, 12].

Further, the GPU ISAs expose more of the underlying machine state than most CPU ISAs. In general, CPU architectures mask various performance-enhancing structures such as the Register Aliasing Table and Reorder Buffer behind the ISA. GPU architectures lack such structures and devotes a large fraction of the die area to software visible structures such as registers and scratchpad memory [13]. Thus, techniques based on protecting program visible states will find greater utility in GPU architectures.

During microarchitectural design exploration, Architectural Vulnerability Factor (AVF) [14] is often used for the vulnerability assessment of microarchitectural structures (caches, reorder buffer, etc). AVF uses data-flow dependency information between instructions to determine which state bits are required for correct execution at each clock cycle. AVF is computed as the *ratio* of live state necessary for architecturally correct execution (ACE) of the program, to the full machine state. This ratio is averaged over all cycles of program execution.

Program vulnerability factor (PVF) [15] measures the fraction of *program visible* architectural resources that are required for architecturally correct execution (ACE). A resource is any architecturally-visible structure or operation, e.g., register file or a floating-point operation.

Equation 1 taken from [15] defines the PVF of an architectural structure R as the fraction of time that a bit is ACE. An architecturally-visible definition of time is in terms of dynamic instructions *I*. We can calculate the program vulnerability of an architectural resource as follows.

$$PVF_R = \frac{\sum_{i=0}^{I} ACE_{iR}}{B_R \times I} \tag{1}$$

Where B_R is the total number of bits in *R* and ACE_{iR} is the subset of bits that are ACE at instruction *i*.

To capture errors introduced in combinational logic such as the ALU, we extend the concept of ACE to instructions, where an instruction is ACE if the bits it produces are ACE. Equation 2 defines the PVF for an instruction type *T*, such as integer or floating point operation, as the fraction of dynamic instructions *I* of that type that are ACE. This yields the *ratio* of instructions that are vulnerable to transient faults.

$$PVF_T = \frac{\sum_{i=0}^{I} ACE_{iT}}{\sum_{i=0}^{I} inst_{iT}} \tag{2}$$

Where ACE_{iT} = 1 if instruction i is of type T and is ACE, otherwise 0 and $inst_{iT}$ = 1 if instruction i is of type T, otherwise 0.

Sridharan et al. showed a correlation between AVF and PVF [15]. We note that AVF values between microarchitecture structures are correlated [16]. Consequently we expect that the PVF computed as a function of program visible structures will behave similarly to the AVF of that device. As a result, we have a means to assess a measure of the vulnerability of the execution of a program at the (virtual) ISA level, independent of a detailed hardware implementation.

We target NVIDIA's PTX virtual ISA in our current implementation. The PTX intermediate representation is based on an infinite register set and is JIT compiled to the machine-specific ISA. Our proof of concept implementation illustrates the utility of vulnerability analysis at this level. Thus, SRE techniques are portable across families of implementations of the same ISA and to a great extent forward portable across GPU generations as long as the virtual ISA is stable.

III. Dynamic Reliability Management Framework

Our DRM vision is illustrated in Figure 1. An application executing on a host CPU launches one or more compute kernels for execution on the GPU. These kernels are processed either offline or online by the DRM (Section IV-A), where the kernels are parsed into an internal representation (IR). A series of transformation layers and analysis passes are executed over the IR. First, a data-flow analysis pass is applied to assess kernel vulnerability (Section IV-B). Second, the runtime overhead from employing each candidate SRE technique is assessed using a performance model. Third, these two analyses feed a decision model to select appropriate SRE techniques based on trade-offs between performance overhead and improvements in program vulnerability. Finally, software implementations of the selected SRE techniques are inserted via a transformation pass over the kernel IR. The kernel is now launched through the standard interface - i.e., translated by the driver (CUDA) or finalizer (HSA) - to generate native binaries for execution on the device.

Note the framework itself admits to code injection for error detection, error recovery, error masking, etc. As a proof of concept, this paper describes the injection of error detection mechanisms to protect vulnerable data structures.

