ArmorAll: Compiler-based Resilience Targeting GPU Applications

CHARU KALRA, FRITZ PREVILON, NORM RUBIN, and DAVID KAEI,
Northeastern University

The vulnerability of GPUs to soft errors has become a first-class design concern as they are increasingly being used in accuracy-sensitive and safety-critical domains. Existing solutions used to enhance the reliability of GPUs come with significant overhead in terms of area, power, and/or performance. In this article, we propose ArmorAll, a light-weight, adaptive, selective, and portable software solution to protect GPUs against soft errors. ArmorAll consists of a set of purely compiler-based redundancy schemes designed to optimize instruction duplication on GPUs, thereby enabling much more reliable execution. The choice of the scheme determines the subset of instructions that must be duplicated in an application, allowing adaptable fault coverage for different applications. ArmorAll can intelligently select a redundancy scheme that provides the best coverage to an application with an accuracy of 91.7%. The high coverage provided by ArmorAll comes at an average improvement of 64.5% in runtime when using the selected redundancy scheme as compared to the state-of-the-art.

CCS Concepts: • **Hardware** → **Error detection and error correction**; • **Computer systems organization** → **Reliability**; **Redundancy**; **Single instruction, multiple data**; • **Software and its engineering** → **Compilers**;

Additional Key Words and Phrases: GPUs, LLVM, soft errors, fault tolerance

ACM Reference format:

Charu Kalra, Fritz Previlon, Norm Rubin, and David Kaeli. 2020. ArmorAll: Compiler-based Resilience Targeting GPU Applications. *ACM Trans. Archit. Code Optim.* 17, 2, Article 9 (May 2020), 24 pages.
<https://doi.org/10.1145/3382132>

1 INTRODUCTION

The problem of reliability has confronted both designers and users of Graphics Processing Units (GPUs) since their deployment in general purpose computing. Since GPUs were originally designed for graphics rendering, an operation that is inherently quite resilient to bit flips, little emphasis was placed on the reliability of these devices. As the scope of GPU applications has steadily expanded, they have been introduced in a number of domains that come with significant reliability constraints. These domains include scientific computing [3, 52], bioinformatics [47], molecular modeling [49], and finance [12], where workloads typically demand high precision

New article; not an extension of a conference paper.

Authors' address: C. Kalra, F. Previlon, N. Rubin, and D. Kaeli, Department of Electrical and Computer Engineering, Northeastern University, 100 Forsyth Street, Boston, MA 02115; email: {ckalra, previlon}@ece.neu.edu, n.rubin@northeastern.edu, kaeli@ece.neu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](https://permissions.acm.org).

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2020/05-ART9

<https://doi.org/10.1145/3382132>

and correctness; and autonomous vehicles that perform a range of safety-critical tasks such as pedestrian detection and avoidance, and vehicle control [22, 39].

The scaling of transistor sizes, operating voltages, and design margins has exacerbated the susceptibility of these devices to transient faults [33]. Transient faults can be caused by temperature and voltage variations, electromagnetic interference, crosstalk, and high energy particles in the atmosphere. An alpha or neutron particle strike can invert the state of a logic gate or memory cell, and may drive a wrong value temporarily [30]. Since these faults are temporary, they do not reoccur when the operation is re-executed in the future. Transient faults induced by radiation have received much of the attention, as reliability is claimed to be one of the major challenges for exascale computing [9, 28].

Hardware techniques, such as Error Correction Codes (ECC) and parity, have been used to enhance the reliability of the storage structures on a GPU, targeting DRAM, caches, and register file [8]. However, a large portion of GPU chip area is dedicated to execution units, such as ALUs. Since GPUs execute a massive number of threads concurrently, protecting the execution in these computational units is essential in order to maintain low error rates. Faults in the datapath cannot be detected by ECC as the correction codes will be generated *after* the faulty data is written to the register/memory. Traditional hardware-level solutions duplicate the entire processor to provide datapath reliability. However, processor duplication is expensive in terms of area, power, and performance overheads incurred. In addition, hardware-based solutions are inflexible and cannot be easily adapted for individual applications with varying reliability characteristics.

In contrast to hardware-based solutions, software-based approaches can be selectively employed using compiler-generated redundant threads [18, 54], but still come with some significant overhead due to synchronization across threads. Software-based redundancy can also be applied at a finer grain than a thread, providing selective redundancy on individual instructions. Software-based instruction-level replication has been studied extensively on CPUs, but it is still an under-explored research area on GPUs. Prior work has leveraged a combination of hardware and software techniques to improve the performance of full instruction duplication on GPUs [29]. However, reliability researchers have not explored the rich data analyses provided by the compiler.

In this article, we present our work that attempts to bridge this gap. Our framework is called *ArmorAll*, which is a set of compiler-based redundancy schemes designed to optimize instruction duplication on GPUs and enable researchers to explore reliability-performance tradeoffs. Our work focuses on reliable execution on the processor pipelines, while memory, caches, and register file are assumed to be protected by ECC/parity. This assumption is in line with other work in this domain [14, 29, 51]. The novel features and benefits of *ArmorAll* include:

- **Lightweight:** Duplicating all instructions can increase the register usage per thread, adversely impacting the performance and occupancy (number of active threads) of a GPU application [16, 53]. Instruction duplication techniques developed for CPUs do not have to deal with the same issues that GPUs do in terms of the register usage versus occupancy tradeoff. Moreover, CPU redundancy schemes have only been evaluated on a single thread [7, 10, 45]. Therefore, applying CPU-based techniques directly to GPUs without considering these constraints can result in a significant loss in performance. Furthermore, for each instruction duplicated, there are additional verification and notification instructions inserted, which can increase the binary size and result in performance degradation.

One of the benefits of *ArmorAll* is that it judiciously applies instruction duplication to protect only those segments of code that are likely to result in user-visible faults when experiencing a soft error. Equipped with *ArmorAll*, we can produce a GPU binary that leverages our low-cost, high-coverage, solution for addressing soft errors on GPUs.

- **Selective:** Exhaustively applying duplication everywhere in the code can significantly impact the performance of GPU applications. To provide both high fault coverage and low overhead, ArmorAll first intelligently identifies *critical instructions* in an application that produces program visible outputs. It then performs dependence analysis to identify candidate instructions that directly or indirectly influence these critical instructions. A transient fault in any one of these instructions is likely to propagate and cause a program-visible corruption. ArmorAll judiciously applies instruction duplication to these instructions to optimize coverage, while minimizing performance overhead. On average, the ArmorAll redundancy schemes duplicate between 20.4% to 52.2% of the static instructions, considerably fewer than exhaustive duplication.
- **Adaptive:** Prior research on instruction-based duplication replicated a fixed class of instructions in all applications [29, 45]. Our work differs significantly from conventional approaches by recognizing that not all applications are equally vulnerable. The subset of instructions selected by ArmorAll as candidates for duplication will vary, depending on the characteristics of the application (e.g., address-bound vs. compute-bound). This not only makes ArmorAll selective, but also adaptive in nature.
- **Portable:** The redundancy schemes provided with ArmorAll are implemented in an open-source LLVM compilation framework, which provides independence from the underlying hardware and supports portability [24]. Though our evaluation is done on a specific GPU architecture, ArmorAll can be easily applied to other existing or future GPU architectures.
- **Static:** ArmorAll does not rely on any dynamic profiling counters for assistance in choosing the best redundancy scheme. All of the decision-making by ArmorAll is done by heuristics at compile-time, which eliminates the need of executing an application multiple times on the GPU to gather statistics.

We present a technique that can adjust fault coverage, trading off application reliability for performance improvement. The key contributions of this work include:

- We propose three novel reliability-aware compiler-based redundancy schemes that can provide the desired level of fault coverage to an individual GPU application.
- We evaluate the error detection capability of ArmorAll by performing an extensive fault injection campaign on a diverse set of CUDA applications. Our results show that, depending on the redundancy scheme chosen, ArmorAll can achieve an average reduction of 100% of the Detected Uncorrectable Errors (DUEs) or 98% of the Silent Data Corruptions (SDCs), or both, while only requiring us to cover 60.7%, 42.8%, or 90.1% of the dynamic instructions, respectively.
- We compare the performance overhead of our redundancy schemes with the state-of-the-art and observe an average improvement of 64.5% in the runtime when using the chosen redundancy scheme across the studied applications.
- We develop a technique, which intelligently selects a redundancy scheme that provides the best coverage to an application at compile-time, achieving an accuracy of 91.7%.

