NVBitFI: Dynamic Fault Injection for GPUs

Timothy Tsai, Siva Kumar Sastry Hari, Michael Sullivan, Oreste Villa, Stephen W. Keckler
{timothyt,shari,misullivan,ovilla,skeckler}@nvidia.com
NVIDIA

Abstract*—GPUs have found wide acceptance in domains such as high-performance computing and autonomous vehicles, which require fast processing of large amounts of data along with provisions for reliability, availability, and safety. A key component of these dependability characteristics is the propagation of errors and their eventual effect on system outputs. In addition to analytical and simulation models, fault injection is an important technique that can evaluate the effect of errors on a complete computing system running the full software stack. However, the complexity of modern GPU systems and workloads challenges existing fault injection tools. Some tools require the recompilation of source code that may not be available, struggle to handle dynamic libraries, lack support for modern GPUs, or add unacceptable performance overheads. We introduce the NVBitFI tool for fault injection into GPU programs. In contrast with existing tools, NVBitFI performs instrumentation of code dynamically and selectively to instrument the minimal set of target dynamic kernels; as it requires no access to source code, NVBitFI provides improvements in performance and usability. The NVBitFI tool is publicly available for download and use at https://github.com/NVlabs/nvbitfi.

***Keywords*-Fault injection, GPU, dynamic instrumentation, error propagation**

I. Introduction

Complex computing systems increasingly use GPUs due to their high performance and power efficiency. General-purpose computing with GPUs has been used with large-scale machines and high-performance computing (HPC) for over a decade; this trend looks likely to continue as a majority of the ten fastest and most efficient supercomputers in the world currently use GPUs [1], [2]. Another trend that has accelerated the use of GPUs is the advent of deep neural networks (DNNs) with implementations that are readily amenable to the massive parallelism available in GPUs. While many of these DNN installations are located in data centers, DNNs are also finding widespread adoption in safety-critical systems such as autonomous vehicles (AV) for processing data from cameras and other sensors.

Both HPC and safety-critical systems have dependability requirements, albeit in different ways. An HPC system presents a large state-time space for faults to occur: a large system size contains many potentially faulty components, and the relatively long run-time of HPC applications allows faults over a large time period to potentially affect outputs. For an HPC system, both reliability (are the results correct?) and availability (are results produced?) are important. In contrast, AV systems (an example of a safety-critical system) are embedded control systems that continuously process a stream of incoming telemetry from cameras, radars, LiDARs, GPS, and other sensors to accurately perceive the environment around the vehicle. While the state-time space for an AV system is smaller due to less computing hardware and lower run-time (at least in terms of processing a single set of telemetry), the safety requirements for AV systems are very stringent because of the potential for loss of life or significant property damage. As a result, AV computing systems generally have backup modes of operation even in the event of catastrophic failure (e.g., power or clock signal failures). However, undetected errors may fail to trigger the backup mode and instead result in silent data corruption (SDC) that leads to erroneous vehicle behavior and accidents.

Although faults can occur in the CPU, GPU, memory, storage, network, power supplies, and other components of a computing system, this paper focuses on faults that occur in GPUs. For these GPU-based systems, much of the computation and live state is in the GPUs, and thus much of the opportunity to affect live state is in the GPU. However, most of these GPU faults do not propagate to affect the outputs of the program. For example, faults may flip values in non-live bits that are never read after the corruption, or the erroneous values may be not be mathematically significant to the computation of the final program outputs.

Estimation of the probability that a fault will affect program outputs is needed to understand the expected failure rate of a system and its components. The architectural vulnerability factor (AVF) [3] is the probability that a fault will result in a visible error in the final output of a program. The product of the raw error rate and the AVF results in the visible error rate for the program on a particular system. The AVF is affected by both the control and data flow of the program as well as the design of the underlying hardware. As a result, estimating the AVF requires understanding a specific program as it executes on a specific hardware processor.

Several techniques exist for estimating the error propagation of a GPU-based computing system. Section II provides an overview of these techniques and the related published literature. Some of the AVF estimation techniques use models that simulate a microarchitectural representation of the GPU and inject faults into the simulation model to estimate AVF. Other techniques perform architectural fault injection by injecting errors into physical GPUs.

This paper presents a new tool called NVBitFI that instruments a target program to inject errors into NVIDIA GPUs. In contrast to existing tools, NVBitFI offers the following key advantages:

- **Binary instrumentation.** NVBitFI instruments the SASS

TABLE I
PHYSICAL-GPU FAULT INJECTION TOOLS.

