Optimizing Software-Directed Instruction Replication for GPU Error Detection

Abdulrahman Mahmoud[†], Siva Kumar Sastry Hari[‡], Michael B. Sullivan[‡], Timothy Tsai[‡], Stephen W. Keckler[‡]
[†]University of Illinois at Urbana-Champaign [‡]NVIDIA
amahmou2@illinois.edu, {shari, misullivan, timothyt, skeckler}@nvidia.com

***Abstract*—Application execution on safety-critical and high-performance computer systems must be resilient to transient errors. As GPUs become more pervasive in such systems, they must supplement ECC/parity for major storage structures with reliability techniques that cover more of the GPU hardware logic. Instruction duplication has been explored for CPU resilience; however, it has never been studied in the context of GPUs, and it is unclear whether the performance and design choices it presents make it a feasible GPU solution. This paper describes a practical methodology to employ instruction duplication for GPUs and identifies implementation challenges that can incur high overheads (69% on average). It explores GPU-specific software optimizations that trade fine-grained recoverability for performance. It also proposes simple ISA extensions with limited hardware changes and area costs to further improve performance, cutting the runtime overheads by more than half to an average of 30%.**

***Index Terms*—Reliability, Fault tolerance, Redundancy, Software protection, High performance computing**

I. Introduction

Safety-critical and high-performance computer systems must be highly reliable despite various sources of hardware errors. Transient hardware errors from high-energy particle strikes (also known as soft-errors) are of particular concern due to their risk of silent data corruption (SDC) [1], [2]. As GPUs become more pervasive in safety-critical (e.g., autonomous driving systems) and high-performance computing (HPC) systems, designers must ensure that the computations that are offloaded to them are resilient to transient errors. State-of-the-art GPUs employ ECC or parity protection for major storage structures such as DRAM, caches, and the register file [3], [4], [5], [6]. However, prior work indicates that datapath errors originating from unprotected latches scattered across the processors will contribute significantly to the total SDC rate [1], [7]. Without datapath reliability mechanisms, such systems may not be able to maintain high reliability at future error rates and system scales.

Traditional hardware-only solutions that duplicate the entire processor can provide datapath reliability [8], [9]. However, processor duplication is expensive and excessive for workloads or sections of code that are inherently resilient. Software-based redundancy overcomes these issues and can provide the flexibility of protecting just the vulnerable parts of the program without incurring the high design, debug, and testing costs attributed to hardware-only schemes. For GPUs, software-based redundancy can be introduced at various granularities such as the process, kernel, thread, and assembly instruction level. Of the four, only instruction-level duplication can be applied seamlessly to workloads that produce non-deterministic results at a coarse granularity (e.g., at the function, GPU kernel, or application output level) without requiring spare hardware resources to be reserved solely for redundancy purposes. Higher level duplication techniques often suffer from limitations in one of these aspects. We discuss these trade-offs in detail in Section II-B.

In this work, we target software-directed instruction replication for GPU error detection. Software instruction-level duplication has been studied extensively for CPUs and has been shown to provide runtime overheads that are significantly lower than 100% by exploiting under-utilized hardware resources (approximately 60% using a 4-way issue super-scalar processor and 40% for Intel Itanium CPUs) [10], [11], [12]. Duplication at this level has never been explored for GPUs. Prior research has shown that many GPU workloads under-utilize GPU cores [13], [14], indicating potential for a low-overhead instruction-level duplication solution.

This paper introduces SInRG (pronounced "synergy"), **S**oftware-managed **In**struction **R**eplication for **G**PUs, which is a family of techniques that optimize software-based instruction duplication for GPUs. Our work is the first to establish a practical approach to software-directed instruction duplication for GPU-based systems, identify GPU-specific opportunities for overhead reduction, and explore software and hardware performance optimizations to lower the overheads significantly.

SInRG first implements a commonly-studied CPU instruction duplication algorithm in NVIDIA's production compiler and evaluates it on real GPUs. This algorithm duplicates the data-flow chains leading to non-duplicated instructions and maintains two register spaces such that the original and duplicate instructions operate on the original and shadow register spaces, respectively [11]. Whenever a non-duplicated instruction is executed, the source register values are verified, and a higher layer in the system is notified if verification fails. Our results show an average runtime overhead of 69%.

Main contributors to the overhead stem from two sources. (1) Doubling the number of required registers per thread can adversely affect performance for some workloads because the register file is a shared resource and its inefficient use can limit the number of concurrent threads (and performance). (2) The total number of executed instructions increase with the introduction of additional verification and notification instructions. These new instructions also introduce new dependencies, which can limit the ability to software-pipeline instructions, further increasing runtime overheads.

SC18, November 11-16, 2018, Dallas, Texas, USA
978-1-5386-8384-2/18/$31.00

SInRG addresses the first issue by employing an instruction duplication algorithm that trades off the per-thread register requirement for more verification and notification instructions. We propose a set of solutions to reduce the overheads caused by executing verification and notification instructions. (1) SInRG removes direct dependencies between the verification and notification instructions, using a per-thread flag to defer error notification to the end of a thread. It trades off error containment for performance, which can be mitigated through a small ISA extension that provides error notification capability to the verification instructions. (2) We discover that the verification and deferred notification can be implemented using just a single, high-throughput assembly instruction supported by current GPUs. (3) With the aim of eliminating the verification and notification instructions altogether, SInRG accelerates the first solution above through hardware support. Some of these solutions defer error detection until the end of the kernel, which may affect fine-grain recoverability. The increase in detection latency is however not a concern for coarse grain coordinated checkpointing solutions [15], [16], [17], [18].

Results show that SInRG reduces the average runtime overhead to 36% (1.94x lower than the naïve implementation) with software-only techniques. Simple hardware extensions that eliminate verification and notification instructions reduce the average overhead to 30%. We compare SInRG to a prior state-of-the-art GPU software-based approach – thread-level duplication (TLD) [19], [20], [21] – and further optimize it for comparison. Results show that SInRG is faster than TLD for a majority of the workloads studied despite it not needing/utilizing spare thread resources.

We evaluate the error detection capabilities of SInRG by quantifying the dynamic instruction coverage (counting the executed instructions that are protected by SInRG). Results show that on average, 87% of dynamic instructions are covered. We also conduct architecture-level error injection experiments to show that SInRG is effective in reducing SDCs. Results show that the percentage of injected errors that result in SDCs is always lower than the percentage of uncovered dynamic instructions. Lastly, we evaluate the effect on the true failure rate (measured as Failure In Time or FIT, where 1 FIT = 1 failure in 1 billion hours of operation) reduction by conducting accelerated high-energy particle testing. Results show that SInRG can reduce the SDC FIT rate by an order of magnitude.

II. Related Work and Challenges

A. GPU Background

We review basic GPU architecture terminology and the NVIDIA GPU compilation flow because we implement SInRG using NVIDIA's technology. However, the ideas presented in this paper can be applied to other GPU architectures.

GPU programming models consider thousands of threads that each execute the same code. Threads are grouped into 32-element vectors called *warps* to improve efficiency. The threads in each warp execute in a SIMT (single instruction, multiple thread) fashion. Many warps are assigned to execute concurrently on a single GPU streaming multiprocessor (SM). An SM offers resources that are shared by all the executing threads, such as the register file and shared memory (or scratchpad). A GPU consists of many SMs attached to a memory hierarchy that includes SM-local scratchpad memories, L1 caches, a shared L2 cache, and multiple DRAM channels.

A user can write parallel programs using high-level programming languages such as CUDA [22] or OpenCL [23], and use a front-end compiler to generate intermediate code in a virtual ISA called parallel thread execution (PTX) [24]. A backend compiler optimizes and translates PTX instructions into machine code that can run on the device. NVIDIA's native ISA is called SASS [25]. The backend compiler can be invoked in two ways: (1) ahead-of-time compilation of compute kernels via a PTX assembler (ptxas) or (2) just-in-time compilation by the GPU driver (if the PTX code is part of the binary).