2 BACKGROUND

2.1 Error Outcomes

Although a *fault* causes an undesired change of state in the hardware, it may or may not cause an *error* in the outcome of the program. For instance, if the location that was impacted by a transient fault is never read by the program, or was over-written by a subsequent operation before the faulty value is read, the fault will not manifest into an error. The fault may also be corrected by a

Fig. 1. Overview of NVIDIA Kepler GK110 architecture.

redundancy mechanism, such as ECC, before it propagates through the application and produces an undesirable output. When a fault does not manifest into an error, it is said to be *Masked*.

Other possible outcome categories are DUEs and SDCs. DUEs occur when a system is able to detect an error but is unable to recover from the error. This can occur when a fault causes the program to take an incorrect execution path, resulting in a system hang, crash, or other unexpected behavior. Another example of a DUE is when a fault alters an address, causing the program to access an unallocated memory location. SDCs occur when a faulty bit is used by the program, and that use results in production of a wrong program output (i.e., a visible error). Error rates in this domain are commonly measured using Failures-in-Time or FIT rate, which is the expected number of failures in 10^9 hours of operation.

2.2 GPU Architecture and Compilation Flow

Next, we describe the NVIDIA Kepler architecture, given that we have chosen it to serve as our evaluation platform [38]. While our approach can be applied to more recent NVIDIA GPUs, our present infrastructure is only supported on Kepler.

Streaming Multiprocessors (SMX) are the fundamental units of computation on NVIDIA GPU architectures, as shown in Figure 1. The smallest unit of execution is a *thread*, which are scheduled in groups of 32 parallel threads, called *warps*. Threads within a warp execute in a Single Instruction Multiple Data (SIMD) fashion. Programs that run on NVIDIA GPUs are written in the CUDA C/C++ language and compiled using NVIDIA’s LLVM-based CUDA Compiler, *nvcc* [36]. The front-end converts CUDA code into an intermediate representation, LLVM IR, and subsequently to a virtual instruction set architecture (ISA) called Parallel Thread Execution (PTX), as shown in Figure 2. The backend compiler, *ptxas*, translates PTX to machine code, called SASS, that runs on the GPU. We run our profiling and fault injection campaign at the LLVM intermediate representation (IR) level. We will describe the fault injection framework in further detail in Section 6.

IRs are low-level, machine-independent languages that compilers use to analyze and optimize code in a retargetable way. IRs allow compilers to maintain a generic set of optimizations for multiple ISAs, rather than duplicating these optimizations for each supported ISA. The LLVM IR, a form of Linear IR, resembles assembly code for some abstract machine. The algorithms iterate over simple, linear, sequences of instructions. A Control Flow Graph (CFG) uses a linear IR to represent blocks of straight code and a graph to represent the control flow among those blocks.

3 RELATED WORK

In order to detect errors in the execution state caused by transient faults, most fault protection techniques ultimately rely on some sort of redundancy (either temporal or spatial). We reflect on

Fig. 2. The flow of the NVIDIA *nvcc* CUDA compiler.

our approach in the context of prior research in the field of software and hardware-based fault mitigation on GPUs and CPUs.

Software Redundant Multithreading (RMT): Wadden et al. proposed RMT, which is implemented at a software-level targeting GPUs. RMT creates two copies of a thread, a leading thread and a trailing thread [54], which execute in parallel on different CUDA cores/processing elements. The leading thread places the value in a local or global buffer at every store instruction, and the trailing thread verifies the value and commits the executed instructions. This inter-thread communication requires synchronization for correct operation, which causes significant performance and power overheads. The leading and the trailing threads execute in parallel on different processing elements/CUDA cores to avoid deadlock. Subsequent work by Gupta et al. proposed compiler optimizations to reduce the high synchronization overhead of RMT on GPUs [13]. Typically, software-based instruction-level duplication does not incur the high synchronization overhead experienced by RMT. Programmer intervention is needed to modify the host code to ensure spare hardware resources are available for the redundant threads [29]. For example, each GPU has a maximum block size that it can support. The user must ensure that when the threads are duplicated, the permitted block size is not exceeded. If the host code transformation could be applied automatically, RMT would fail given maximum block size constraints.

Software Instruction Duplication: SInRG [29] implements instruction duplication at a compiler level. Unlike our work, SInRG is not selective, as it does not employ heuristics to identify instructions eligible for duplication. Instead, SInRG duplicates the same classes of instructions in every application, regardless of their resiliency. This approach can cause unnecessary duplication and verification overhead in applications. Since SInRG was implemented in NVIDIA's proprietary CUDA backend compiler, it provides tighter control over the SASS instructions generated in the final binary. The authors proposed specialized ISA instructions to optimize the duplication and verification process in order to limit any extra performance overhead. SInRG and ArmorAll focus on optimizing different aspects of redundancy through the compiler—SInRG optimizes the verification process, whereas ArmorAll optimizes instruction selection. Therefore, integrating SInRG with ArmorAll's proposed techniques should further improve performance and paves a path for future work.

On the CPU front, SWIFT duplicates all computational instructions and no memory instructions, thereby reducing memory pressure and improving performance [45]. It also proposes a control flow checking (CFC) mechanism to detect control flow errors. Trident calculates the SDC probability on a per-instruction basis using the values of the input operands and instruction opcodes, only duplicating those instructions that have a high SDC probability [27]. In Shoestring [10], the authors propose a compiler technique that combines symptom-based error detection with selective

instruction duplication. Their approach statically identifies *Symptom Generating* and *High Value* instructions and duplicates all instructions that appear between *Safe* and *High Value* instructions. In order to determine the probability of whether an instruction is safe, they use a *distance* metric, which is the number of statically scheduled instructions between producer and consumer instructions. However, the distance between instructions could vary, depending on the code scheduler, and is therefore prone to inaccuracies.

Moreover, these CPU-based techniques are evaluated on single-threaded applications. Applying these techniques directly on GPUs without modifying them would lead to a significant loss in performance. This is because GPUs and CPUs are architecturally quite different. On a GPU, the register usage can impact the number of concurrent threads that can be executed on the SM. CPUs do not experience many of these constraints; hence, these techniques must be optimized before applying them to GPUs.

Data Flow analysis has been explored in test-suite reduction algorithms to identify a subset of the test data that would maximize the error detection rate in software [44]. Motivated by this approach, our work attempts to use information flow in order to make a more informed decision about which instructions should be selected for duplication, without any additional hardware support.

Hardware Redundancy Schemes: Error tolerance through hardware can be achieved by using N-modular redundancy across computer nodes. The most popular forms of this redundancy are Dual Modular Redundancy (DMR) or Triple Modular Redundancy (TMR), as deployed in business-class systems such as the IBM Z Series [4]. The main problem with DMR and TMR is that they incur prohibitive energy and area overheads, which makes them unsuitable for deployment in many classes of systems. Hardware mechanisms, such as RISE and Warped-DMR, take advantage of under-utilized resources on GPUs for error detection. RISE includes techniques to predict and leverage idle SM cycles and compute cores to execute redundant work [51]. Warped-DMR provides error detection capabilities within a warp (Intra-warp DMR) and across warps (Inter-warp DMR) by exploiting underutilized resources on the GPU [14]. Both RISE and Warped-DMR require complex hardware changes and are not directly comparable with our approach. SwapCodes uses a cooperative hardware/software mechanism to reduce the overhead of intra-thread instruction duplication on GPUs [50]. The authors leverage the register file ECC logic to detect pipeline errors. Such a technique can be combined with ArmorAll to further improve application performance.

Memory protection schemes: For protecting the contents of storage elements such as the register file, caches, and main memory, ECC is applied to ensure error-free execution of the program. Schemes such as ECC and parity can detect faults only *after* valid data is written to the register file/memory and correction codes are properly generated (if applicable). In contrast, ArmorAll focuses on error detection in the datapath of the instruction *before* the output is written back to the register or memory.

4 COMPILER FOUNDATION ON DEPENDENCE ANALYSIS

To identify candidate instructions for duplication, our redundancy schemes leverage dependence analysis, which generates constraints on the execution order of instructions [2]. Dependences represent two different kinds of constraints on program transformation—(1) data dependence and (2) control dependence. Data dependence arises due to constraints designed to ensure that data is produced and consumed in the correct order. Control dependence arises when an instruction's execution depends on the outcome of a preceding instruction. We track both data and control dependence in the program, since maintaining these constraints is essential in guaranteeing program correctness.

Sample Code:	DU chains (Forward Analysis):
S0: if (c) S1: x = 10; S2: else S3: x = 5; S4: y = x;	S1->S4 S3->S4
	UD chains (Backward Analysis):
	S4->S1 S4->S3
	Control Dependence:
	S0->S1 S0->S3

Fig. 3. An example of data dependence and control dependence evaluation.