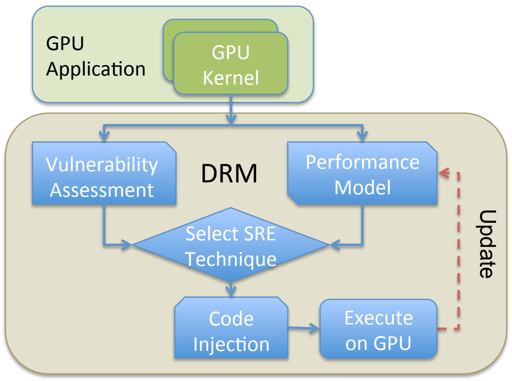

Fig. 1: Our vision of software dynamic reliability management

IV. Dynamic Reliability Management Implementation

Our implementation is based on CUDA implementing the key components shown in Figure 1. We use GPU Ocelot [17] as the code transformation and analysis infrastructure and the Lynx dynamic instrumentation engine [18] for code injection. In the examples presented in this paper, vulnerability analysis is performed offline using the Ocelot PTX emulator while kernel transformation and JIT compilation generates executables for NVIDIA GPUs.

A. Software Reliability Enhancement

In this paper we demonstrate the SRE concept using existing software error detection techniques for transient errors on vulnerable data structures. PTX implementations of error-detection techniques are injected into a kernel to protect the ACE bits of various architectural structures such as the register file, ALU, and memory unit. Due to limitations of the current GPU execution environment, errors are detected and reported to the DRM when a kernel completes execution.

The following sections describe the three error detection techniques implemented in this work.

1) Register Check: Yim et al. [19] presented a lightweight technique called Hauberk that enabled low overhead error detection of register values. The technique is used in this work to protect value live ranges. A live range spans from when a value is first created to when it is last used, measured in instructions executed. A single signature register is allocated for error-checking live ranges of all values in the kernel. For each live range, the target value is XORed with the signature register twice: once at the creation of the value and a second time after its last use. If no error has occurred between the two XORs, the signature register returns to its initial value. Since XOR is commutative, multiple live ranges can XOR with the same signature register in any order without changing the result. At the end of kernel execution, this value is stored to global memory where our DRM can access and evaluate if an error has occurred and perform recovery functions as needed.

There are several possible extensions to this technique. Multiple signature registers could be used to partition value live ranges into groups based on vulnerability. Recovery mechanism can be implemented based on this grouping. Another possible extension is to find idempotent code regions and assign each a separate signature register. This enables fine-grain error detection and seamless rollbacks. However, these extensions increase register pressure and potentially decrease thread occupancy and overall performance.

2) Instruction Check: While transient single-bit upsets typically occur on storage cells, they can also occur in combinational logic such as the ALU [20]. Instructions duplication and output comparison can detect such transient errors. Instructions are duplicated if they 1) produce an ACE value 2) are one of a load instruction, integer instruction, or floating-point instruction. The duplicate instruction uses the same source operands but writes to a new destination operand. Using the Hauberk technique the results can be XORed to a single signature register shared across all live range values just as in Register Check.

3) Control-flow Checker: As described in [21] this checker detects transient faults occurring in the branch address by verifying the legality of each control-flow redirection. This is achieved by comparing a running signature with a per basic block (BB) signature. A running signature is a register initialized with a default value at the entry block and transformed at the beginning of each BB, such that it matches with the per BB signature only if it came directly from a legal predecessor block. Since an illegal control flow cannot be guaranteed to reach the exit block, a comparison and optional store is inserted at each error-detection location. A more detailed description can be found in [4, 21]. Note, this method will not catch an errant but legal jump as a result of a corrupted condition value. However, the register check technique can be applied in conjunction to catch this case.

B. Vulnerability Assessment

In this paper, vulnerability assessment is performed as an offline PVF analysis pass using Ocelots PTX instruction set emulator [22]. By keeping track of data flow dependencies between instructions, we can determine which values (and therefore bits in a structure such as the register file) are ACE at each instruction as well as the live range of all variables. Consequently, we can also quantify the reduction in vulnerability due to the use of each error detection technique described in Section IV-A by measured improvement in PVF. This analysis can be used to select a combination of error detection techniques for injection based on acceptable performance overhead. The PVF analysis described in this paper addresses transient single-bit faults.