B. Related Work

Software introduced redundancy: Prior techniques have introduced redundancy at multiple granularities including the process, GPU kernel, thread, and assembly instruction level. Process-level redundancy replicates the process and compares results at system call boundaries [26], [27]. This approach suffers from limitations for multi-threaded workloads. Kernels or thread blocks can be re-executed and their outputs then compared to ensure correctness [19]. This approach is challenging for workloads where the kernel or block outputs are non-deterministic, which can arise from rounding errors and reading clock values during execution. Thread-level duplication (TLD) has been employed for CPUs [28], [29], [27], [30] and GPUs [19], [20], [21] and requires spare hardware resources. Wadden et al. [20] and Gupta et al. [21] each proposed a compiler-based approach for GPUs that duplicates thread-blocks and threads, and observed high overheads for block-level duplication due to inter-block communication and synchronization. We quantitatively compare SInRG to TLD in this paper. One drawback for TLD is that programmer intervention may be required to ensure spare hardware resources are available. Intra-warp communication constructs such as warp vote and shuffle operations, for example, must be handled accordingly for proper TLD operation.

Software instruction-level duplication does not have these limitations and has been explored for CPUs [10], [11], [12]. Oh et al. [10] proposed a technique to duplicate instructions at the assembly level and insert checking instructions to validate the results. The average runtime overheads were approximately 60% on a 4-way issue superscalar processor. SWIFT [11] proposed a compiler-based approach and exploited wide, underutilized processors by scheduling both original and duplicated instructions in the same execution thread, and reported overheads of about 40% on Intel Itanium CPUs. The applicability of such techniques for GPUs has not been studied previously, and SInRG addresses this gap. To reduce the overheads further, Shoestring developed a compiler technique to selectively duplicate instructions by trading off coverage for performance [31]. Combining such a technique with SInRG is an interesting future direction.

Hardware introduced redundancy: Traditional business-class systems (e.g., IBM Z Series machines [8]) employ expensive hardware-managed dual- or triple-modular redundancy schemes at prohibitively high cost for commercial use. In safety-critical systems, similar techniques are being employed to meet the safety integrity requirements [9]. Recent server and business-class processors (e.g., IBM System Z machines [32]) adopt fine-grained hardware checkers to detect errors in individual processor components, presumably with substantive design effort. Such an approach has also been explored for GPUs [33].

Warped-DMR and RISE proposed hardware mechanisms to exploit underutilized parallelism in GPUs for error detection [7], [34]. Specifically, Warped-DMR uses the idle SIMT lanes to redundantly execute some of the threads within the warp and achieve intra-warp DMR execution. RISE proposed mechanisms to predict and use idle SM cycles and SIMT lanes to execute redundant work [34]. Warped-RE extended these approaches and introduced redundancy to verify every warp instruction [35]. It re-executes the instruction to correct any detected errors. Each of these techniques requires complex hardware changes and they are not directly comparable to SInRG techniques.

C. Challenges with GPU Instruction Duplication

Overheads of an instruction duplication algorithm arise from the introduction of the three types of instructions: redundant, verification, and notification instructions. Prior optimizations target leveraging under-utilized resources and reduce the number of the added instructions. GPUs, however, present several new challenges in developing a cost-effective solution.

Limited shared resources: Since SMs provide resources that are shared among all the executing threads, inefficient per-thread usage of these shared resources may limit the number of warps that can simultaneously run on the SM (also known as warp occupancy). The register file is one such resource; doubling the per-thread register requirement can limit the warp occupancy and increase the overall runtime.

Additional dependencies: The verification and notification instructions added by an instruction duplication algorithm introduce read-after-write dependencies between themselves, which may limit ILP and the instruction scheduler's ability to pipeline instructions. Such limitations can significantly increase the runtime overheads if the workload is not capable of executing enough concurrent threads. Prior research has also noted the importance of ILP for GPUs [36].

Extra instructions: Verification and notification instructions increase the dynamic instruction count. Moreover, throughput offered by the assembly instructions used for them can be low. For example, compares have half of the maximum throughput offered by some instructions on NVIDIA GPUs [22], [25].

III. SInRG Overview

A. Sphere of Replication (SoR)

GPUs used in HPC and safety-critical systems protect major memory structures such as DRAM, caches, and the register file using ECC/parity. However, unprotected execution units and pipeline registers remain susceptible to soft-errors. Since GPUs are designed to maintain high throughput for arithmetic intensive workloads, the datapaths constitute a significant fraction of the chip area, unlike CPUs which devote most of their non-cache logic to instruction delivery and control speculation.

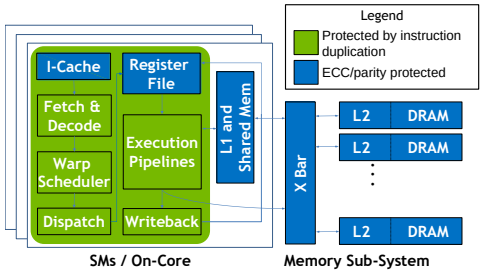


Fig. 1: The GPU hardware structures in the SoR. We focus on protecting the GPU datapath against transient errors.

To protect the execution units and pipeline stages, we employ assembly instruction-level duplication without duplicating values in memory. We use the term sphere-of-replication (SoR) to identify at a high-level what is duplicated and which hardware structures we expect to be protected by this approach. This technique can detect errors that affect program text (instructions) and computation (instruction execution). Since not all instructions are duplicated (e.g., branch instructions and atomic operations remain unduplicated), the coverage is high but not complete. We quantify coverage in later sections.

Figure 1 shows the hardware structures that SInRG protects. Almost all of the SM units used to execute an instruction (from instruction fetch to write-back) receive protection from single-event errors. SInRG also delivers additional protection to some of the structures that are protected by hardware ECC/parity (e.g., I-cache for all SInRG schemes and register file for the schemes that duplicate registers). This additional protection comes for free.

B. Instruction Duplication Algorithms

SInRG duplicates all instructions that (1) produce deterministic values, (2) do not directly modify the control flow, and (3) do not write to memory. We call such instructions *duplication eligible*.

SInRG performs duplication using two main base algorithms. The first algorithm, inspired by Reis et al. [11], duplicates all the instructions in a data-flow chain leading to a non-duplicated instruction and verifies the values only at the end of the chain. Duplicated instructions operate on a *virtual shadow register space*. For instructions that are not duplication eligible and write to a register (e.g., atomic operations and special registers), we copy the result of the original instruction to the shadow register space to maintain the functionality of shadow execution. We call this algorithm *DRDV*, because it **d**oubles the virtual **r**egister space and **d**elays **v**erification until the end of a data-flow chain.

The second base algorithm duplicates all duplication eligible instructions and places them just before the original instruction. The duplicated instruction reads the same source registers used by the original instruction and writes to a new virtual register. We immediately verify the value in the new register with

the destination register value of the original instruction, and notify the runtime layer for appropriate event handling if the verification fails. This scheme adds verification and notification instructions for every duplication eligible instruction, increasing the total number of dynamic instructions significantly (and hence is often ignored by CPU implementations). We call this algorithm *SRIV* because it uses a **s**ingle **r**egister space and **i**mmediately **v**erifies each instruction's result.

C. SInRG Optimizations

This section presents techniques we propose to address the challenges mentioned in Section II-C. Table I summarizes the trade-offs offered by these optimizations. Each SInRG duplication technique is listed with a qualitative comparison of its attributes (relative to an uninstrumented workload), which are discussed in more detail below.

TABLE I: Summary of the SInRG techniques

		Attributes			
		✓= low, O = medium, ✗= high		✓= yes, ✗= no	
	SInRG Technique	# Verification Instructions	Register requirement per thread	Error Containment	Error Masking Potential
DRDV	Base	✗✗	✗	✓	✓
	FastSig	✓	✗	✗	✓
	HW-Notify	✓	✗	✓	✓
	HW-Sig	✓✓	✗	✗	✗
SRIV	Base	✗✗	✗	✓	✗
	FastSig	O	✓	✗	✗
	HW-Notify	O	✓	✓	✗
	HW-Sig	✓✓	✓	✗	✗

Paper Abbreviations:

DRDV:	Double register space, delayed verification
SRIV:	Single register space, immediate verification
FastSig:	Software-only, fast signature-based checking
HW-Notify:	Hardware instruction to compare-then-trap
HW-Sig:	Signature-based checking in hardware
TLD:	Thread Level Duplication [20]
TLD-Sig:	TLD [20] with delayed notification using signatures

Trading off additional dynamic instructions to reduce register requirements: The DRDV algorithm doubles the virtual registers required per thread. Running the NVIDIA compiler's production-quality register allocator after the instruction duplication pass can reduce the real register usage per thread. Despite this optimization, DRDV often observes a significant increase in the number of registers used per thread. For workloads where the register file is a critical resource, this approach can either reduce the number of threads that can run in parallel or increase the number of register spill/fill instructions. The SRIV algorithm naturally provides an interesting trade-off because it does not alter the original application's register requirement by much, but instead executes more dynamic instructions. This trade-off can benefit some workloads, especially when the register file is a critical resource, which we analyze in more detail in Section VI.