Data Dependence: Data Dependence is usually performed by tracking the data flow in the CFG using Data Flow Analysis (DFA), either statically or dynamically. Unlike dynamic DFA, static DFA does not require the program to be executed at all on a machine. Depending on the direction of the analysis, DFA can be performed as *forward* analysis or *backward* analysis.

One of the auxiliary data structures constructed during DFA are Use-Def (UD) chains (and their counterpart, Def-Use (DU) chains). We describe the construction of UD chains and DU chains using an example shown in Figure 3. In the code provided, variable x is “defined” at statement S1 and S3, while its value is “used” in statement S4. A DU chain links each *definition* of x to those *uses* that a definition can reach. Therefore, they are generated through a forward traversal from S1 (the *def* of x) to S4 (the *use* of x), and similarly S3 to S4, given that no other definition of variable x exists on the path between S1→S4 and S3→S4. Alternatively, UD chains link each *use* of variable x to the *definition*, which reaches that *use*. Therefore, they are generated through a backward traversal from S4 (the *use* of x) to S1 and S3 (the *def* of x), given that no other definition of variable x exists between S4→S1 and S4→S3.

LLVM uses Static Single Assignment (SSA) form as its primary code representation, which ensures that each variable is written only once, and each use of a register is dominated by its definition [6]. To maintain the SSA form, a phi (Φ) node is often used to select a value depending on the predecessor of the current block. A Φ node is an IR instruction, which is not implemented in the ISA for most architectures. The SSA property of LLVM simplifies the creation of UD chains, which are a key component in our implementation.

Control Dependence: An instruction, x , is control dependent on another instruction, y , if y controls whether or not x is executed. Such instructions include simple branching instructions `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmple`, `if_icmpgt`, `if_icmpge`, and so on. Since these instructions can control which instructions execute, they are a direct source of control dependence. As shown in Figure 3, S3 and S1 are control dependent on S0. However, S4 does not depend on S0 because S4 is always executed irrespective of the outcome of S0.

We ignore some sources of control dependences mentioned such as unconditional branching instructions, e.g., `br label %bb_label`. LLVM adds an unconditional branch instruction at the end of each basic block. These instructions are an artifact of LLVM and do not appear in the final assembly code generated by the backend. Moreover, though these instructions can change the flow of control for instruction execution, they are usually used with conditional branching instructions and, hence, were left out of consideration.

5 ARMORALL REDUNDANCY SCHEMES

Next, we introduce three compiler-based redundancy schemes that constitute ArmorAll. We analyzed the reliability behavior of GPU applications and designed the redundancy schemes based on the following observations:

Observation 1: Illegal memory accesses and addressing exceptions are caused not only by transient faults in registers that hold addresses, but also propagation of an earlier fault through the computation tree leading to those registers.

Observation 2: An incorrect output is often a result of corruption caused by either a bit flip in a register that holds an output value, or propagation of an earlier fault through the computation tree leading to one of those registers.

All three schemes leverage dependence analysis and UD chains, covered in Section 4, to create a sequence of instructions, which we refer to as a *path*. A path, for instance, contains a sequence of instructions – $\{u_n, u_{n-1}, u_{n-2}, \dots, u_1, u_0\}$, where:

- u_i : use of the value v_i , which is *defined* in u_{i-1} , where $i \in \mathbb{N}$ or,
- u_i : is an instruction, which is control dependent on u_{i-1} , where $i \in \mathbb{N}$.
- The last instruction in the path is u_0 : *def* of the value *used* in u_1 .

Since the path is generated through a *backward* walk, we use descending order to enumerate the instructions.

A path always begins from the *root* instruction, u_n and terminates when no more instructions can be traversed in that path. However, an application may contain multiple paths, depending on the number of root instructions identified. It is possible for an instruction to be part of multiple paths; in such cases, the instruction is only considered once. The two factors that distinguish the three schemes we propose are: (1) their root instruction and (2) the final worklist constructed. The root varies depending on the entity that is being protected. The entity could be an address, a value, or both. The worklist includes an unordered set of instructions that appear on the path being traversed, starting from the root.

Once the final worklist is created, all instructions in the worklist (with a few exceptions) are considered *duplication-eligible* and subsequently duplicated. The duplicate instruction reads the same source operands as the original instruction but writes to a new destination register. The destination operands of the instruction are compared with their clones via additional verification instructions. If the verification fails, a mismatch warning is issued to the user. We also do not duplicate Φ operations (if they appear in the worklist) and unconditional branch instructions, as these instructions do not translate into actual instructions in the final binary, as mentioned in Section 4. Next, we describe our three schemes, and how they protect these entities, in greater detail.

5.1 Address Armor

Key Idea: As the name suggests, Address Armor (AA) protects the *addresses* used by memory instructions. To address the first observation, the goal of AA is to track all instructions that participate in memory address computation. Since both load and store instructions access memory, they naturally become the root for this scheme. AA *does not* protect the value loaded or stored by the instructions, only their addresses.

Implementation: In LLVM, the instruction *getelementptr* (GEP) is typically used to compute addresses for both load and store instructions. Therefore, the Address Armor first initializes a worklist with all root/*getelementptr* instructions present in the program. For each GEP instruction in the worklist, it tracks all instructions that appear in the data and control dependence path and

adds them to the worklist, if not previously added. This process continues until the path for all GEP instructions is completely traversed and no further instructions can be added to the worklist. The final worklist, S_{AA} , is a set of instructions, expressed as the union of all paths $\bigcup P_j$, where P_j is the sequence of instructions traversed, starting from the root $j \in \mathbb{N}$.

In AA, we do not duplicate load and store instructions themselves. Since the addresses of these instructions are protected by AA, there is no need to execute them twice. One might argue that corruption in their value can still lead to an incorrect output. However, protecting the value is not the objective of AA. We relax the constraints by assuming that when an application is protected using AA, the likelihood of the values getting corrupted is low.

Summary:

- Goal: To protect *addresses* used by load and store instructions.
- Root: Load and Store instructions.
- Which instructions from the worklist are duplicated? ALU, Floating point, Compare, and GEP instructions.
- Which instructions are *not* duplicated? Loads, Stores, Barriers, Phi, and Unconditional Branch instructions.

5.2 Value Armor

Key Idea: This scheme, as its name suggests, is based on our second observation and protects the *values* written to memory by store instructions. The goal of Value Armor (VA) is to track all instructions that contribute to the computation of an output value. Since the output value is written to memory using store instructions, they become the root instruction for this scheme. VA does not protect the addresses used by a store (or a load) instruction, only the values.

Implementation: VA first initializes a worklist with all the root instructions (here, store instructions). For each root instruction in the worklist, it tracks all instructions that appear in the data and control dependence path and adds them to the worklist. A path terminates when it either encounters a load instruction, or if there are no further instructions that can be added to the path. If VA keeps tracking dependence beyond load instructions (Syntax: `%val = load i32* %ptr`), then its path will include instructions that compute the address of the load, which is contrary to the goal of the VA. Therefore, all instructions are added to the worklist until one of the termination conditions (described above) is encountered. The final worklist, S_{VA} , is a set of instructions, expressed as the union of all paths $\bigcup P_k$, where P_k is the sequence of instructions traversed, starting from the root $k \in \mathbb{N}$.

Since the value must be protected by VA, we also duplicate the load instructions that appear on the worklist, unlike AA. In this case, we assume that when an application is protected using VA, the likelihood of the address being corrupted is low.

Summary:

- Goal: To protect *values* used by load and store instructions.
- Root: Store instructions.
- Which instructions from the worklist are duplicated? ALU, Floating point, Compare, Load, and GEP instructions.
- Which instructions are *not* duplicated? Stores, Phi, Barriers, and Unconditional Branch instructions.

5.3 Hybrid Armor

Key Idea: This scheme, as its name suggests, protects both the *values and addresses* of load and store instructions. The goal of Hybrid Armor (HA) is to track all instructions that contribute to

the computation of the output value *and* addresses. Since the output value is written to memory using store instructions, they are also the root for this scheme.