C. Code Injection Mechanism

GPU kernels take the form of a fat binary that includes a text-based representation of the PTX code. On kernel invocation the Ocelot runtime extracts the PTX code from the executable, imports it to the Ocelot IR, applies a transformation pass to insert the selected error detection technique using Lynx, and forwards the resulting modified IR to the driver for

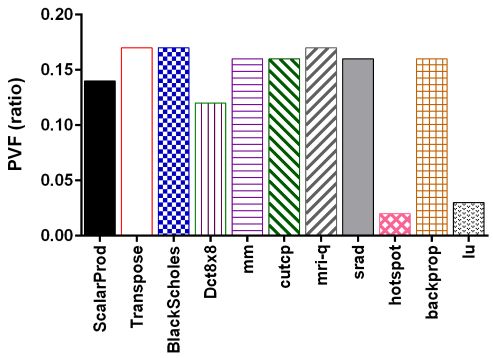

Fig. 2: PVF of register files shows variability in vulnerability between benchmarks

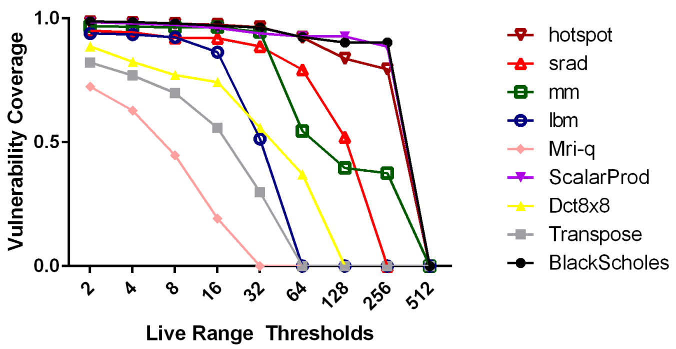

Fig. 3: Vulnerability coverage as a function of Live-Range threshold

JIT compilation. The code injection pass is executed for each error detection technique. In this paper, we present results for the application of each error detection technique in isolation.

V. Results and Discussion

We evaluated the vulnerability and SRE performance overhead of kernels selected from the NVIDIA CUDA SDK, Parboil [23], and Rodinia benchmark suites [24]. The input data set sizes were increased to ensure high utilization for proper comparison. All performance overhead assessments were performed on a commodity NVIDIA Kepler GTX680 GPU with 8 SMXs [25]. Performance counters and run-time were collected with the built-in NVIDIA CUDA Command Line Profiler [26]. Each listed run-time is averaged over several executions to remove the impact of outliers. In the following, we analyze error detection techniques over specific program visible structures. The goal is to understand, in each instance, the potential improvements in program vulnerability.

The register file can be protected from transient faults by using the register check error detector. Figure 2 illustrates the *normalized* PVF achieved with this combination. Since the PTX ISA uses an infinite register set, we normalize the PVF values to the maximum number of PTX variables that are live at any point in time. This would correspond to the total number of registers needed per SIMD lane during execution with no register spilling. This model mimics the register allocation algorithm in the driver that transforms from an infinite to a finite register set in hardware. Actual number of hardware registers will vary, but changes in PVF will reflect similar changes in the underlying AVF.

Most benchmarks hover at a similar level of PVF, indicating similar usage patterns. The lower PVF values are due long series of highly sequential code where a value is used immediately after being generated. Some benchmarks, such as hotspot and LU decomposition are much lower because they use many registers during initialization but for a short amount of time. The fraction of vulnerable registers versus allocated registers are smaller, making these kernels more resilient to transient errors.

We are also interested in the distribution of vulnerability across variables in a kernel over live range thresholds for the register check SRE. We show a disproportionate concentration of program vulnerability in a small set of variables. We compute the ratio of overall PVF (denoted as vulnerability coverage in Figure 3), that is contributed by protecting live ranges of K or more instructions. For example, for Transpose, we can cover 60% of the vulnerability when we protect variables with live ranges of size 16 instructions and greater. In fact most benchmarks show a large fraction of vulnerability occur with larger live ranges. In these cases, the error detection overhead is smaller since the fixed cost of detection overhead is amortized over a larger live range.