Year	Tool	Injection mechanism	Fault model level	Needs source code?	Inject libraries?
2020	NVBitFI	NVBit	SASS	No	Yes
2017	SASSIFI [4]	SASSI	SASS	Yes	No
2016	LLFI-GPU [5]	LLVM	LLVM IR	Yes	No
2014	GPU-Qin [6]	cuda-gdb	SASS	No	Maybe
2011	Hauberk [7]	source code	C++	Yes	No

[8] code that represents the lowest level GPU assembly code, which allows NVBitFI to perform instrumentation without the need for source code.

- **Dynamic instrumentation.** NVBitFI intercepts dynamic GPU kernel calls, which allows it to target dynamically loaded libraries, including libraries that are not known at build time. The instrumentation is limited to the minimal set needed for error injection, thus limiting the total performance overhead of the instrumentation code.
- **Architectural abstraction.** NVBitFI presents a single interface that works on all recent NVIDIA architecture families including Kepler, Maxwell, Pascal, Volta, and Ampere GPUs.

NVBitFI is a module built using the NVBit dynamic binary instrumentation framework [9]. NVBit and the NVBitFI architecture, injection mechanisms, supported fault models, and usage are described in Section III. We present examples of usage for NVBitFI using the SpecACCEL benchmark in Section IV, including showing how different tool options affect AVF results and performance overhead. Section V discusses future work and ideas. The NVBitFI tool is available for download at `https://github.com/NVlabs/nvbitfi`.

II. Related Work

Hardware injection through program transformation. Program transformation tools inject faults into a program running on a physical GPU by instrumenting the target GPU program. This instrumentation injects errors by corrupting the architecturally visible program state, such as register or memory values. The injected error then propagates at hardware speeds as the instrumented program executes on the GPU. While this paper focuses on CUDA-programmable GPUs, the same concepts are applicable to other GPUs. Table I lists several program transformation tools and summarizes the fault injection mechanism for each one.

Hauberk [7] uses direct modification of C++ source code to introduce injection code. Because C++ code must be translated into GPU assembly code, the ultimate GPU state corruption depends on how the compiler translates both the original program code and the injection code. LLFI-GPU [5] inserts injection code into the LLVM intermediate representation (an IR that is closer to the underlying hardware architecture), which is then compiled into PTX [10] and then the SASS [8] GPU assembly code.

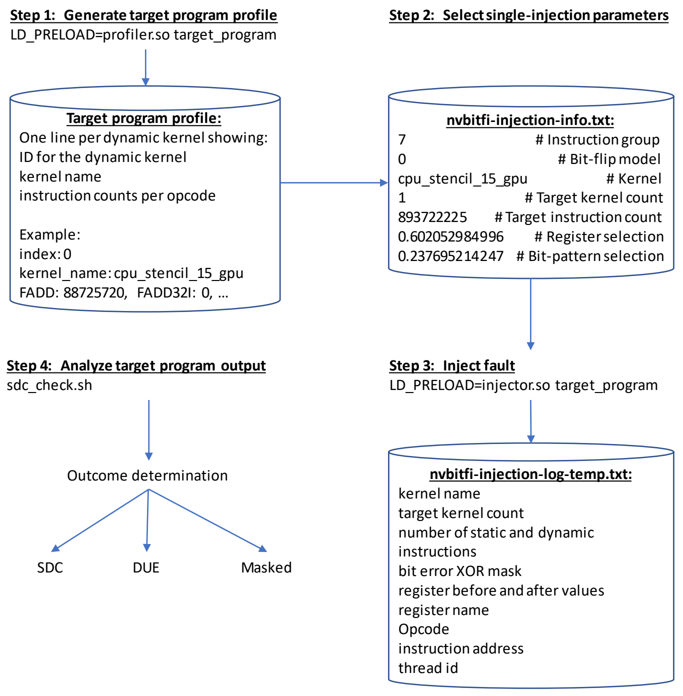
Several tools inject errors directly into the GPU assembly code (SASS), which is closest to the hardware and is not subject to compiler scheduling variations. SASSIFI [4] is a compiler framework based on the SASSI [11] framework that inserts error injection code as a part of the final compiler pass that generates the SASS code. GPU-Qin [6] is a debugger-based tool that uses cuda-gdb [12] to set breakpoints at which error are injected using debugger commands. Our NVBitFI tool is similar to SASSIFI and GPU-Qin as they all operate on the SASS level. The key difference is that NVBitFI uses the NVBit [9] framework to perform dynamic code instrumentation that intercepts dynamic kernel calls and inserts error injection code on the fly (1) without requiring any source code or recompilation and (2) without affecting any instruction scheduling or register allocation of the target program.