Implementation: The HA scheme first initializes a worklist with all of the root instructions (i.e., store instructions). Similar to AA and VA, for each root instruction in the worklist, HA tracks (using a backward traversal) all instructions that appear in the data and control dependence path and adds them to the worklist. The major difference between HA and VA is that HA does not terminate the UD chain at load instructions (Syntax: `%val = load i32* %ptr`). Since HA is a combination of VA and AA, the path for HA must include instructions that compute the memory address (`%ptr`) as well. Therefore, we take into account the path beginning from load instructions, without explicitly designating them as root instructions. This process continues until the path for all *store* instructions is completely traversed, and no further instructions can be added to the worklist. The final worklist, S_{HA} , can be expressed as the union of set of instructions in AA and VA, $S_{AA} \cup S_{VA}$, where S_{AA} and S_{VA} are the final worklists generated by AA and VA, respectively. Since both the value and address of a store are protected by HA, there is no need to duplicate the store instruction.

Summary:

- Goal: To protect both *values and addresses* used by load and store instructions.
- Root: Store instructions.
- Which instructions from the worklist are duplicated? ALU, Floating point, Compare, Load, and GEP instructions.
- Which instructions are *not* duplicated? Stores, Phi, Barriers, and Unconditional Branch instructions.

6 EVALUATION METHODOLOGY

Fault Injection Framework: As mentioned in Section 1, we focus on the reliability of the processor pipeline. Traditional methods to protect the execution units require the entire pipeline to be duplicated, which is prohibitively expensive. Therefore, one way to emulate transient faults in the execution units is to flip random bits in destination registers using software. Injecting faults in a destination register is equivalent to a fault occurring in the execution unit. If a fault occurs in a destination register and propagates and pollutes subsequent instructions, the impact will be captured by ArmorAll. In this work, we assume that memory is protected by ECC.

In this article, we use an LLVM-based fault injection framework called LLFI-GPU [26]. The benefits of using LLVM are that it offers portability and modularity by isolating itself from the high-level language, as well as from the ISA and other hardware-specific details. Faults can be injected either at a hardware or software level [5, 31, 48]. Hardware methods can inject faults into components such as chip pins and caches, which are not accessible to a software fault injector. LLFI-GPU captures the faults that are visible at the application level and are not masked by the hardware. Microarchitectural fault injection studies are typically done on a simulator, enabling faults to be injected in hardware structures such as caches, which are not exposed to the application [15]. Our study injects faults in the program state while running on real hardware. We anticipate that our conclusion would not differ significantly if microarchitecture-level bit flips were detected at the application level (i.e., faults that are not masked).

Though we use a NVIDIA Kepler K20 to perform our fault injection campaign, it is important to note that our redundancy schemes are not tied to any specific GPU architecture. The choice of the Kepler is tied to the LLFI-GPU toolset. Moreover, the LLVM framework is agnostic to the source language, as well to the hardware specifics. There are no known limitations of using LLVM that prevent ArmorAll from being effective on GPUs belonging to the same or different vendors.

LLFI-GPU is hosted in the `nvcc` compiler stack, labeled as the “fault injector” in Figure 2. The LLVM IR is not directly exposed to the user; LLFI-GPU attaches a dynamic library to `nvcc`, which

can intercept the call to the LLVM compilation module [1]. At this point, the passes of LLFI-GPU are invoked to instrument the program. LLFI-GPU then returns the instrumented LLVM IR to *nvcc*, which proceeds with the rest of the compilation process to transform it into PTX code. Our compilation passes are implemented as the last step in the LLFI-GPU framework, in order to leverage the existing LLVM optimizations and to ensure that no LLVM optimization can further modify the code. We further analyzed the SASS code of all CUDA applications using the NVIDIA tools, *cuobjdump* and *nvdiasm*, to ensure that our code was not optimized by the *ptxas* backend [35]. The output of the duplicate instructions are verified with the original instructions to notify the user of a mismatch; hence, they are not treated as dead code by the backend. The fault model of LLFI-GPU considers transient hardware faults that occur in the computational elements of the GPU, including pipeline stages, flip-flops, ALUs, and the register file [26]. It assumes GPU memory or cache, its control logic, and instruction encodings are protected with ECC. This fault model is inline with other work in the area [19, 29, 46].

Benchmarks: We evaluate the error detection capability of ArmorAll by performing an extensive fault injection campaign on a diverse set of CUDA applications from the CUDA SDK [37]. We achieve a 95% confidence interval and a 3.1% error margin by performing 1,000 injections in each application [25]. In prior work, achieving a 95% confidence interval, with a <5% error margin, has been deemed *accurate* [17, 20, 34, 42, 43]. We inject single-bit flips into the destination register of a randomly selected LLVM instruction (one fault per run of the application). When checking for errors, we compared the output of the kernel with the fault injected against the golden output (without any fault injection). For precision-based applications, we used the default values of the L1 and L2-Norm provided in the benchmark, which typically ranged between 5e-4 to 1e-6. To gather GPU performance and occupancy statistics, we used the *nvprof* tool and measured the application runtime on a real Kepler K20.

Outcome Classification: Before we dive deeper into the error detection capabilities of each redundancy scheme, we will describe the error outcomes that are plausible when a fault is injected. As mentioned in Section 5, the duplicate instruction reads the same source operands as the original instruction, but writes to a new destination register. The destination operands of the instructions are verified (with their clones) via additional verification instructions. If the verification fails, a mismatch warning is issued to the user. However, we allow the program to continue execution, even if a mismatch is detected. This enables us to evaluate the accuracy of the detection scheme, and if we were being too conservative. Our fault injector design ensures that faults are not injected in the verification instructions. However, in an actual implementation, verification instructions could be affected by transient faults. In any reliability scheme, the checker is a potential point of failure in the system.

When a fault is injected in the program, one of the outcomes described in Table 1 is possible. When a fault *does not* manifest into an error upon program completion, and no mismatch is detected during its execution by ArmorAll, we categorize it as *Masked:undetected*. This is an ideal case that leverages the inherent resiliency of the application. However, since our redundancy schemes duplicate the instructions and verify their output with the clones immediately, it does not provide much opportunity for masking. Sometimes a fault occurring in a duplication-eligible instruction could have been masked if it was allowed to propagate; instead, a mismatch was detected early on due to the conservative nature of ArmorAll. Such a category is called *Masked:detected*.

When a faulty bit is read by the program and causes a wrong program output (i.e., a visible error), an SDC is said to have occurred. If the propagation of the fault is detected by ArmorAll before it corrupts the output of the program, it is considered an *SDC:detected*. The higher the number of SDC:detected outcomes means better error protection. If the fault corrupts the output value and the fault was not detected during its propagation, it is referred to as *SDC:undetected*.

Table 1. Possible Error Outcomes for ArmorAll Redundancy Schemes

Outcome	Synopsis	Behavior
Masked:undetected	Fault was masked and no mismatch was detected	Desirable (Higher is better)
Masked:detected	Fault that would have potentially been masked, but a mismatch was detected by the Armor	Undesirable (Lower is better)
SDC:undetected	SDC that was not detected by the Armor	Undesirable (Lower is better)
SDC:detected	SDC that was detected by the Armor	Desirable (Higher is better)
DUE:undetected	Crash/DUE that was not detected by the Armor	Undesirable (Lower is better)
DUE:detected	Crash/DUE that was detected by the Armor before it occurred	Desirable (Higher is better)

We allow the program to continue execution, even if a mismatch is detected. This enables us to evaluate the accuracy of the detection scheme.

Finally, DUEs occur when a system is able to detect an error but is unable to recover from the error. Our goal is to detect errors that can cause these DUEs *before* they occur so that they can be prevented. We refer to the errors that were detected by ArmorAll before the DUE occurred as *DUE:detected*. If ArmorAll was unable to detect them and the system eventually crashed or completed execution with any other undesirable behavior, it is called *DUE:undetected*.

While our focus in this work is Error Detection, our proposed redundancy schemes can be augmented with any appropriate Error Recovery mechanism, such as checkpointing, to allow applications to rollback to the checkpoint and restart execution when a mismatch is detected [11, 23]. Checkpointing schemes on GPUs require saving the state so that the application can be restarted. The majority of the checkpointing overhead comes from transferring data from the GPU to the host. Recent checkpointing studies demonstrated a fast checkpoint-recovery scheme on GPUs, introducing an average overhead of 6% to application runtime [11]. Their work aims to optimize the checkpointing overhead by overlapping computation with the storage of a checkpoint image to a stable storage system. Error correction techniques, such as instruction triplication, can also be used when a mismatch is detected. Error recovery or correction are interesting avenues for future research, but are currently beyond the scope of this work.

7 RESULTS

Next, we will evaluate the detection coverage and the associated overhead of the three redundancy schemes described in Section 5. We will also propose a set of heuristics that will allow the compiler to choose the best redundancy scheme out of the three.