Deferring error notification: The code to notify the upper layers of the system (e.g., a trap instruction and the control flow instructions to skip the trap in fault-free executions) is typically added after every verification instruction for error containment. The added dependency between the two instructions can contribute significantly to performance overheads, as mentioned in Section II-C. To reduce the overheads, we investigate deferring the notification until the end of the function. The results of all verification instructions are accumulated to produce a single flag (signature), which is then used by a single error notification instruction at the end of the function. Similar approaches have been explored and shown to be effective in the context of software testing [37], [38], [39]. This optimization drastically reduces the number of error notifications (and associated control flow instructions) and enables better instruction scheduling. However, it allows some erroneous values to propagate to memory before the error is detected and notified. While this optimization may violate the error containment assumptions of some recovery schemes, it works fine for coarse-grain coordinated checkpoint systems that discard memory values in the event of a detected error to roll back to a previous checkpoint [15], [16], [17], [18].

Verification and accumulation of the result must be implemented efficiently for a low overhead solution. This can be accomplished using one or two high-throughput assembly instructions on current GPUs. This approach also addresses the third challenge mentioned in Section II-C and explained in Section IV-C. We call this combined software optimization *FastSig* and it applies to both the DRDV and SRIV algorithms.

Eliminating notification instructions: To eliminate explicit error notification instructions and provide high error containment, we propose a simple extension to an existing GPU instruction that is used to compare two values. This extension raises an exception in hardware if the values mismatch. We call this technique *HW-Notify*.

Eliminating verification and notification instructions: We eliminate the verification and notification instructions by proposing *HW-Sig*, which uses the same principles as FastSig and provides hardware support to maintain the signature register. This register is initialized at kernel launch time. It is updated by each of the original duplication-eligible and duplicate instructions such that it will have the same initialized value at the end of a fault-free kernel execution. Maintaining one register per thread can be expensive in GPUs because SMs support thousands of threads. We overcome this challenge by maintaining just one signature register per hardware lane, which would be used by all threads (from different warps) that execute on the lane. HW-Sig improves performance without sacrificing error coverage. Since it defers the error notification, similar to FastSig, the trade-offs are also similar.

We also considered a scheme that extends the ISA such that each duplicate instruction automatically verifies the result produced by the original instruction and notifies upper layers upon failure. The source operands of the original instruction should not be updated before the duplicate instruction executes, which introduces new instruction scheduling constraints. As our evaluation showed lackluster performance, we do not discuss it further in this paper.

IV. Implementation

A. GPU Compilation Flow

GPU instruction duplication can be implemented at several places in the compiler tool chain. While performing it early in

the flow before PTX code generation is perhaps easiest to implement, later compiler optimization passes may transform the program and eliminate the resilience-oriented instructions. Inserting the replicated and checking instructions directly into the SASS code ensures tight control over the final program binary, but requires re-implementation of instruction scheduling and register allocation which are already in the back-end compiler.

Avoiding these limitations, we implement SInRG within the back-end compiler (ptxas), applying our transformations on the intermediate representation there. The duplication algorithm runs after all back-end optimizations are performed, but before the instruction scheduling or register allocation passes. This approach leverages the production-quality instruction scheduler already implemented in the back-end compiler, which helps to lower the performance overheads of the duplication and verification code. It also enables instruction duplication on programs for which only the PTX code (rather than the CUDA or OpenCL source code) is available. Figure 2 summarizes the compilation flow for NVIDIA GPU programs, including the SInRG instruction duplication pass. This paper evaluates SInRG using the ahead-of-time compilation flow, but the just-in-time compiler can employ the same instruction duplication algorithms.

B. Instruction Duplication Compiler Pass

SInRG duplicates each duplication-eligible instruction once. Instructions that are *not* eligible for duplication include memory writes, control-flow instructions, instructions that produce non-deterministic values, barrier spill/fill instructions, and instructions that write to pre-assigned physical registers. Non-deterministic instructions—those where the replica and the original instruction can produce different values—include S2R instructions that read special registers whose values change over time (e.g., the clock value), atomic operations, and volatile and non-cached memory reads [25]. A load can be non-deterministic if there is a data race in the program. While we would ideally only mark the race-vulnerable loads as non-deterministic, identifying only this subset of loads is not feasible. Instead, we conservatively mark all generic, global, shared, texture, and surface loads as non-deterministic. We mark local and constant loads as deterministic because they cannot partake in data races. Local memory offers per thread storage (which cannot be accessed by other threads) and constant memory is read-only (and cannot be written to). We do not apply SInRG passes to built-in CUDA Runtime API calls.

For DRDV, verifying the inputs of a non-duplicated instruction requires adding a set of verification and notification instructions, one for each source operand. We implement an optimization where we insert only one error notification instruction per non-duplicated instruction (as opposed to one per source operand) by chaining multiple verification instructions.

For SRIV, we place the duplicate *before* the original instruction because the original instruction may overwrite a source operand, and we want the duplicate to generate the same result as the original instruction using the same source operands. We do not duplicate the original move operations because they naturally duplicate the source register value

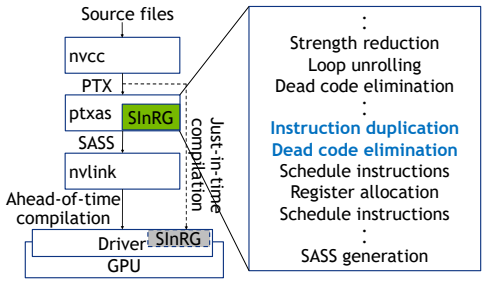


Fig. 2: The GPU compiler flow with SInRG.

into the destination register. We verify them by comparing the source and destination registers of the original operation. Verification and notification consist of a comparison operation, a conditional branch instruction, and a trap instruction (`BPT`).

Figures 3(1) and 3(2) show an example of how we duplicate and verify an add instruction using SRIV. Base verification includes two additional instructions in the critical path for each duplicated instruction and creates sequential dependencies which can affect performance. The branch and trap instructions also limit the instruction scheduler, which does not efficiently schedule instructions across trap instructions or basic blocks.

C. SInRG Optimizations

FastSig: This technique accumulates the results of the verification instructions into a signature register and uses it at the end of the function for deferred error notification. This signature (flag) register is initialized to zero at the beginning of a function. On every register verification, the values produced by the original and duplicate instructions are added to and subtracted from the signature register, respectively. If the signature register is not equal to zero at the end of the function, an error has occurred.

Using simple add and subtract operations may miss some errors due to over/under-flow. Instead, we compute bit-wise difference between the destination registers of the original and duplicate instructions using *XOR*, and then *OR* the result with the signature register to update it. During a fault-free execution, the signature register will remain zero. We discovered that the `LOP3` operation supported by the current NVIDIA GPUs can create any arbitrary logical function using three source operands and is well suited for the signature accumulation [24], [25]. Moreover, it offers the highest throughput among all supported instructions. We maintain a separate predicate signature register and use the `PSETP` instruction to perform a similar accumulation operation in one instruction.

Figure 3(3) shows an example of how this optimization reduces the number of static verification instructions from three to one. Furthermore, this optimization allows us to predicate the verification instructions if the original store instruction is predicated, providing added benefit. In the base approach, the verification (`ISETP`) instruction cannot be predicated because it must generate a correct predicate register for the subsequent branch instruction.