Simulator-based injection. Simulation-based tools may be based on microarchitectural, RTL, netlist, or circuit models. Because simulation-based models have a trade-off between fidelity and simulation speed, most tools in the published literature are based on microarchitectural models, although proprietary tools based on lower-level representations may exist. Several GPU families are represented by prior tools, including SIFI [13] based on the AMD Southern Islands family using Multi2Sim v4.2 [14], GUFI [15] and GPGPU-SODA [16] based on NVIDIA CUDA [17] GPUs both using GPGPU-Sim [18], and tools based on AMD APUs [19], [20] using gem5 [21]. Because these tools are based on openly available simulators, they cannot truly represent the behavior of a commercial GPU. While simulator-based injection tools can capture the effects of specific microarchitecture errors, they are also several orders of magnitude slower than hardware-based injection frameworks, limiting them to small programs or program fragments.

Discussion. The key advantages of NVBitFI are due to dynamic instrumentation of program binaries and the use of a generalized GPU architecture abstraction. Binary instrumentation allows targeting of programs without requiring source code and can instrument static or dynamically loaded libraries. NVBitFI can also inject errors into dynamically selected basic blocks or kernels, eliminating instrumentation overhead in the rest of the program. Although SASS is the name for the NVIDIA GPU instruction set architecture (ISA), SASS instructions and their encodings can change across GPU generations. NVBitFI leverages the architectural abstraction offered by NVBit to handle all recent NVIDIA GPU families, from Kepler to Ampere. Finally, NVBitFI runs at GPU speeds, meaning error injection campaigns are performed at rates of billions of instructions per second.

III. NVBitFI Architecture and Description

NVBitFI is a program transformation fault injection tool that offers ease of use, low overhead, and the ability to target


Step 1: Generate target program profile
LD_PRELOAD=profiler.so target_program
Step 2: Select single-injection parameters
Step 3: Inject fault
LD_PRELOAD=injector.so target_program
Step 4: Analyze target program output
sdc_check.sh
Outcome determination
SDC
DUE
Masked

Target program profile:
One line per dynamic kernel showing:
ID for the dynamic kernel
kernel name
instruction counts per opcode

Example:
index: 0
kernel_name: cpu_stencil_15_gpu
FADD: 88725720, FADD32I: 0, …

nvbitfi-injection-info.txt:

```
7                        # Instruction group
0                        # Bit-flip model
cpu_stencil_15_gpu       # Kernel
1                        # Target kernel count
893722225                # Target instruction count
0.602052984996           # Register selection
0.237695214247           # Bit-pattern selection
```

nvbitfi-injection-log-temp.txt:
kernel name
target kernel count
number of static and dynamic instructions
bit error XOR mask
register before and after values
register name
Opcode
instruction address
thread id

Fig. 1. Injection procedure for a single fault showing profiling and injection commands along with parameter and output files.

dynamic libraries and other software for which source code is not available. These qualities are due in large part to the NVBit framework. In this section, we describe (1) the NVBitFI software architecture, (2) the supported fault models, and (3) the parts of the NVBit framework that are pertinent to NVBitFI. NVBitFI supports both transient and permanent fault models.

A. *NVBitFI Software Architecture*

The NVBitFI package consists of two types of tools, profilers and injectors, which are implemented as dynamic libraries that are attached to a target program. Profilers analyze a target program to build a profile of dynamic instruction counts for every opcode in every kernel. The instruction profile represents the uniform distribution of dynamic faults from which a random set of faults can be selected. Injectors use the profile for a given program to inject faults into that program. Figure 1 shows the steps used to inject a transient fault including (1) generating a program profile to determine the set of eligible injection points, (2) selection of one or more injection points for a particular experiment, (3) injecting the fault(s) by modifying the program binary, and (4) running the modified program and comparing to a *golden* output state to determine if an error propagated to the program output.