7.1 Instruction Duplication Overhead

Static: We first measure the instruction coverage as a fraction of the number of static instructions in the program that are assumed to be protected via instruction duplication. As shown in Figure 4, the range of static instructions duplicated for AA varies from 8–54%, with an average coverage of 35.9%. VA duplicates between 2%–42% the instructions, with an average of 20.4% of the static instructions. Since HA protects both values and addresses of load and store instructions,

Fig. 4. Percentage of static instructions duplicated for each Armor. Average duplication for AA, VA, and HA are 35.9%, 20.4%, and 52.2%, respectively.

Table 2. Min, Max, and Average Increases in the Program Binary Size for ArmorAll Redundancy Schemes as Compared to the Baseline with No Duplication

Aarmor	Min	Max	Average
Address Armor	8.2%	53.3%	26.2%
Value Armor	7.4%	27.6%	19.2%
Hybrid Armor	9.2%	54.8%	27.5%

The program binary includes the non-duplicated host code.

the duplication rate is higher than with AA and VA. Its coverage ranges between 38%–62%, with an average of 52.2% of the static instructions duplicated. For each instruction duplicated, there is an additional verification instruction to compare the outputs of the original instruction and the duplicated instruction. The average code bloat (increase in the size of the program binary, including the unduplicated host code) for AA, VA, and HA is 26.2%, 19.2%, and 27.5%, respectively. The baseline used for comparison does not perform any instruction duplication. For all Armor types, the minimum increase in binary size was observed in Vector Addition (VECADD), whereas the maximum increase was observed in Matrix Transpose (TRNPOS) applications, as shown in Table 2. The size of the binary is one indication of the size of the memory footprint for the application. It can stress the cache performance and memory bandwidth on the GPU.

Dynamic: Next, we evaluate how the static coverage translates into dynamic instruction overhead. Figure 5 shows the percent increase in the dynamic instructions executed, as compared to the baseline. The dynamic instructions executed in the Armors include the duplicated instructions, verification instructions, as well the notification instructions. The baseline does not provide any duplication. The percent increase in the dynamic instruction count varies across applications, with an average increase of 60.7%, 42.8%, and 90.1% for AA, VA, and HA, respectively. As expected, HA protects both addresses and values; hence, it adds more duplication and verification to the kernel. The instruction overhead of the Hybrid method is not always the same as the sum of the overheads for the individual Value and AA methods. The instructions duplicated by Value and AAs may not be mutually exclusive. For example, instructions that handle control dependence (if-else) can fall in the path of both Address and VA but are only considered once during HA. This explains the behavior of TRNSPOS and VECADD in Figure 5. The number of verification instructions can be reduced by delaying the verification after a few instructions by using a signature register [29].

Fig. 5. Percentage increase in dynamic instruction count compared to the baseline. Average increase for AA, VA, and HA are 60.7%, 42.8%, and 90.1%, respectively.

Fig. 6. Fault injection results showing error detection capability of AA. AA is able to detect all DUEs and some SDCs.

However, there is a risk that an application may crash/hang before the delayed check is encountered. Therefore, VA might benefit more from delaying verification than AA or HA as the risk of DUEs is assumed to be lower in case of applications protected by VA.

7.2 Error Detection Capability of ArmorAll

As mentioned earlier, we allow an application to continue execution even if a mismatch is detected. This allows us to evaluate the accuracy of the detection and know whether we were too conservative. When a fault is injected, one of the outcomes listed in Table 1 is possible. We discuss the detection capability of each Armor for DUEs, SDCs, and Masked, in this order.

7.2.1 Address Armor. In this section, we will focus on the error detection capability of AA. The goal of AA is to protect the addresses used by load and store instructions. Typically, illegal memory accesses can cause DUEs, which can be prevented if the instructions computing memory addresses are protected. As shown in Figure 6, AA is highly efficient and is able to detect 100% of the DUEs (right-most bars) for the given set of applications, despite duplicating only 35.9% (on average) of the static instructions (see Figure 4). However, DUEs triggered by the hardware may not be captured by ArmorAll, since the application can crash before the duplicated instruction is executed.

Fig. 7. Fault injection results demonstrating the error detection capability of VA. VA can detect some SDCs, but no DUEs.

Interestingly, one of the unexpected behaviors in AA is that it is also able to detect SDCs (middle bars), along with DUEs. This is because AA was able to prevent threads from accessing incorrect (but not illegal) memory addresses and reading/writing wrong values, which can eventually lead to an SDC. Therefore, AA is able to protect more than what it was originally designed for.

Due to the conservative nature of ArmorAll, we are able to detect some of the outcomes that would have been masked if the error was allowed to propagate. However, AA is still able to provide some undetected error masking in all applications. We will discuss the masking offered by ArmorAll in further detail in Section 8.

7.2.2 Value Armor. Figure 7 shows the error detection capability of VA. The goal of VA is to protect the values used by load and store instructions.

As shown in Figure 7, VA is unable to detect any DUEs, which is an expected behavior because protecting against DUEs is not the primary goal of VA. Protecting output values can help detect SDCs, as seen in many applications. In case of BlackScholes (BLKSCH) and Matrix Multiplication (MATMUL), VA is able to detect more than 95% SDCs with about 42% and 19% instruction coverage, respectively, for the two applications. Similar behavior can be seen in the case of ConvolutionFFT2D (CFFT2D), Finite Difference Time Domain (FDTD3D), Histogram (HISTO), Scalar Product (SPROD), Scan (SCAN), and Sobol Quasi Random Number Generator (SQRNG), where VA can detect over 70% the SDCs, with an average instruction coverage of 26%.

For the Discrete Walsh Transform (DWT1D), Matrix Transpose (TRNPOS), and Vector Addition (VECADD) workloads, VA does not perform as well. This is because these applications are mainly address-bound (computing addresses). In TRNPOS, the kernel performs a large number of matrix-based moves (loading from one location in the matrix and storing to a different location). All other instructions in this kernel compute addresses. This behavior can also be observed in Figure 4, which shows that only 2% of the instructions are duplicated by VA. Therefore, TRNPOS does not benefit from VA.

In the case of VECADD, the dominant operation is vector addition of two vectors A and B, expressed as: $C[i] = A[i] + B[i]$, where $i = blockDim.x * blockIdx.x + threadIdx.x$. Here, i holds the offset from the base address of each vector. The effective address is computed using the LLVM `getelementptr` instruction, which returns a pointer to the value at (base address + offset). If a fault corrupts the value of i , it changes the offset. Thus, a thread could end up reading the value from

Fig. 8. Fault injection results, demonstrating error detection capability of HA. HA can detect almost every error, but is also quite conservative (high percentage of Masked:detected outcomes).

another index belonging to another thread. On GPUs, all threads can access the global memory space and nothing prohibits a thread from reading another global memory location. In addition, a fault in the value of i can also cause DUEs if the effective address computed is beyond the legal memory space assigned to the application. This is because bit flips in more significant (i.e., high order) bits can cause larger deviation in the offset resulting in memory exceptions. Since VA does not protect addresses, it will be unable to capture this behavior.

Lastly, VA shows high Masked:undetected for applications that are address-bound, or have a mix of addresses and data values, such as MATMUL. This is because most of the instructions in these applications are not duplicated using VA; hence, there is more opportunity for Masking. In contrast, applications such as BLKSCH, SQRNG, and BINOPT are floating-point applications. Therefore, any slight change in their values will be detected by VA, reducing the likelihood of Masking.

7.2.3 Hybrid Armor. Figure 8 shows the error detection capabilities of HA. HA protects both values and addresses used by load and store instructions.

Similar to AA, HA is able to detect all DUEs. In terms of SDC detection, HA provides the combined detection capability of AA and VA, and can detect over 98% of the SDCs across all applications. However, HA is more conservative than AA and VA, leaving little room for any potential masking opportunity. This can be seen in the high percentage of Masked:detected outputs across all applications. This overhead comes with high coverage requirements.

7.3 Performance Overhead

In this section, we analyze the performance overhead of our three schemes as compared to a baseline that uses no instruction duplication. In addition to comparison with the baseline, we also quantitatively compare our results with a prior state-of-the-art software-based approach that protects all execution units using thread-level duplication (TLD) [54]. We recognize that TLD provides additional coverage to the register file and local memory, which we assume is protected by ECC. Figure 9 shows the execution time of applications run with ArmorAll and with TLD, relative to the baseline (without any duplication). The average increase in runtime for AA, VA, and HA versus the *uninstrumented baseline* is 1.7X, 1.39X, and 2X, respectively. Compared to TLD, the average improvement in the runtime is 42.6%, 53.2%, and 32.2% for AA, VA, and HA, respectively. We discuss the sources of slowdown in both TLD and ArmorAll.