Since FastSig relaxes error containment, an error can propagate to memory and in some cases result in a crash/hang before the notification instruction is executed. If a function

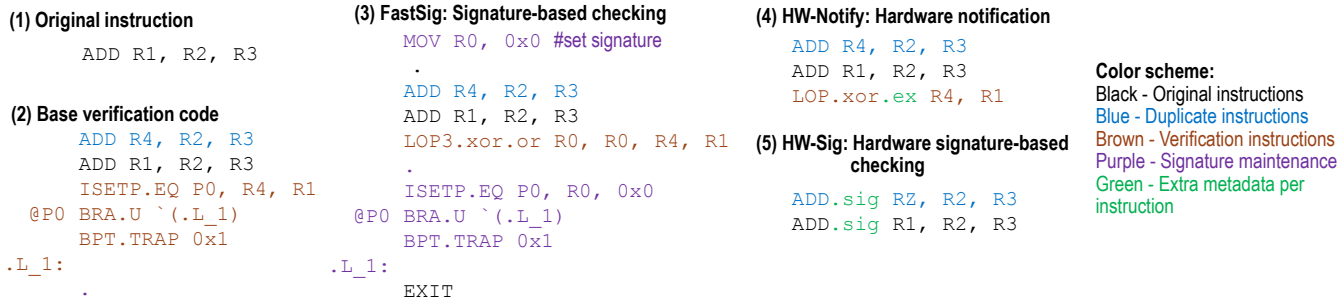
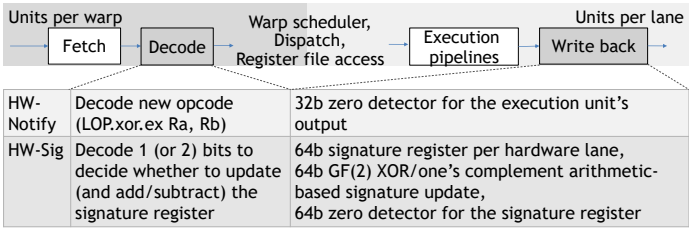


Fig. 3: Optimizing the verification and error notification code.



HW-Notify	Decode new opcode (LOP.xor.ex Ra, Rb)	32b zero detector for the execution unit's output
HW-Sig	Decode 1 (or 2) bits to decide whether to update (and add/subtract) the signature register	64b signature register per hardware lane, 64b GF(2) XOR/one's complement arithmetic-based signature update, 64b zero detector for the signature register

Fig. 4: Summary of the two hardware techniques.

has many conditional return instructions, the number of error notification instructions will also be high, increasing overheads.

HW-Notify: We propose a new branch-free instruction that compares two values and raises an exception on a mismatch to provide low-latency error detection with full error containment. This instruction replaces the signature update operation used by FastSig and avoids the need to maintain a signature register. It is similar to a logical (`LOP`) operation except that it does not need a destination register. Hardware changes, as summarized in Figure 4, include instruction decoder support for the new operation and some logic in the register write-back stage to raise an exception based on the results of a bit-wise equality check. Since current GPUs support exception reporting and handling [40], HW-Notify can leverage this existing framework for traps. Figure 3(4) illustrates how the instruction is used.

HW-Sig: This technique eliminates all verification and notification instructions. As mentioned in Section III-C, maintaining one register per thread requires significant on-chip storage (10's of kB/SM) because an SM supports thousands of threads. We propose using just one signature register per hardware lane (not per thread) per context. Since instructions can write to one or two 32-bit registers, we propose using a 64b signature register. We initialize the signature register to zero at the kernel launch time (using a synchronous reset signal) and ensure that it is zero at the end of the kernel. As each instruction executes, it updates the signature register by adding or subtracting its destination register values based on whether the instruction is original or duplicate, respectively. Operations that are commutative, easy to design in hardware, and require low area overhead are good candidates for signature updates. For example, binary Galois Field arithmetic (GF(2)) that uses *XOR* operations can be used for signature accumulation and subtraction [41]. We need one extra meta-data bit in the instruction to indicate whether the signature register should be updated by the instruction. When HW-Sig is employed with SRIV, the duplicate instruction only updates the signature register; its destination is replaced with *RZ*. The signature update logic need not be in the critical path and can be performed in parallel with the write back stage while the result of the instruction is being written back to the register file. Figure 4 summarizes the hardware changes.

At the end of the kernel, we activate the register checking logic using a global signal. If the value is non-zero, an exception is raised. This approach trades off the ability to detect the error until the end of the kernel and diagnose which thread is corrupted, which is not a concern for existing coarse grained checkpointing solutions [15], [16], [17], [18].

Storage overhead can be reduced by accumulating the ECC bits of each result, instead of the result itself. Hence the signature register only needs to be as wide as the error code (e.g., 7b SEC-DED is used for the 32b GPU registers [3], [5]). The signature update can take place in a pipeline stage following ECC encoding without performance concerns because this logic is not in the critical path of the datapath.

A hardware error can be missed if (1) both the original and duplicate instructions see the same corruption in their results, which is not possible for single instruction error model, or if (2) a single error propagates to an even number of instructions in the same thread and these affected instructions update the signature register such that the observed errors happen to cancel out. The second scenario is impossible for SRIV (duplicate instructions do not write to registers), and highly improbable for DRDV because the conditions are challenging to meet. The likelihood of this scenario can be reduced by choosing a signature update function that is less likely to cancel errors (e.g., one's complement add as opposed to GF(2) XOR).

V. Evaluation Methodology

Our experimental flow targets NVIDIA Pascal (Titan-Xp) architecture-based GPUs with Compute Capability 6.1 [5]. We modify NVIDIA's production back-end compiler and use it with the CUDA 8.0 toolkit. The host system has an Intel i7-3930K CPU (3.2GHz) and 32GB of system memory. We evaluate SInRG using 16 workloads, 15 of which are from the Rodinia benchmark suite (version 3.0) [42]. The last workload is the matrix multiplication program (referred as *mm* in this paper) provided as a sample in the CUDA 8.0 toolkit.

A. Performance Metrics

We measure runtime overheads by running workloads directly on the system with the GPU. For the application-level runtime, we take the average time from five consecutive runs

after a warm-up run. We obtain the GPU kernel-level runtime by analyzing a GPU execution trace that contains the times when the kernels are launched and their duration. The *--print-gpu-trace* option for the *nvprof* tool prints this trace. We exclude the time spent copying data between the GPU and host memory.

To understand the source of slowdowns, we collect the total number of dynamic instructions, number of spill/fills, increase in register usage per thread, warp occupancy, warp execution efficiency, and stall reasons using *nvprof*. We measure the increase in the binary file size as a secondary overhead metric.

Evaluating the optimizations that require hardware support: We implement the modifications needed for HW-Notify and HW-Sig in NVIDIA's production compiler and measure the performance overhead using real GPUs. Since these techniques propose using new ISA extensions that are not available on current GPUs, we generate instructions that are closest in term of performance and functionality to measure expected runtime overheads. For example, we generate `LOP` with a dummy destination register (*RZ*) in place of the HW-Notify compare-and-trap instruction. For HW-Notify, we remove the notification instructions and the control-flow to branch around them. For HW-Sig, we remove all the signature update instructions such that there are no verification and notification instructions.

Comparison to Thread-Level Duplication: We quantitatively compare SInRG to a prior competitive GPU software-based solution – thread-level duplication (TLD) [19], [20], [21]. We implemented a TLD algorithm that is similar to the *Intra-Group-LDS FAST* configuration from [20] and the *Intra-Permute* configuration from [21], which is the most aggressive organization that they consider. On every memory write, TLD communicates the address and value to the neighboring redundant thread using a `SHFL` instruction, compares them with the local values, and notifies higher layers on an error (where only the redundant thread performs this last task). We also implemented an optimization called *TLD-Sig* that defers the notification until the end of the function using a predicate signature register (not explored by prior work). A thorough exploration of TLD optimizations is beyond this papers's scope.

B. Coverage Metrics

Dynamic instruction coverage: We measure the fraction of dynamic instructions in a program that are assumed to be protected by SInRG. We measure this by first categorizing instructions at compile-time by modifying the back-end compiler as follows. (1) All duplication-eligible instructions are categorized as *covered original*. (2) All duplicate instructions are categorized as *covered duplicated*. (3) We categorize verification and notification instructions as *verification*. An error in these instructions will likely result in a verification failure, assuming only a single fault occurs during a program run. (4) Instructions that are not duplicated are categorized as *uncovered*. We conservatively mark instructions as uncovered if we cannot identify an instruction that covers the instruction following compiler transformations that are performed after the duplication pass. (5) Remaining instructions, mostly consisting of register spills and fills, are categorized as *others*. Since most of the registers are duplicated in DRDV, a corruption during a spill or fill will likely be detected by the code that verifies the register value once it is filled. An error in a spilled register that is never filled has no consequence.

We next obtain dynamic instruction counts per static instruction using a binary instrumentation tool, which is similar to SASSI [43]. Combining this data with the above instruction categories, we obtain the dynamic instruction coverage. This metric assumes that all instructions have equal vulnerability.