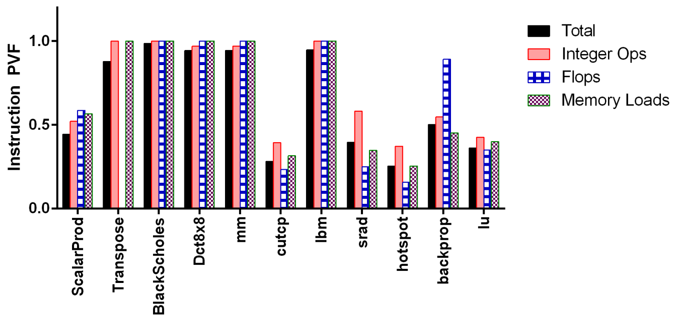

Fig. 4: PVF by instruction type shows the variability in vulnerability between benchmarks

Vulnerability in combinational logic is covered by PVF_T described in Equation 2. Figure 4 captures the diversity of control flow and its effect on PVF_T. The figure is organized by instruction type. In benchmarks such as BlackScholes and matrix multiplications, there are few loops or branches that do not contribute to the output of the program. Thus their PVFs are close to 1. However in benchmarks such as srad and hotspot exhibit high branching factor, as a result all instruction types exhibit lower PVF. Code inspection reveals input data or thread ID dependent control-flow can render large sets of computation unACE.

Figure 5 shows the normalized overhead for different SREs applied to the matrix multiply benchmark. While the instructions executed overhead is quite high for many SRE, the actual run-time overhead is much less. Duplicate integer operations resulted in 81% increase in instructions executed while only

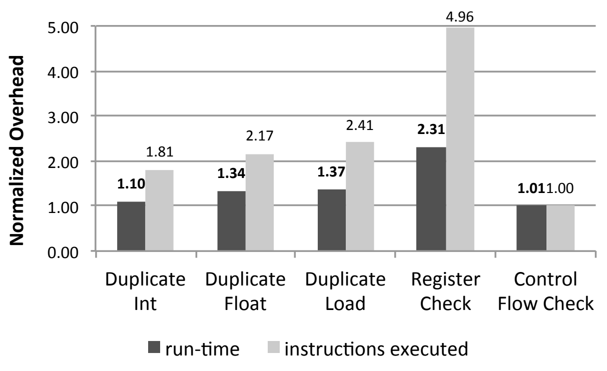

Fig. 5: Runtime overhead for different SREs applied to matrix multiplication, showing maximum potential slow down over a variety of SRE

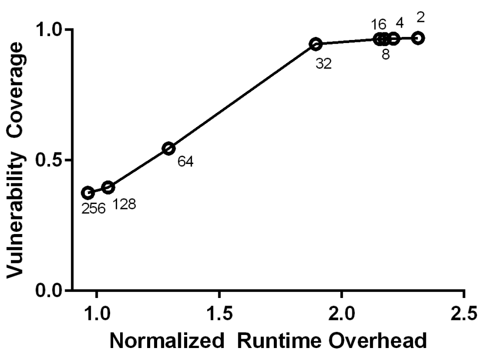

Fig. 6: The cost/benefit of Register Check injected into matrix multiplication based on live range thresholds shows linear trade-off between vulnerability coverage and performance overhead

increasing run-time by 10%. Other duplicate instruction types yielded similar difference in instruction executed and run-time overhead. The relative increase in instructions executed for duplicate load versus duplicate floating-point instructions indicates much of the compute overhead is masked by long memory operations. In the case of duplicate load instructions, since they are adjacent instructions, their temporal proximity could benefit from cache locality. The register check SRE executed 4.96x the instructions of the original kernel with only 2.31x the execution time. Again, long memory operations hide much of the increase in instruction execution. The Control Flow Check technique yielded almost no increase in runtime or instructions executed. This is because the loops of the kernel has been extensively unrolled so there are few control-flow redirections for code injections.