Fig. 9. Runtime of applications equipped with ArmorAll and with TLD, relative to the baseline (no duplication). The average improvement in the runtime, as compared to TLD, is 42.6%, 53.2%, and 32.2% for AA, VA, and HA, respectively.

Fig. 10. Occupancy of applications with and without ArmorAll.

State-of-the-art: (1) TLD creates a duplicate copy of each thread, identified as a leading thread and a trailing thread. At every store instruction, the leading thread places the value in a global buffer, and the trailing thread reads the value from the buffer and verifies it with its private copy before committing the store instruction. This process requires explicit synchronization between the leading and trailing thread to ensure that the values are written and read in the correct order. The high overhead in TLD can mainly be attributed to inter-block communication and synchronization between threads during store instructions.

(2) Unlike ArmorAll, programmer intervention is needed in TLD to modify the host code to ensure spare compute resources are available for the redundant threads. Workloads can benefit from TLD if they have spare resources available (e.g., lower than 50% occupancy); however, many of the GPU applications we studied have an occupancy greater than 50%, as shown in Figure 10. Despite the availability of spare compute resources, the synchronization overhead with TLD can adversely impact the performance of an application.

Our approach: For ArmorAll, there are three sources of slowdown across applications. First, due to additional instructions added to the kernel (for duplication, verification, and notification), there are more dynamic instructions executed per thread, which increases the runtime. Second,

the addition of these new instructions can introduce new dependencies in the kernel and limit opportunities to schedule and software pipeline instruction sequences. These added dependencies can contribute significantly to the performance overhead.

Third, the duplicate instruction reads the same input operand and writes to a new destination register. This may increase the overall register pressure, depending on the number of instructions duplicated, and consequently, impact the warp occupancy of the application. Occupancy is the number of active warps, divided by the maximum number of warps that can be placed on an SM. One of the ways to determine the number of active warps is based on the maximum resources used by the kernel. The three resources that limit kernel occupancy are (1) local/shared memory usage, (2) block size (number of threads in a block), and (3) register usage. All three metrics can be extracted statically to provide an upper bound on the kernel occupancy. The results shown in Figure 10 demonstrate the occupancy of the baseline (no duplication) and after ArmorAll was applied. The local memory usage and block size remained the same in the baseline and the ArmorAll-equipped applications. Only the kernel's register usage changes, as additional duplicate instructions are inserted in the code. The duplicated instruction reads the same input operand and writes to a new destination register. This may increase the overall register pressure, depending on the number of instructions duplicated. Therefore, the change in occupancy shown in the figure is a result of changing register pressure.

From our experimentation, most of the applications did not show much change in their occupancy when executed with the Armors as compared to the baseline. The only exceptions were BINOPT, BLKSCH, and MATMUL. In the case of BINOPT and MATMUL, the reduction in occupancy had an adverse impact on the execution performance. However, reduction in occupancy may not always reduce the execution performance. This can be observed in BLKSCH where there is not much change in the runtime of the application, despite a reduction in occupancy. Sometimes a reduction in occupancy can also improve the performance of an application, although we did not observe this phenomenon in our applications. Running fewer threads on the SM can reduce memory contention and improve the runtime of the application. Therefore, low occupancy does not necessarily imply poor performance. Occupancy is an important factor for ArmorAll methods, as it may negatively impact performance. The question of whether changing occupancy will impact the performance when ArmorAll is applied depends heavily on the nature of the application and requires further investigation.

7.4 Choosing the Best Armor for an Application

In this section, we propose heuristics to identify the redundancy scheme that is likely to offer the best error protection to an application. Our heuristics utilize static information, which allows the compiler to select the redundancy scheme at compile-time. We choose to not use the commonly-used performance metrics, such as memory intensity, or compute intensity for characterizing reliability as they can be misleading [21]. Our results have shown that faults in instructions that compute addresses for load/store instructions can also cause illegal memory accesses, and not just load and store instructions themselves. Therefore, solely focusing on the number of loads and stores in an application may not provide the complete picture.

Instead, we measure the *cardinality* of the final worklist generated by each Armor as a metric to choose the best Armor. As mentioned in Section 5, each Armor constructs a worklist, which is a set of unique static instructions that appear on the data and control dependence path, starting from a root instruction. Mathematically, cardinality is defined as the number of elements present in a given set. In this context, cardinality is the number of static instructions in the final worklist constructed by the Armor, after removing the duplication-*ineligible* instructions. The remaining instructions are considered highly vulnerable and must be duplicated. We use the following

notation to represent cardinality:

$|S_{AA}|$ = Cardinality of the worklist generated by AA

$|S_{VA}|$ = Cardinality of the worklist generated by VA

$|S_{HA}|$ = Cardinality of the worklist generated by HA

If the cardinality of AA is greater than the cardinality of VA, it implies that the application will be better protected if AA is chosen. The rationale behind this scheme is that the longer the total path, the more opportunities there are for the fault to propagate to the output. Also, a larger number of instructions will expose more hardware logic to transient faults, resulting in a higher error probability. As these instructions are considered highly vulnerable, the likelihood of an application failing is higher if they are left unprotected. For example: if $|S_{AA}| = 80$ and $|S_{VA}| = 10$, it would be more logical to guard the application using AA rather than VA, as 80 instructions are more likely to cause an error than 10 instructions. Similarly, if $|S_{VA}| \ll |S_{AA}|$, it is better to invoke VA. However, if the $|S_{AA}| \approx |S_{VA}|$, then HA is expected to provide better overall coverage.

We summarize the selection criteria as follows:

$|S_{AA}| \gg |S_{VA}| \rightarrow \text{Address Armor}$

$|S_{AA}| \ll |S_{VA}| \rightarrow \text{Value Armor}$

$|S_{AA}| \approx |S_{VA}| \rightarrow \text{Hybrid Armor}$

The cardinality of each application is reflected in Figure 4 as a fraction of the total number of static instructions in the application. In the case of BLKSCH, the cardinality of VA is clearly much higher than for AA. The error detection results also reflect that AA is not able to protect against most of the SDCs, whereas VA can successfully overcome 90% of the errors. Therefore, choosing VA will provide better error protection in the case of BLKSCH. For applications DWT1D, FDT3D, HISTO, MATMUL, SQRNG, TRNSPOS, and VECADD, the cardinality for AA is 2X higher than the cardinality of VA. This can be seen in Figure 6, showing AA can protect against more than 80% of the DUEs and SDCs in these applications, on average.

For BINOPT, CFFT2D, SPROD, and SCAN, the AA to VA cardinality ratio is close to 1. Therefore, HA is expected to provide better coverage for these applications. However, in the case of SPROD, HA can be too conservative as AA can protect against most of the errors. Overall, our heuristic has been able to choose the Armor correctly for 11 out of 12 of the applications (91.7% accuracy), with SPROD being the only exception. We observe an average improvement of 64.5% in runtime versus TLD when employing the best redundancy scheme chosen using the proposed selection criteria.

In our approach, the cardinality is measured using the number of static instructions in the worklist. It does not take into account the execution frequency of these instructions. The number of times an instruction executes can change the dynamic path length of the redundancy schemes and impact the selection criteria. However, for the applications we studied, the trend of static and dynamic instructions has remained the same for most applications, as seen in Figures 4 and 5. Using the dynamic instruction count is an alternative; however, it will not improve the overall prediction accuracy for the current set of applications. Moreover, making a decision at compile-time further saves the overhead of re-executing an application on the GPU. By calculating the cardinality of AA, VA, and HA statically, the compiler can choose the most appropriate Armor without relying on dynamic profiling information. The selection criteria can be extended in the future to include the weights of the instructions in the worklist as more applications are added to the suite.

Fig. 11. Fault injection results by using approximate error detection in HA. The detection precision used here is 0.1. Relaxing error detection increases the percentage of Masked:undetected outcomes in floating point applications, making HA less conservative.

8 APPROXIMATE ERROR DETECTION

The error detection results show that HA is quite conservative, as it detects most of the errors that could have potentially been masked. The drawback of being conservative is that when ArmorAll is augmented with any error recovery scheme (e.g., check-pointing), the application must roll back to a checkpoint and restart from there whenever a mismatch is detected. This can cause significant execution overhead. The more conservative an approach is, the more the application may suffer from rollback-restart overhead; and this may be unnecessary many times.