SDC reduction: We conducted architecture-level error injections for all of our workloads using a modified version of the SASSIFI tool [44]. We injected single-bit flips into the destination registers of randomly selected SASS instructions (one error per run). This methodology, unlike dynamic instruction coverage, accounts for architecture-level propagation. We observe no error detections during error-free runs, which confirms that SInRG's false-positive rate is zero. We calculate 95% confidence intervals using the Wilson score interval and find that all intervals are less than 5% of the estimated mean. We modified some of the workloads such that SDC identification is feasible, which include printing the final result and fixing the random number generator's seed for deterministic runs.

We also conducted accelerated high-energy particle beam experiments to quantify the effect of employing SInRG on the true SDC rate at the full GPU level. Accelerated particle beam testing is one of the most accurate and widely-accepted methods of measuring FIT. We conducted the experiments at a proton facility with particle energy >200MeV. We used a Volta-based GPU [4] with ECC enabled and targeted the entire GPU package. We used our modified back-end compiler with the CUDA 9.0 toolkit and recompiled the workloads for Compute Capability 7.0 without any technical challenges, which demonstrates that SInRG algorithms are portable across toolkits and applicable to different architectures. Due to the statistical nature of the experiments and limited availability of beam time, we studied the FIT rate reduction for only the matrix multiplication workload. We used two DRDV versions: one based on the Section IV-B (DRDV) and a second similar version that duplicates loads using a function-level heuristic. This heuristic marks all loads as deterministic based on the non-existence of an atomic operations in the function, which was appropriate for this workload. These two versions, referred to in Section VI as *DRDV* and *DRDV with LD dup*, respectively, allow us to understand the effect of not duplicating most loads.

C. Area Costs and Effectiveness Analysis

We implement Verilog models of the structures needed for HW-Sig to estimate their hardware costs. The circuits are synthesized with the Synopsys toolchain using a 16nm industrial technology library [45]. We estimate circuit area using a NAND2 gate-equivalents metric. We conducted gate-level error injections to evaluate the efficacy of using a SEC-DED accumulator with HW-Sig, as a single error in the pipeline can propagate to many erroneous bits, and the SEC-DED code can alias if more than three bits are erroneous. We use the Hamartia frame-

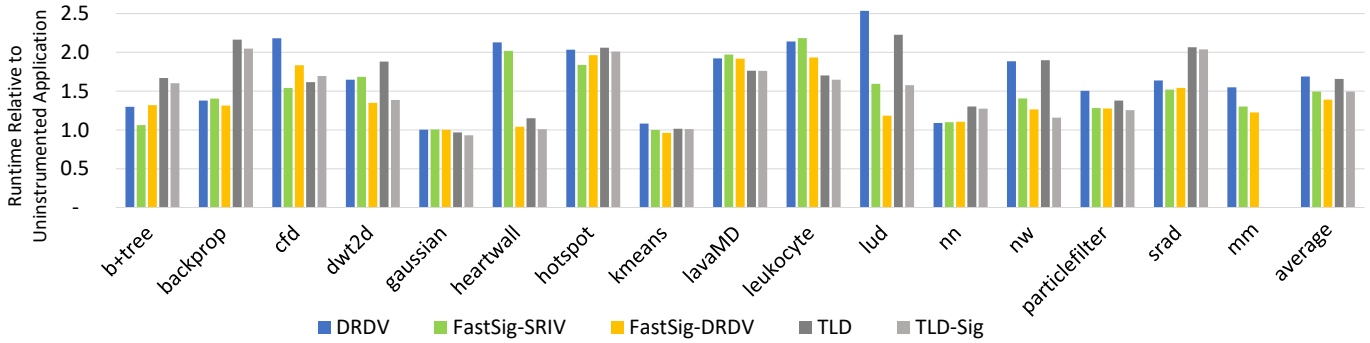


Fig. 5: The runtime overheads of the base DRDV along with optimized software-only FastSig SInRG versions.

work [46] to flip the the output of a single gate or flip-flop per injection. This methodology is similar to that of Nedel et al. [47], though we use netlist rewriting to simulate errors without modifying the gate-level simulator. Our results are based on six unpipelined DesignWare components [48], using random inputs.

VI. SInRG Evaluation

A. Software-only techniques

Performance: We begin our evaluation by measuring the performance overheads of the baseline SInRG versions. As the baseline SRIV incurs very high overheads (>100% for all our workloads), we do not analyze it further. The GPU kernel runtime for DRDV incurs an arithmetic average overhead of 69%, as shown in Figure 5. Employing the software-only FastSig optimization reduces the average runtime overheads to 39% and 49% for DRDV and SRIV, respectively. Workloads with low baseline IPC have more potential for improvement when employing SInRG. Underutilized resources (due to memory operations, hazards, poor code, etc.) can cause low IPC, which SInRG exploits by hiding duplication overhead. To understand the effect of SInRG's runtime overheads on different architectures, we evaluated FastSig-DRDV on a Volta-based GPU for a subset of workloads and observed similar overhead trends.

Our results show that the average application-level runtime overheads are only 6%, 4%, and 5% for baseline DRDV, FastSig-DRDV, and FastSig-SRIV, respectively. This is much lower than the GPU kernel-level overheads because these runtimes include time spent on host and copying memory.

FastSig-DRDV outperforms FastSig-SRIV for some workloads, such as *lud* (18% versus 59% overhead). This phenomenon can be explained by the difference in dynamic instruction count, which is 33% higher for FastSig-SRIV. FastSig-SRIV has better performance for some workloads, despite executing more dynamic instructions. For example, the runtime overheads for *b+tree* are 32% versus 6% for FastSig-DRDV and FastSig-SRIV, respectively. Although the SRIV version executes more dynamic instructions, the warp occupancy is 1.82× higher, resulting in better overall performance.

Figure 6 further explains the dynamic instruction count increase. The results are normalized per workload to the total instruction count of FastSig-DRDV and show that FastSig-SRIV always executes more instructions than FastSig-DRDV. As discussed above, this is not the only indicator of performance. Warp occupancy, which is also plotted in Figure 6 on the secondary axis, is almost always reduced by using FastSig-DRDV. This relative decrease correlates well with the relative runtime overhead increase (Figure 5) compared to FastSig-SRIV. In summary, selecting an instruction duplication algorithm that is aware of the GPU resource requirements for a workload can provide better performance.

Comparison to thread-level duplication: If a workload has spare thread and register resources, it can benefit from TLD beyond what SInRG offers because SInRG does not exploit spare thread resources. This is the case for *lavaMD* and *leukocyte*, as shown in Figure 5. These workloads exhibit low warp occupancy (Figure 6) for both FastSig versions, since they are not register resource limited.

As mentioned in Section V-A, TLD-Sig optimizes TLD by deferring error notification until the end of the function, which reduces the runtime overheads for most of the workloads. Results show that despite this optimization, one of the FastSig optimized SInRG versions outperforms TLD-Sig for a majority of the workloads.

Code bloat: The average increase in the program binary file size, which includes non-duplicated host code, was a modest 12% for FastSig-DRDV and 19% for FastSig-SRIV. SInRG's static instruction overhead ranges from 74%–115% for FastSig-DRDV and from 180%–227% for FastSig-SRIV. The overheads for FastSig-SRIV are relatively higher because it adds more verification instructions.