The most basic mode of usage of NVBitFI consists of attaching either a profiler or an injector library to the target program via the `LD_PRELOAD` environment variable. For example, attaching the profiler library to the `target_program` is performed in the `bash` shell with the following command:
`LD_PRELOAD=<path>/profiler.so target_program`

Profiler. The profiler is deployed in a `profiler.so` library. To prepare for the injection campaign, the profiler creates a profile containing one line for every dynamic kernel and the total dynamic instruction counts for every opcode in every thread in that dynamic kernel (Figure 1, step 1). Instructions that are not executed based on a predicate register are not included in the profile. A dynamic instruction will be selected from the set of executed instructions by choosing a random number n from the set $1..N$, where N is the total number of profiled dynamic instructions (Figure 1, step 2). This n^{th} instruction is then translated into a tuple of <*kernel_name, kernel_count, instruction_count*> values that instructs the transient fault injector to inject an error for the indicated dynamic instruction. When that instruction is reached, the injection code will corrupt the destination register of the instruction based on the bit-pattern selection value. The NVBitFI package includes convenience scripts to automate the profiling and fault selection process.

Because the profiling process requires the instrumentation of every dynamic instruction, it can take a lot of time. To address this issue, NVBitFI offers two types of profiling, exact and approximate. Exact profiling counts every dynamic instruction. Approximate profiling only counts the dynamic instructions in the first instance of every static kernel and assumes that the instruction counts for subsequent instances of the same static kernel are the same. In Section IV, we compare the results and overheads for these two types of profiling.

A permanent fault injection campaign does not require an instruction profile for a target program. However, a profile increases efficiency by only injecting instructions that are known to be executed for that program. The set of executed opcodes consists of all opcodes from the profile with a non-zero dynamic instruction count.

Injector. The injector is deployed in an `injector.so` library for transient faults and a `pf_injector.so` library for permanent faults. The transient fault injector injects a fault for a single dynamic instruction, whereas the permanent fault injector corrupts all instances of a specified opcode.

B. *Fault Models*

Transient fault model. The main fault model supported by NVBitFI is a transient fault that occurs in the GPU compute pipeline or memory read subsystem. Table II shows the parameters for the transient fault injector. Each parameter is specified in the parameter file as a separate line. The first two parameters indicate the fault type, with the arch state id specifying the type of instructions that should be injected and the bit-flip model indicating the type of bit-level corruption. The last five parameters specify the exact fault to be injected. As described above, the tuple of <*kernel_name, kernel_count, instruction_count*> values specifies the dynamic instruction to inject. The exact error pattern depends on the selected bit-flip model indicated for the bit pattern value.

Users that are concerned about reliability and safety will often select processors with ECC protection for large on-chip memory structures, such as register files and caches. Thus, the remaining vulnerability to faults lies outside of these ECC-protected structures. For faults in the unprotected components

TABLE II
TRANSIENT FAULT PARAMETERS.

Category	Parameter name	Description
Fault types	arch state id	An integer representing the instruction subset to inject 1) G_FP64, FP64 arithmetic instructions 2) G_FP32, FP32 arithmetic instructions 3) G_LD, instructions that read from memory 4) G_PR, instructions that write to predicate registers only 5) G_NODEST, instructions with no destination register 6) G_OTHERS 7) G_GPPR, instructions that write to general purpose and predicate registers, i.e., G_GPPR = all - G_NODEST 8) G_GP, instructions that write to general purpose registers, i.e., G_GP registers = all - G_NODEST - G_PR
	bit-flip model	An integer representing the type of bit-error pattern 1) FLIP_SINGLE_BIT, flip a single bit 2) FLIP_TWO_BITS, flip two adjacent bits 3) RANDOM_VALUE, write a random value 4) ZERO_VALUE, write value 0
Specific target	kernel name	The name of the target GPU kernel
	kernel count	An integer n representing the $(n+1)$th dynamic instance of the target kernel, e.g., 0 indicates the first dynamic instance
	instruction count	An integer n representing the $(n+1)$th dynamic instance of the target instruction
	destination register	A floating-point value [0,1) that determines which general-purpose or predicate register to inject depending on the arch state id
	bit-pattern value	A floating-point value [0,1) that determines the bit-error mask depending on the bit-flip model 1) FLIP_SINGLE_BIT: `0x1<<(32 × value)` 2) FLIP_TWO_BITS: `0x3<<(31 × value)` 3) RANDOM_VALUE: `0xffffffff × value` 4) ZERO_VALUE: mask is same as original register value, so that `XOR` produces `0x0`

TABLE III
PERMANENT FAULT PARAMETERS.

Parameter name	Description
SM id	An integer $0..N-1$ indicating which of the N SMs to inject
Lane id	An integer 0..31 indicating which hardware lanes to inject
Bit mask	An integer representing the XOR bit mask
Opcode id	An integer $0..N-1$ indicating which opcode to inject. Each ISA will have a different set of N opcodes. For example, the Volta ISA contains 171 opcodes.

to propagate to the application outputs, the error propagation must corrupt at least one register. Thus, NVBitFI models faults in terms of their eventual effect on registers.