In this section, we try to relax the detection constraints in ArmorAll. One of the opportunities to relax and improve ArmorAll's detection capabilities is during execution of floating-point applications/operations [40]. We observed that in floating point operations, detecting a small perturbation in a value may not always impact the final program output. This prompted us to experiment with different *detection precisions*, starting with a difference of 0.1, as shown in Figure 11. In this case, we only report a mismatch when the difference in the output of the original and duplicated (floating point) instruction is greater than 0.1. In other words, we allow any difference less than 0.1 to propagate, and observe its impact on the output of the application, as well as the behavior of ArmorAll. We do not modify the default L1 or L2-Norm used to compare the final value(s) with the golden output in these applications. The value of 0.1 was selected experimentally and is used to test the potential of approximate detection. We also experimented with a precision of 0.001, but did not observe any significant change in the results. For the sake of brevity, we only show the results for a detection precision of 0.1. In future work, we plan to leverage Algorithmic Differentiation to help identify the best precision value to use for error detection [32].

To demonstrate the results for approximate detection, we only chose a subset of the applications, focusing on those that use floating point operations. We have chosen HA to demonstrate approximate detection since it was found to be the most conservative. However, the approach can also be applied to AA and VA to increase the number of Masked:undetected outcomes, thereby improving their performance. Most applications show an increase in Masked:undetected outcomes, as intended, while BINOPT and BLKSCH show the largest increase. SQNRG primarily uses integer operations and eventually converts those values to floating point, hence, not leaving much room for approximate detection. Some applications, such as BLKSCH, FDTD3D, and VECADD also show a slight increase in the number of undetected SDCs. This side effect is because some of the deviations under 0.1, which were not detected, did not get masked and eventually propagate to the output.

In practice, it may not be possible to assure *a priori* that we can eliminate all causes of unreliability. Therefore, the goal of fault tolerance is to reduce the system failure rate, increasing the availability to an acceptable level [33]. ArmorAll provides opportunistic coverage with no provable guarantees for systems having real-time constraints, such as safety and mission-critical systems. This makes our approach more suitable for commodity systems and selected classes of HPC applications. Performing radiation testing on ArmorAll-equipped applications will provide accurate Failure-in-time (FIT) estimates and determine ArmorAll's suitability to safety and mission-critical systems [41].

9 CONCLUSION

In this article, we propose three novel reliability-aware compiler-based redundancy schemes that can provide adjustable levels of fault coverage to individual GPU applications. The redundancy schemes are designed with the goal of providing selective coverage to applications by protecting one or both of these entities—address and/or values used by load/store instructions. Our evaluation shows that, depending on the redundancy scheme chosen, ArmorAll can achieve an average reduction of 100% of DUEs or 98% SDCs or both, while only requiring to cover 60.7%, 42.8%, or 90.1% of the dynamic instructions, respectively. In comparison to a state-of-the-art scheme, the performance overhead of our redundancy schemes shows an average improvement of 64.5% when using the best redundancy scheme across the studied applications. For future work, we plan to implement a deferred verification technique to further improve the performance of ArmorAll, as well as integrating the toolset with recovery mechanisms. One potential technique that could reduce the overhead of HA is to track the execution frequency of the instructions in final worklist and only duplicate those instructions that execute most often. This way, runtime information can be used to improve the performance of HA. We also plan to perform a quantitative comparison of ArmorAll with prior instruction duplication techniques in future work.

REFERENCES

- [1] [n.d.]. Enabling on-the-fly manipulations with LLVM IR code of CUDA sources. Retrieved from <https://github.com/apc-llc/nvcc-llvm-ir>.
- [2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA.
- [3] E. Alerstam, T. Svensson, and S. Andersson-Engels. 2008. Parallel computing with graphics processing units for high-speed Monte Carlo simulation of photon migration. *Journal of Biomedical Optics* 13, 6, Article 060504 (2008) <https://doi.org/10.1117/1.3041496>
- [4] W. Bartlett and L. Spainhower. 2004. Commercial fault tolerance: A tale of two systems. *IEEE Transactions on Dependable and Secure Computing* 1, 1 (Jan. 2004), 87–96.
- [5] Ian Briggs, Arnab Das, Mark Baranowski, Vishal Sharma, Sriram Krishnamoorthy, Zvonimir Rakamaric, and Ganesh Gopalakrishnan. 2019. FailAmp: Relativization transformation for soft error detection in structured address generation. *ACM Transactions on Architecture and Code Optimization* 16, 4, Article 50 (Dec. 2019), 21 pages. DOI : <https://doi.org/10.1145/3369381>
- [6] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct. 1991), 451–490. DOI : <https://doi.org/10.1145/115372.115320>
- [7] Moslem Didehban and Aviral Shrivastava. 2016. NZDC: A compiler technique for near zero silent data corruption. In *Proceedings of the 53rd Annual Design Automation Conference (DAC)*.
- [8] Martin Dimitrov, Mike Mantor, and Huiyang Zhou. 2009. Understanding software approaches for GPGPU reliability. In *Proceedings of the 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU-2)*. ACM, New York, NY, 94–104. DOI : <https://doi.org/10.1145/1513895.1513907>
- [9] Jack Dongarra, Pete Beckman, Terry Moore, Patrick Aerts, Giovanni Aloisio, Jean-Claude Andre, David Barkai, Jean-Yves Berthou, Taisuke Boku, Bertrand Braunschweig, Franck Cappello, Barbara Chapman, Xuebin Chi, Alok Choudhary, Sudip Dosanjh, Thom Dunning, Sandro Fiore, Al Geist, Bill Gropp, Robert Harrison, Mark Hereld, Michael Heroux, Adolfo Hoisie, Koh Hotta, Zhong Jin, Yutaka Ishikawa, Fred Johnson, Sanjay Kale, Richard Kenway, David

- Keyes, Bill Kramer, Jesus Labarta, Alain Lichnewsky, Thomas Lippert, Bob Lucas, Barney Maccabe, Satoshi Matsuoka, Paul Messina, Peter Michielse, Bernd Mohr, Matthias S. Mueller, Wolfgang E. Nagel, Hiroshi Nakashima, Michael E. Papka, Dan Reed, Mitsuhsa Sato, Ed Seidel, John Shalf, David Skinner, Marc Snir, Thomas Sterling, Rick Stevens, Fred Streitz, Bob Sugar, Shinji Sumimoto, William Tang, John Taylor, Rajeev Thakur, Anne Trefethen, Mateo Valero, Aad Van Der Steen, Jeffrey Vetter, Peg Williams, Robert Wisniewski, and Kathy Yelick. 2011. The international exascale software project roadmap. *International Journal of High Performance Computing Applications* 25, 1 (Feb. 2011), 3–60. DOI: <https://doi.org/10.1177/1094342010391989>
- [10] Shuguang Feng, Shantanu Gupta, Amin Ansari, and Scott Mahlke. 2010. Shoestring: Probabilistic soft error reliability on the cheap. In *Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems (ASPLOS XV)*. ACM, New York, NY, 385–396. DOI: <https://doi.org/10.1145/1736020.1736063>
- [11] Rohan Garg, Apoorve Mohan, Michael Sullivan, and Gene Cooperman. 2018. CRUM: Checkpoint-restart support for CUDA’s unified memory. In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster’18)*.
- [12] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. 2013. Accelerating financial applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units (GPGPU-6)*. ACM, New York, NY, 127–136. DOI: <https://doi.org/10.1145/2458523.2458536>
- [13] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta. 2017. Compiler techniques to reduce the synchronization overhead of GPU redundant multithreading. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 1–6. DOI: <https://doi.org/10.1145/3061639.3062212>
- [14] H. Jeon and M. Annavam. 2012. Warped-DMR: Light-weight error detection for GPGPU. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 37–47. DOI: <https://doi.org/10.1109/MICRO.2012.13>
- [15] Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, and Dimitris Gizopoulos. 2015. Accelerated microarchitectural fault injection-based reliability assessment. DOI: <https://doi.org/10.1109/DFT.2015.7315134>
- [16] Charu Kalra. 2015. *MS Thesis: Design and Evaluation of Register Allocation on GPUs*. Master’s Thesis. Northeastern University.
- [17] Charu Kalra. 2019. *Compiler-based Resilience Prediction and Enhancements for GPU Applications*. Ph.D. Dissertation. Northeastern University.
- [18] Charu Kalra, Daniel Lowell, John Kalamatianos, Vilas Sridharan, and David Kaeli. 2016. Performance evaluation of compiler-based software rmt in an hsa environment. In *The 12th Workshop on Silicon Errors in Logic-System Effects, SELSE*.
- [19] Charu Kalra, Fritz Previlon, Xiangyu Li, Norman Rubin, and David Kaeli. 2018. Analyzing the vulnerability of vector-scalar execution on data-parallel architectures. In *The 14th Workshop on Silicon Errors in Logic - System Effects, SELSE*.
- [20] Charu Kalra, Fritz Previlon, Xiangyu Li, Norman Rubin, and David Kaeli. 2018. PRISM: Predicting resilience of GPU applications using statistical methods. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC’18)*. IEEE Press, Piscataway, NJ, Article 69, 14 pages. <http://dl.acm.org/citation.cfm?id=3291656.3291748>
- [21] Andrew Kerr, Gregory Diamos, and Sudhakar Yalamanchili. 2009. A characterization and analysis of PTX kernels. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC’09)*. IEEE Computer Society, Washington, DC, 3–12. DOI: <https://doi.org/10.1109/IISWC.2009.5306801>
- [22] Junsung Kim, Hyoseung Kim, Karthik Lakshmanan, and Ragunathan Rajkumar. 2013. Parallel scheduling for cyber-physical systems: Analysis and case study on a self-driving car. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems (ICCPs)*.
- [23] R. Koo and S. Toueg. 1987. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering* SE-13, 1 (Jan 1987), 23–31. DOI: <https://doi.org/10.1109/TSE.1987.232562>
- [24] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO’04)*. IEEE, 75–86.
- [25] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. 2009. Statistical fault injection: Quantified error and confidence. In *Design, Automation Test in Europe Conference Exhibition (DATE’09)*. 502–506. DOI: <https://doi.org/10.1109/DATE.2009.5090716>
- [26] G. Li, K. Pattabiraman, C. Y. Cher, and P. Bose. 2016. Understanding error propagation in GPGPU applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 240–251. DOI: <https://doi.org/10.1109/SC.2016.20>
- [27] Guanpeng Li, Karthik Pattabiraman, Siva Kumar Sastry Hari, Michael Sullivan, and Timothy Tsai. 2018. Modeling soft-error propagation in programs. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE.
- [28] Robert Lucas, James Ang, Keren Bergman, and Shekhar et al. Borkar. 2014. Top Ten Exascale Research Challenges. <http://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20140210/Top10reportFEB14.pdf>.