Dynamic instruction coverage: Results in Figure 6 show that the original programmer-defined instructions (other than compiler-inserted spill and fill code) account for an average of 36% and 25% of the total dynamic instructions for FastSig-DRDV and FastSig-SRIV, respectively. The duplicated instructions account for a similar fraction. The percentage of verification instructions varies significantly based on the workload and the algorithm. As expected, the average percentage is 2.4× more for FastSig-SRIV compared to FastSig-DRDV (35% versus 15%). While a small fraction of instructions are categorized as *others* for most workloads, spills and fills increase the prevalence of this instruction class for some register constrained workloads. We assume all the above instructions are covered by SInRG. Finally, the fraction of uncovered instructions also varies by workload and depends on the prevalence of control, global and shared memory reads, and

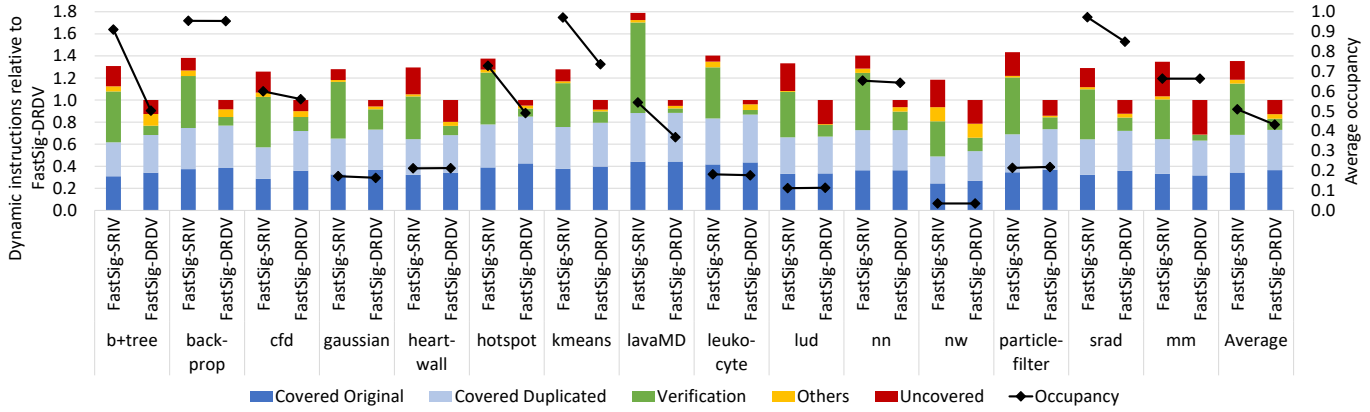


Fig. 6: Dynamic instruction-class counts. The secondary y-axis shows the average warp occupancy of each workload. The fraction of uncovered instructions is generally small, as shown by the top segment of each bar.

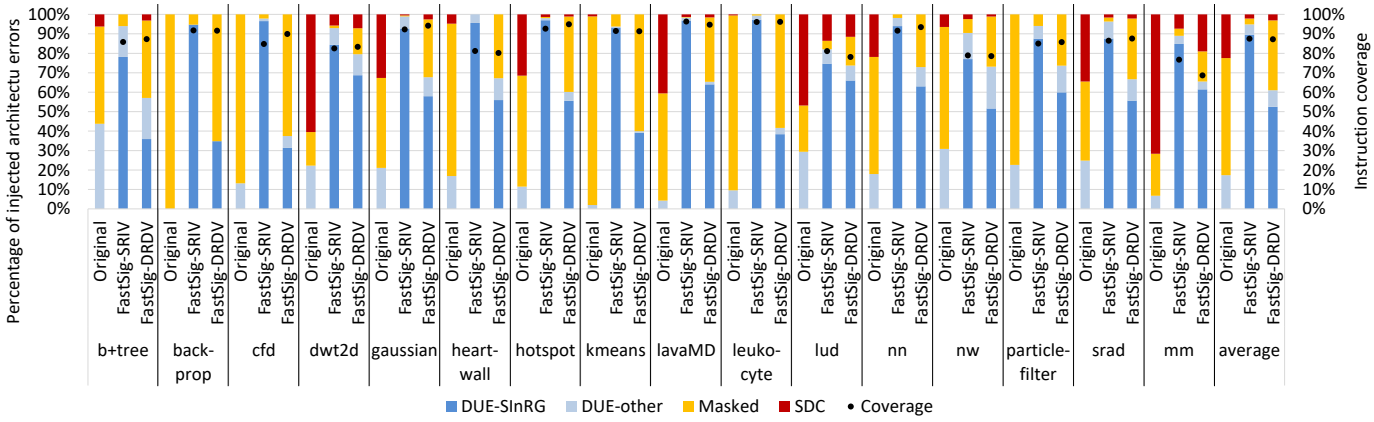
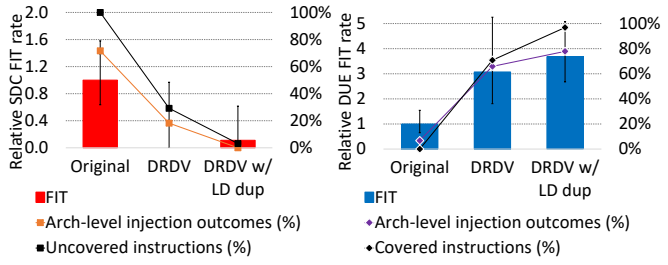


Fig. 7: Architecture-level error injection results. Original refers to uninstrumented program.



(a) SDC (b) DUE

Fig. 8: SDC and DUE FIT rates for the mm workload, normalized to the original mm. Architectural injection and instruction coverage results are plotted on the secondary y-axis.

atomics in the program. On average 88% and 87% of the dynamic instructions are considered covered by FastSig-SRIV and FastSig-DRDV, respectively.

SDC reduction through architecture-level error injections: Figure 7 shows architecture-level error injection results for the uninstrumented programs and the two SInRG versions. It shows that SInRG is effective in reducing the SDC percentage. We expect the dynamic instruction coverage, plotted on the secondary y-axis, to correlate well with FastSig-SRIV's DUE (Detected Unrecoverable Errors [49]) percentage because FastSig-SRIV performs immediate verification, providing no opportunity for error masking. Since the error notification is delayed, some of the errors may result in DUE-other (crashes/hangs) prior to being flagged by SInRG as DUE-SInRG (e.g., b+tree, lud). FastSig-DRDV provides opportunity for masking until the end of the data-flow chains, lowering the expected DUE rate. The results clearly show this trend — the SDC percentage is always lower than the percentage of uncovered instructions.

SDC reduction through accelerated particle testing: Figure 8a shows the effectiveness of SInRG in reducing the GPU SDC FIT rate while running the *mm* workload. The observed SDC rate for both SInRG versions is an order of magnitude lower than the uninstrumented program. Figure 8b shows that the DUE FIT rate increases with SInRG; we observed the evidence of several SInRG error detections in the system logs. These results establish that SInRG is effective in significantly improving the reliability of GPUs.

We plot the dynamic instruction coverage and architecture-level error injection results in Figure 8 (on the secondary y-axes) to analyze the trends. These results, however, cannot be directly compared to FIT rates because these methods estimate the program-level error propagation probabilities once the error has manifested at the architecture level.

Figure 8a shows that the percentage of uncovered dynamic instructions reduces from 100% to 29% and 3% for DRDV without and with load duplication, respectively. The

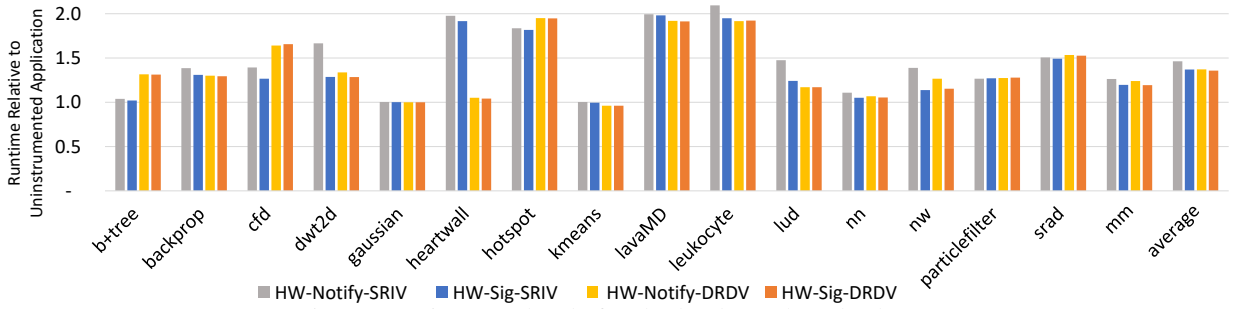


Fig. 9: Runtime overheads for the hardware-based schemes.

corresponding SDC percentage reduction from architecture-level error injection are 18% and 0%, down from 72%. These trends correlate strongly with each other and the FIT estimates. We noted similar strong correlations for the DUE results.

B. Optimizations through Hardware Support

Performance: We apply HW-Notify to the two FastSig versions to overcome their limitation and provide perfect error containment. Figure 9 shows the average overheads for HW-Notify-DRDV and HW-Notify-SRIV are 37% and 46%, respectively, which are similar to the FastSig versions.