Figure 6 shows the cost-benefit of using Register Check on `matrix multiplication` is linear for most of vulnerability coverage with larger overhead for the last regions of vulnerability. The cost is with respect to run-time overhead normalized to the original kernel run-time. The benefit is in terms of vulnerability coverage as introduced for Figure 4. The run-time overhead is annotated with the value *K*, where variables with a live range of *K* or greater are protected with error detection. As the figure illustrates, the majority of improvement in vulnerability is achieved by protecting variables with a live range of 32 instructions or greater beyond which little additional gains in vulnerability are realized. While the performance/reliability trade off in this instance is mostly linear, other applications with a different distribution of value live ranges could result in different trade off characteristic.

VI. Related Work

There are several approaches to evaluating vulnerability. One way is to perform statistical-based random fault injection in the stack and heap through manual instrumentation [27] or automatic methods [28]. Fault-injection requires many thousands of executions to produce statistically-significant results, our approach requires one execution in an emulator. Efforts have been made to reduce error-injection sites by using static and dynamic program analysis to eliminate redundant program points [29]. Even with these methods fault-injection is still a time-consuming process and is not suitable for use at runtime. Mukherjee et al. [14] introduced *architectural vulnerability factor* (AVF) to quantify the probability a fault in a microarchitectural structure will result in an error in program output. Biswas et al. [30] took this concept and applied it to address-based processor structures. Others [16] have identified strong correlation between AVF and a subset of processor metrics. However such analysis requires costly cycle-accurate processor models and simulation infrastructure. Sridharan and Kaeli [15, 31] decoupled microarchitectural vulnerability masking from the program. This enables resiliency assessment of applications without access to (proprietary) low level microarchitectural models. We extend this approach to the GPU architecture.

There is a vast body of work on software-based error detection on both CPU and GPU. Algorithmic based fault tolerance has been introduced by [32], but must be done at algorithm-design stage and require manual involvement. Instruction Replication and other general approaches to software error detection [19]. Erez et. al. [6] proposed a fault tolerance technique using redundant execution, checkpoints at control flows that governs write backs, and control flow checking only at checkpoints. This technique focuses on the Merrimac architecture. Sheaffer et. al. [7] proposed redundant hardware execution resources to provide resilience in the face of transient faults in computational logic. Dimitrov et. al. [33] proposed three methodologies of redundant execution to achieve software reliability in GPU applications with approximately 100% overhead. One was duplicate kernel execution, and two types of redundancy: instruction-level and thread-level. Redundant multithreading in GPGPU by Wadden et al. [34] showed non-obvious slowdowns, and sometimes speedups, as a result of duplicating work at varying granularities.

This work enables application of these detectors selectively as appropriate to meet user criteria in error coverage and overhead costs. Li et al. [35] proposed SREs to reduce vulnerability of GPU applications in a seamless and customizable manner. In this work, we extend that to account for program-dependent vulnerability characteristics using PVF analysis for both sequential and combinational logic and leverage this knowledge to drive the application of SREs.

VII. Conclusion

In this paper we presented a framework for dynamic reliability management (DRM) for GPU applications. The DRM concept used program vulnerability factor (PVF) to characterize GPU kernels and a run-time code transformation to inject software reliability enhancement (SRE) techniques at the virtual ISA level. Using these two mechanisms we presented a vision of dynamic reliability management runtime that can select a combination of error-detection techniques for insertion into a GPU kernel that acknowledges a vulnerability/performance overhead trade off. We demonstrated a proof of concept implementation to characterize and reduce vulnerability of GPU kernels. Our results showed a wide range of vulnerability and performance overhead characteristics that can potentially benefit from a per-benchmark tunning to decrease both vulnerability and run-time overhead. In future works we will investigate a single, unified vulnerability metric for evaluating program vulnerability across multiple architectural structures to enable code injection of multiple SRE techniques.

Acknowledgment

This research was supported in part by the National Science Foundation under award ACI 1148310 and equipment grants from NVIDIA Corp.

References

[1] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," *IEEE Micro*, vol. 23, no. 4, pp. 14–19, Jul. 2003.

[2] NVIDIA, "NVIDIA GeForce GTX 980 (white paper)," Sep. 2014.