Specifically, our transient fault models the effect of a transient fault as corrupting a single destination register of a single dynamic instruction. Furthermore, we model the bit-level corruption as either a single or double bit-flip, random corruption, or a zeroing effect. Although the errors from realistic faults will likely have more complicated manifestations, we offer these fault model options as a generalizable fault model. Errors may manifest across multiple registers if the fault affects persistent microarchitectural state. Also, bit-level error patterns are likely dependent on the opcode and instruction inputs. Because these more realistic error effects are difficult to generalize in a parameterized fault model, we offer a simpler, but more generalizable fault model. Future directions include targeting a specified thread, more complex bit patterns, the use of fault models with a greater number of more complex parameters as well as a fault dictionary that is parameterized based on opcodes, input registers, and other state.

Permanent fault model. NVBitFI also supports a permanent fault model, which assumes that the fault affects all dynamic instances of an instruction type. For example, a permanent fault in an ALU would affect the results of all ADD instructions. The NVBitFI permanent fault model takes a simplistic approach to specifying the effects of this type of fault model, with the destination registers of all dynamic instances of a particular opcode being corrupted with the same bit-flip XOR mask.

Table III shows the parameters for permanent faults. The list of parameters is simpler than for transient faults because the dynamic instruction to inject does not need to be specified. Rather, the opcode is specified, and all dynamic instructions with that opcode are injected. Because the permanent fault model is mapped to a physical location on the GPU, the SM and lane parameters indicate which SM and which hardware lane to target for injection. All threads that execute in the target SM and lane are considered for injection.

As with transient faults, we focus on a simple permanent fault model in order to present a generalizable fault model. We realize that realistic permanent faults likely have error effects that are not always dependent on the execution of specific opcodes. Some permanent fault models may be easy to specify. For example, a stuck-at fault on the output of a register file could be emulated as a corruption of the n^{th} bit of every read of the register file. A fault in an ALU could corrupt the result of multiple opcodes that utilize that ALU. Future work includes determining which specialized permanent fault models represent faults that users are likely to find interesting. Also, a fault dictionary approach could be utilized, especially if that fault dictionary is derived from circuit and microarchitectural simulation. We discuss more sophisticated permanent fault models in Section V, but our current model allows analysis of a fault that repeatedly creates errors.

C. NVBit

The `profiler.so` and `injector.so` libraries are built using NVBit [9]. NVBit is a dynamic binary instrumentation framework for NVIDIA GPUs that provides a convenient API for instruction inspection, callbacks to CUDA driver APIs, and injection of arbitrary CUDA functions into any application before kernel launch. NVBit allows instrumentation tools to be created for CUDA programs without requiring the tool writer to have detailed knowledge of the underlying GPU architecture.

By leveraging NVBit, the same tool without recompilation can be used to inject faults into any CUDA executable, which simplifies usage of the fault injector. The dynamic library is attached to a CUDA process using the `LD_PRELOAD` environment variable.

As each dynamic kernel is launched, NVBit will determine if that kernel must be instrumented. If so, the kernel is instrumented and built with just-in-time compilation. That kernel is cached so that a subsequent launch uses the cached compiled version. A kernel that does not need to be instrumented is executed with no modification. This mechanism allows selective and fast instrumentation.

IV. Tool Evaluation

NVBitFI has been applied successfully to a large, commercial autonomous vehicle software (AV) application [22]. This complex application uses many dynamic libraries from several software packages. Thus, fault injection tools that require source code recompilation would struggle to manage the source across these multiple packages. A fault injection tool based on cuda-gdb would not require recompilation or management of source code. However, cuda-gdb is a general debugger that is not designed specifically for fault injection and therefore must maintain a large amount of state for each dynamic kernel. This additional state management imposes a significant performance penalty. Because the AV application is a real-time system, the performance overhead from cuda-gdb would have triggered real-time assertions in the application. Among the fault injection tools that we are aware of, NVBitFI is the only tool that allows fault-injection based evaluation of such a large, real-time system.

TABLE IV
SpecACCEL OpenACC 1.2 benchmark programs.