HW-Sig eliminates the verification instructions altogether providing a faster solution. HW-Sig-SRIV is significantly faster than the HW-Notify-SRIV, with average overheads of 33% versus 44%. This improvement is expected because FastSig-SRIV has many verification instructions that HW-Sig eliminates. We did not observe such a high improvement for HW-Sig-DRDV over HW-Notify-DRDV (35% to 33%). Even though the average overheads between HW-Sig-SRIV and HW-Sig-DRDV are similar, we observe significant differences for different workloads. When HW-Sig is employed with DRDV, maintaining the shadow data-flow chain and the shadow register space provides additional instruction scheduling flexibility, which can be beneficial for workloads that are not register limited.

Hardware costs: Table II gives the circuit area estimates following synthesis, as well as a 32b adder and a SEC-DED encoder for reference. The HW-based SInRG schemes require modest amounts of new hardware per lane—the total area of the new structures is similar to or less than that of an adder, which itself represents a small fraction of the pipeline logic. The 64b HW-Sig accumulator is shown with two signature accumulation algorithms—GF(2) XOR and one's complement. The HW-Sig accumulator is the largest new structure, but it can be efficiently replaced by a SEC-DED accumulator at less than 1/6 (GF(2) XOR) or 1/8 (one's complement) the cost.

The SEC-DED accumulator potentially has imperfect error coverage because a single-event transient error in the pipeline can propagate to many erroneous bits, and the SEC-DED code can alias if more than three bits are in error. Our gate-level error injection campaign determined this risk to be minimal—over 3,862 logically unmasked errors injected into six fixed-point and floating-point arithmetic units, only one would remain uncaught by the SEC-DED accumulator. This leads to a 95% confidence interval of (0.0%, 0.2%) for the percentage of errors that the SEC-DED accumulator would miss.

TABLE II: Area costs per lane for hardware extensions.

Structure	Technique	# FFs	Area (NAND2)
32b Zero Detector	HW-Notify	1	19
64b XOR Accumulator	HW-Sig	65	496
64b One's Accumulator	HW-Sig	65	1044
7b SEC-DED XOR Accumulator	HW-Sig	8	73
7b SEC-DED One's Accumulator	HW-Sig	8	130
32b Adder	Reference	96	715
2x 32b SEC-DED Encoders	Reference	14	354

We target a 2GHz clock (assuming 50% margin for uncertainty and unmodeled control circuitry), which is an efficient operating point for more complex functional units such as the multiply-add unit. All of the considered circuits achieve this speed using automatic register retiming.

VII. Automated Duplication Technique Selection

As explained earlier, either DRDV or SRIV can perform best for a specific GPU kernel, depending on its requirements and the available GPU resources. We explore heuristics to automatically select the SInRG algorithm for each dynamic kernel at kernel launch time. Based on our initial study, we find that supervised learning methods such as Decision Tree and Random Forest perform well for this task. Auto-selected duplication algorithm for FastSig, HW-Notify, and HW-Sig reduces the average runtime overheads to 36%, 34%, and 30%, respectively. These are significantly lower when compared to the average overheads obtained by DRDV and SRIV individually for the respective techniques.

VIII. Conclusions

As GPUs pervade HPC and safety-critical systems, it becomes important to ensure that the application which are accelerated are resilient to transient hardware errors. Software-based instruction duplication is attractive because it can be employed on state-of-the-art systems and can be selectively applied to resilience-critical workloads. In this paper, we implement intra-thread instruction duplication on GPUs (inspired by prior CPU work) and find the overheads to be high, averaging 69% over a variety of workloads. We propose several software-only and software-hardware optimizations to reduce the overheads and implement them in NVIDIA's production compiler. Our GPU-specific software optimizations trade off error containment for performance and reduce the average runtime overhead to 36%. We also propose new ISA extensions with limited hardware changes and area costs to further lower the average runtime overhead to just 30%.

REFERENCES

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. A. Chien, P. Coteus, N. A. DeBardeleben, P. C. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. V. Hensbergen, "Addressing Failures in Exascale Computing," *The International Journal of High Performance Computing Applications*, vol. 28, pp. 129–173, May 2014.

[2] C. Constantinescu, "Trends and Challenges in VLSI Circuit Reliability," *IEEE Micro*, vol. 23, pp. 14–19, July 2003.

[3] D. A. Oliveira, P. Rech, L. L. Pilla, P. O. Navaux, and L. Carro, "GPGPUs ECC Efficiency and Efficacy," in *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT)*, pp. 209–215, October 2014.

[4] NVIDIA, "NVIDIA Tesla V100 GPU Architecture, The World's Most Advanced Datacenter GPU." http://www.nvidia.com/object/volta-architecture-whitepaper.html, 2017.

[5] NVIDIA, "NVIDIA Tesla P100, The Most Advanced Datacenter Accelerator Ever Built." https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf, 2016.

[6] NVIDIA, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110." https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf, 2012.

[7] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight Error Detection for GPGPU," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 37–47, December 2012.

[8] W. Bartlett and L. Spainhower, "Commercial Fault Tolerance: A Tale of Two Systems," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, pp. 87–96, January 2004.

[9] X. Iturbe, B. Venu, E. Ozer, and S. Das, "A Triple Core Lock-Step (TCLS) ARM® Cortex®-R5 Processor for Safety-Critical and Ultra-Reliable Applications," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, pp. 246–249, June 2016.

[10] N. Oh, P. Shirvani, and E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-scalar Processors," *IEEE Transactions on Reliability*, vol. 51, pp. 63–75, March 2002.

[11] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "SWIFT: Software Implemented Fault Tolerance," in *International Symposium on Code Generation and Optimization (CGO)*, pp. 243–254, March 2005.

[12] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, "Software-controlled Fault Tolerance," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, pp. 366–396, December 2005.

[13] S. Che, J. W. Sheaffer, M. Boyer, L. G. Szafaryn, L. Wang, and K. Skadron, "A Characterization of the Rodinia Benchmark Suite with Comparison to Contemporary CMP Workloads," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–11, December 2010.

[14] M. Burtscher, R. Nasre, and K. Pingali, "A Quantitative Study of Irregular Programs on GPUs," in *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 141–151, November 2012.

[15] A. Nukada, H. Takizawa, and S. Matsuoka, "NVCR: A transparent checkpoint-restart library for NVIDIA CUDA," in *IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pp. 104–113, May 2011.

[16] H. Takizawa, K. Sato, K. Komatsu, and H. Kobayashi, "CheCUDA: A checkpoint/restart tool for CUDA applications," in *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pp. 408–413, December 2009.

[17] N. Toan, H. Jitsumoto, N. Maruyama, T. Nomura, T. Endo, and S. Matsuoka, "MPI-CUDA applications checkpointing," in *IPSJ SIG Technical Report*, pp. 1–7, August 2010.

[18] R. Garg, A. Mohan, M. Sullivan, and G. Cooperman, "CRUM: Checkpoint-Restart Support for CUDA's Unified Memory," in *Proceedings of International Conference on Cluster Computing (CLUSTER)*, September 2018.

[19] M. Dimitrov, M. Mantor, and H. Zhou, "Understanding Software Approaches for GPGPU Reliability," in *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, pp. 94–104, March 2009.

[20] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-World Design and Evaluation of Compiler-Managed GPU Redundant Multithreading," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 73–84, June 2014.

[21] M. Gupta, D. Lowell, J. Kalamatianos, S. Raasch, V. Sridharan, D. Tullsen, and R. Gupta, "Compiler Techniques to Reduce the Synchronization Overhead of GPU Redundant Multithreading," in *Proceedings of the Design Automation Conference (DAC)*, pp. 1–6, June 2017.

[22] NVIDIA, "CUDA C Programming Guide :: CUDA Toolkit Documentation." http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, September 2017.

[23] Khronos Group, "The Open Standard for Parallel Programming of Heterogeneous Systems," March 2017.

[24] NVIDIA, "PTX ISA :: CUDA Toolkit Documentation." http://docs.nvidia.com/cuda/parallel-thread-execution/, September 2017.

[25] NVIDIA, "CUDA Binary Utilities :: CUDA Toolkit Documentation." http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, September 2017.

[26] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures," *IEEE Transactions on Dependable and Secure Computing (TDSC)*, vol. 6, pp. 135–148, April 2009.

[27] Y. Zhang, S. Ghosh, J. Huang, J. W. Lee, S. A. Mahlke, and D. I. August, "Runtime Asynchronous Fault Tolerance via Speculation," in *International Symposium on Code Generation and Optimization (CGO)*, pp. 145–154, April 2012.