- [29] Abdulrahman Mahmoud, Siva Hari, Mike Sullivan, Timothy Tsai, and Steve Keckler. 2018. Optimizing software-directed instruction replication for GPU error detection. In *International Conference for High-Performance Computing, Networking, Storage and Analysis (SC)*.
- [30] Jose Maiz, Scott Hareland, Kevin Zhang, and Patrick Armstrong. 2003. Characterization of multi-bit soft error events in advanced SRAMs. In *IEEE International Technical Digest Electron Devices Meeting (IEDM'03)*. IEEE, 21–4.
- [31] Mei-Chen Hsueh, T. K. Tsai, and R. K. Iyer. 1997. Fault injection techniques and tools. *Computer* 30, 4, 75–82. DOI: <https://doi.org/10.1109/2.585157>
- [32] Harshitha Menon, Michael O. Lam, Daniel Osei-Kuffuor, Markus Schordan, Scott Lloyd, Kathryn Mohror, and Jeffrey Hittinger. 2018. ADAPT: Algorithmic differentiation applied to floating-point precision tuning. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*. IEEE Press, Piscataway, NJ, Article 48, 13 pages. <http://dl.acm.org/citation.cfm?id=3291656.3291720>
- [33] Shubu Mukherjee. 2011. *Architecture Design for Soft Errors*. Morgan Kaufmann.
- [34] Bin Nie, Lishan Yang, Adwait Jog, and Evgenia Smirni. 2018. Fault site pruning for practical reliability analysis of gpgpu applications. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 749–761.
- [35] NVIDIA. [n.d.]. CUDA binary utilities. Retrieved from <http://docs.nvidia.com/cuda/pdf/CUDA-Binary-Utilities.pdf>.
- [36] NVIDIA. [n.d.]. NVCC, <https://developer.nvidia.com/cuda-llvm-compiler>.
- [37] NVIDIA. [n.d.]. NVIDIA, CUDA SDK, V6.0.
- [38] NVIDIA. [n.d.]. NVIDIA Kepler GK110 architecture white paper.
- [39] NVIDIA. [n.d.]. Tegra K1 technical reference manual.
- [40] Hamza Omar, Qingchuan Shi, Masab Ahmad, Halit Dogan, and Omer Khan. 2018. Declarative resilience: A holistic soft-error resilient multicore architecture that trades off program accuracy for efficiency. *ACM Transactions on Embedded Computing Systems* 17, 4, Article 76 (July 2018), 27 pages. DOI: <https://doi.org/10.1145/3210559>
- [41] Fritz G. Previlon, Babatunde Egbantant, Devesh Tiwari, Paolo Rech, and David R. Kaeli. 2017. Combining architectural fault-injection and neutron beam testing approaches toward better understanding of GPU soft-error resilience. In *2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 898–901.
- [42] Fritz G. Previlon, Charu Kalra, David R. Kaeli, and Paolo Rech. 2019. A comprehensive evaluation of the effects of input data on the resilience of GPU applications. In *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. IEEE, 1–6.
- [43] Fritz G. Previlon, Charu Kalra, Devesh Tiwari, and David R. Kaeli. 2019. PCFI: Program counter guided fault injection for accelerating GPU reliability assessment. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*. 308–311. DOI: <https://doi.org/10.23919/DAT.2019.8714781>
- [44] Sandra Rapps and Elaine J. Weyuker. 1982. Data flow analysis techniques for test data selection. In *Proceedings of the 6th International Conference on Software Engineering (ICSE'82)*. IEEE Computer Society Press, Los Alamitos, CA, 272–278. <http://dl.acm.org/citation.cfm?id=800254.807769>
- [45] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. 2005. SWIFT: Software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*. 243–254. DOI: <https://doi.org/10.1109/CGO.2005.34>
- [46] Siva Kumar Sastry Hari, Timothy Tsai, Mark Stephenson, Steve Keckler, and Joel Emer. 2017. SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 249–258. DOI: <https://doi.org/10.1109/ISPASS.2017.7975296>
- [47] Michael C. Schatz, Cole Trapnell, Arthur L. Delcher, and Amitabh Varshney. 2007. High-throughput sequence alignment using graphics processing units. *BMC Bioinformatics* 8, 1 (2007), 1–10. DOI: <https://doi.org/10.1186/1471-2105-8-474>
- [48] Mark Stephenson, Siva Kumar Sastry Hari, Yunsup Lee, Eiman Ebrahimi, Daniel R. Johnson, David Nellans, Mike O'Connor, and Stephen W. Keckler. 2015. Flexible software profiling of GPU architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15)*. ACM, New York, NY, 185–197.
- [49] J. E. Stone, J. C. Philips, P. L. Freddolino, D. J. Hardy, L. G. Trabuco, and K. Schulten. 2007. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry* 28 (2007), 2618–2640.
- [50] Michael B. Sullivan, Siva Kumar Sastry Hari, Brian Zimmer, Timothy Tsai, and Stephen W. Keckler. 2018. Swap-Codes: Error codes for hardware-software cooperative GPU pipeline error detection. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 762–774.
- [51] Jingweijia Tan and Xin Fu. 2012. RISE: Improving the streaming processors reliability against soft errors in Gpgpus. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, NY, 191–200. DOI: <https://doi.org/10.1145/2370816.2370846>
- [52] Yash Ukidave, Fanny Nina Paravecino, Leiming Yu, Charu Kalra, Amir Momeni, Zhongliang Chen, Nick Materise, Brett Daley, Perhaad Mistry, and David Kaeli. 2015. NUPAR: A benchmark suite for modern GPU architectures. In

Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE'15). ACM, New York, NY, 253–264. DOI : <https://doi.org/10.1145/2668930.2688046>

- [53] Dani Voitsechov, Arslan Zulfiqar, Mark Stephenson, Mark Gebhart, and Stephen W. Keckler. 2018. Software-directed techniques for improved GPU register file utilization. *ACM Transactions on Architecture and Code Optimization (TACO)* 15, 3, 38.
- [54] Jack Wadden, Alexander Lyashevsky, Sudhanva Gurumurthi, Vilas Sridharan, and Kevin Skadron. 2014. Real-world design and evaluation of compiler-managed GPU redundant multithreading. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*. IEEE Press, 73–84.

Received July 2019; revised December 2019; accepted February 2020