[3] J. Hursey, J. M. Squyres, T. I. Mattox, and A. Lumsdaine, "The Design and Implementation of Checkpoint/Restart Process Fault Tolerance for Open MPI," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007, pp. 1–8.

[4] N. Oh, P. P. Shirvani, E. J. Mccluskey, and L. Fellow, "Control-Flow Checking by Software Signatures," vol. 51, no. 2, pp. 111–122, 2002.

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

[6] M. Erez, N. Jayasena, T. Knight, and W. Dally, "Fault Tolerance Techniques for the Merrimac Streaming Supercomputer," *ACM/IEEE SC 2005 Conference (SC'05)*, no. c, pp. 29–29, 2005.

[7] 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*, pp. 55–64, 2007.

[8] M. Bernaschi, G. Parisi, and L. Parisi, "Benchmarking GPU and CPU codes for Heisenberg spin glass over-relaxation," *Computer Physics Communications*, vol. 182, no. 6, pp. 1265–1271, Jun. 2011.

[9] T. Stich, "Fermi Hardware & Performance Tips," *NVIDIA*, 2011.

[10] A. Bland, J. Wells, and B. Messer, "Titan: Early experience with the Cray XK6 at Oak Ridge National Laboratory," *Cray User Group*, 2012.

[11] NVIDIA, "Kepler GK110 (white paper)," May 2012.

[12] ——, "NVIDIAs Fermi: CUDA Compute Architecture (white paper)," Sep. 2009.

[13] J. Wadden, A. Lyashevsky, S. Gurumurthi, and V. Sridharan, "Real-World Design and Evaluation of Compiler-Managed GPU Redundant," 2014.

[14] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," ser. MICRO 36. Washington, DC, USA: IEEE Computer Society, 2003, pp. 29–.

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

[16] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," *International Symposium on Computer Architecture*, pp. 516–527, Jun. 2007.

[17] 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.

[18] 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*, pp. 58–67, Apr. 2012.

[19] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU," *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 287–300, May 2011.

[20] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi, "Modeling the effect of technology trends on the soft error rate of combinational logic," in *Proceedings International Conference on Dependable Systems and Networks*. IEEE Comput. Soc, 2002, pp. 389–398.

[21] S. Li, N. Farooqui, and S. Yalamanchili, "Software Reliability Enhancements for GPU Applications," in *Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2013)*, Jan. 2013.

[22] G. Diamos, A. Kerr, and M. Kesavan, "Translating GPU binaries to tiered SIMD architectures with Ocelot," *Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-01, January*, 2009.

[23] J. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu, "IMPACT: Parboil Benchmarks," University of Illinois, Urbana-Champaign, Tech. Rep., Mar. 2012.

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, Oct. 2009, pp. 44–54.

[25] NVIDIA, "NVIDIA GeForce GTX680 (white paper)," 2012.

[26] ——, "CUDA Profiler User's Guide," 2014.

[27] G. Bronevetsky and B. de Supinski, "Soft error vulnerability of iterative linear algebra methods," ser. ICS '08. New York, New York, USA: ACM Press, Jun. 2008, p. 155.

[28] B. Fang and K. Pattabiraman, "Gpu-qin: A methodology for evaluating the error resilience of gpgpu applications," *IEEE International Symposium on. Performance Analysis of Systems and Software (ISPASS)*, 2014.

[29] S. Hari and S. Adve, "Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults," 2012.

[30] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," vol. 00, no. C, 2005.

[31] V. Sridharan and D. R. Kaeli, "Quantifying Software Vulnerability," in *Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, ser. WREFT '08. New York, NY, USA: ACM, 2008, pp. 323–328.

[32] Kuang-Hua Huang and J. Abraham, "Algorithm-Based Fault Tolerance for Matrix Operations," *IEEE Transactions on Computers*, vol. C-33, no. 6, pp. 518–528, Jun. 1984.

[33] 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*, pp. 94–104, 2009.

[34] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 73–84, Oct. 2014.

[35] S. Li, V. Sridharan, S. Gurumurthi, and S. Yalamanchili, "Software Based Techniques for Reducing the Vulnerability of GPU Applications," in *Workshop on Dependable GPU Computing (at DATE)*, Mar. 2014.