[28] A. Shye, T. Moseley, V. Reddi, J. Blomstedt, and D. Connors, "Using Process-level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 297–306, June 2007.

[29] C. Wang, H. Kim, Y. Wu, and V. Ying, "Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection," in *International Symposium on Code Generation and Optimization (CGO)*, pp. 244–258, March 2007.

[30] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 25–36, June 2000.

[31] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operation Systems (ASPLOS)*, pp. 385–396, March 2010.

[32] J. Rivers, M. Gupta, J. Shin, P. Kudva, and P. Bose, "Error Tolerance in Server Class Processors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 30, pp. 945–959, July 2011.

[33] R. Nathan and D. J. Sorin, "Argus-G: Comprehensive, Low-Cost Error Detection for GPGPU Cores," *IEEE Computer Architecture Letters*, vol. 14, pp. 13–16, January 2015.

[34] J. Tan and X. Fu, "RISE: Improving the Streaming Processors Reliability Against Soft Errors in GPGPUs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 191–200, September 2012.

[35] M. Abdel-Majeed, W. Dweik, and M. Annavaram, "Warped-RE: Low Cost Error Detection and Correction in GPUs," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 331–342, June 2015.

[36] P. Xiang, Y. Yang, M. Mantor, N. Rubin, and H. Zhou, "Revisiting ILP designs for throughput-oriented GPGPU architecture," in *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pp. 121–130, May 2015.

[37] A. Nistor, D. Marinov, and J. Torrellas, "Instantcheck: Checking the determinism of parallel programs using on-the-fly incremental hashing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 251–262, December 2010.

[38] J. Ohlsson and M. Rimen, "Implicit Signature Checking," in *Twenty-Fifth International Symposium on Fault-Tolerant Computing. Digest of Papers*, pp. 218–227, June 1995.

[39] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow Checking by Software Signatures," *IEEE Transactions on Reliability*, vol. 51, pp. 111–122, March 2002.

[40] NVIDIA, "XID Errors." http://docs.nvidia.com/deploy/xid-errors/index.html, October 2017.

[41] P. Koopman, K. Driscoll, and B. Hall, "Selection of cyclic redundancy code and checksum algorithms to ensure critical data integrity," Tech. Rep. 3-2015, Carnegie Mellon University, March 2015.

[42] 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 *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, October 2009.
[43] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Connor, and S. W. Keckler, "Flexible Software Profiling of GPU Architectures," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 185–197, June 2015.
[44] S. K. S. Hari, T. Tsai, M. Stephenson, S. Keckler, and J. Emer, "SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation," in *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249–258, April 2017.
[45] Synopsys Inc., "Design Compiler J-2014.09," August 2014.
[46] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Hamartia: A Fast and Accurate Error Injection Framework," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 101–108, June 2018.
[47] W. Nedel, F. Kastensmidt, and J. Azambuja, "Implementation and experimental evaluation of a CUDA core under single event effects," in *2014 15th Latin American Test Workshop (LATW)*, pp. 1–4, March 2014.
[48] Synopsys Inc., "Designware IP library."
[49] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: an architectural perspective," in *11th International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 243–247, February 2005.

APPENDIX

ARTIFACT DESCRIPTION: INSTRUCTION LEVEL DUPLICATION ALGORITHMS

A. Abstract

This section contains the details of implementing instruction level duplication in the backend compiler.

B. Overview

The following terminology is used in the algorithms:

- *Duplication eligible* instructions, which are instructions that (1) produce deterministic values, (2) do not directly modify the control flow, and (3) do not write to memory.
- *Copy eligible* instructions are instructions that are not duplication eligible and write to a register, such as atomic operations and special registers.

The SInRG algorithms operate at the intermediate representation (IR) in ptxas, which is close to SASS assembly code. Since SInRG algorithms run before register allocation, they operate on virtual registers and can easily create new (shadow) registers which are later mapped to the limited set of physical registers.

In SInRG, we duplicate every duplication-eligible instruction, using a data-structure to track already-protected instructions so as not to duplicate them multiple times. We then create the data structure to track the shadow register mapping. In our implementation, we duplicate all major register classes—general-purpose registers, predicate registers, and condition codes—except for the predefined registers such as zero-value register and thread-id. Next, for each original duplication-eligible instruction, we duplicate the instruction and map the duplicate into the shadow register space.

Instructions that are not eligible for duplication include memory writes, control flow instructions, instructions that produce non-deterministic values, barrier spill/fill instructions, and instructions that write to pre-assigned physical registers. We also do not apply the pass to built-in CUDA Runtime API calls such as cudaDeviceGetCacheConfig and cudaDeviceSetCacheConfig, which get and set the preferred cache configuration for the current device, respectively.

C. DRDV: Double the virtual Register space and Delay Verification

For DRDV, we place the duplicate instruction after the original instruction and map the registers used by it into a shadow register space. For all non-duplicated *copy eligible* instructions, we insert a move instruction to copy the destination register value into the shadow register space so that duplicated instructions can use it. Finally, we insert verification instructions to check original and shadow register values for all inputs to non-duplicated instructions. This approach reduces the verification overhead (compared to SRIV) by chaining multiple replicated instructions on the path to a single verification. Algorithm 1 describes our implementation of the DRDV instruction duplication algorithm.

Algorithm 1: The *DRDV* back-end compiler instruction duplication algorithm, run once per function.

```
create list of original instructions
clear original to shadow register mapping
for each instruction in the function do
    if instruction is duplication-eligible and original then
        duplicate the original instruction
        for all operands in the duplicate instruction do
            if shadow register does not exist then
                create a shadow register for the source
            end
            replace original register to shadow register
        end
    else if instruction is copy eligible and original then
        insert a move instruction to copy the
          destination register value to the shadow space
    end
end
for each instruction in the function do
    if instruction is not duplication eligible and is original then
        for all sources in this instruction do
            verify original
              and shadow registers have same value
            if values are different then
                notify error to higher level (trap)
            end
        end
    end
end
```

D. SRIV: Single Register space and Immediate Verification

For SRIV, we replace the destination registers in the duplicate instruction with new virtual registers. Since the original instruction may overwrite its source operand and we want the duplicate instruction to generate the same result as the original instruction using the same source operands, we place the duplicate before the original instruction. Next, we insert verification instructions to check the original and new register values after the original instruction. Verification and notification consist of a comparison operation, a conditional

branch instruction, and a trap instruction (BPT) to notify an outer layer of an error. Algorithm 2 describes our implementation of the SRIV instruction duplication algorithm.

Algorithm 2: *SRIV* back-end compiler instruction duplication algorithm, run on each function.

```
1  create list of original instructions
2  for each instruction in the function do
3      if instruction is duplication eligible and original then
4          duplicate and place it before the original instruction
5          for all
             destination registers in the duplicate instruction do
6            replace
               original register with a new virtual register
7            verify the
               original and new registers have same value
8            if values are different then
9              notify error to higher level (trap)
10           end
11       end
12 end
```

E. FastSig: Signature-based checking

The implementation of both FastSig versions for DRDV and SRIV extend from the base algorithms above, with a few important additions. First, the signature register needs to be set to an initial value at the start of the function. Second, during the verification step, an LOP3 (or any high-throughput update function) is used instead of a comparison operation. Third, error notification is no longer done during the inner-most loop — instead, it is deferred to the end of the function. Finally, the signature register value is checked at the end of the function against the initial value it was set to, and a trap is triggered if the values do not match. Figure 3 in the paper illustrate how the assembly code may look after the FastSig implementation.

F. HW-Sig Implementation Details

For HW-Sig, we first initialize a 64-bit signature register at kernel launch time, and at the end check to see if the value is zero. As each instruction executes, it updates the signature register by adding or subtracting its destination register values based on whether the instruction is original or duplicate, respectively. Operations that are commutative, easy to design in hardware, and require low area overhead are good candidates for signature updates. For example, binary Galois Field arithmetic (GF(2)) that uses *XOR* operations or one's complement accumulation or subtraction operations can be used for signature accumulation and subtraction [41]. We need one extra meta-data bit in the instruction to indicate whether the signature register should be updated by the results of the instruction. Instructions that are not duplicated do not update the signature. We need one more meta-data bit if accumulation and subtraction operations are different (e.g., one's complement arithmetic). The signature update logic need not be in the critical path and can be performed in parallel in the write back stage as the result of the instruction is being written back to the register file.