Program	Description	Static kernels	Dynamic kernels
303.ostencil	Thermodynamics	2	101
304.olbm	Computational fluid dynamics, Lattice Boltzmann Method	3	900
314.omriq	Medicine	2	2
350.md	Molecular dynamics	3	53
351.palm	Large-eddy simulation, atmospheric turbulence	100	7,050
352.ep	Embarrassingly parallel	7	187
353.clvrleaf	Weather	116	12,528
354.cg	Conjugate gradient	22	2,027
355.seismic	Seismic wave modeling	16	3,502
356.sp	Scalar Penta-diagonal solver	71	27,692
357.csp	Scalar Penta-diagonal solver	69	26,890
359.miniGhost	Finite difference	26	8,010
360.ilbdc	Fluid mechanics	1	1,000
363.swim	Weather	22	11,999
370.bt	Block Tri-diagonal solver for 3D PDE	50	10,069

TABLE V
Possible Error Propagation Outcomes.

Outcome	Symptom
SDC	Standard output is different
	Output file is different
DUE	Timeout, indicating a hang (Monitor detection)
	Process crash (OS detection)
	Non-zero exit status (Application detection)
	Application-specific check failed
Masked	No difference detected
Potential DUE	(SDC or Masked) with CUDA error
	(SDC or Masked) with dmesg error

This section uses the SpecACCEL OpenACC v1.2 benchmark [23] to illustrate how NVBitFI can be used to measure error propagation outcomes and associated injection performance overheads. We use the 15 OpenACC applications that are derived from a range of high-performance computing applications, as shown in Table IV. We inject faults into an NVIDIA Titan V GPU as part of a system with an AMD EPYC 7402P 24-Core Processor and 256 GB of memory.

A. Outcome Determination

For each application, we add an SDC checking script to determine if an SDC has occurred. The determination of what constitutes an SDC is both application and user dependent, so SDC checking scripts must always be provided by the user. The SDC checking script should reference a saved, golden version of the standard output and any files created without fault injection. The SpecACCEL package conveniently includes a program-specific checking script with each program, which we use to determine if an SDC has occurred.

The possible outcomes of NVBitFI injections are listed in Table V. SDC outcomes are due to any one of the following conditions: the standard output differs from the golden output, the output file contents differ from the golden file contents, or an application-specific check (e.g., an assertion) failed. DUE outcomes are due to hangs, crashes, and non-zero exit status. If the application passes the SDC check, then the error is masked.

In some cases, a potential DUE can be declared despite symptoms of an SDC or a masked outcome if a non-handled anomaly is recorded by the system. One common anomaly is a non-fatal CUDA message, usually involving a memory access violation. For example, a GPU error that causes a misaligned or non-mapped memory access would normally cause a fatal error for CPU code. However, a similar error on a GPU causes early termination of the current kernel but is otherwise non-fatal to the process unless the CPU code explicitly checks for the error at the end of the kernel. In such cases, the GPU detected the error, but CPU code did not check for it. We consider these cases to be potential DUEs because the program code could be modified to check for and handle the error. In our following results, we count potential DUEs as either SDC or masked.

We conducted a fault injection campaign targeting the SpecACCEL OpenACC v1.2 applications using NVBitFI. Section IV-B discusses the SDC, DUE, and masked outcomes, and Section IV-C shows the overheads incurred by both the profiling and injection steps.

B. Benchmark Fault Injection Outcomes

We conducted three types of fault injection experiments on each of the 15 SpecACCEL programs: (1) transient faults with exact profiling, (2) transient faults with approximate profiling, and (3) permanent faults. For each transient fault experiment, we injected 100 faults for each program. For the permanent fault experiment, we injected one fault for each opcode. We classified the outcomes of each run based on the criteria in Section IV-A. The specific insights we wanted to analyze with these experiments include (1) the differences (between exact and approximate profiling and (2) the manifestation of permanent faults. More injected faults are necessary to tighten the confidence interval for all results [24], [25]. While we show experiments with 100 injections as an example (100 injections provide results with 90% confidence intervals and ±8% error margins), 1000 injections are necessary to obtain results with 95% confidence intervals and ±3% error margins.

Exact versus approximate profiling. As described in Section III-A, approximate profiling is a faster method for profiling but may result in a profile that does not completely match a profile from exact profiling. Figure 2 compares the error propagation outcomes for exact and approximate profiling. The figure shows that although the results do not match completely for exact and approximate profiling, the results for most of the programs appear quite similar. The SDC, DUE, and masked differences are 32.5% versus 37.9%, 4.2% versus 4.5%, and 63.3% versus 57.6%, respectively. These results suggest that approximate profiling produces results that are sufficiently similar to exact profiling to provide sufficient fault injection fidelity. However, the similarity between approximate and exact profiling depends on the application. The NVBitFI

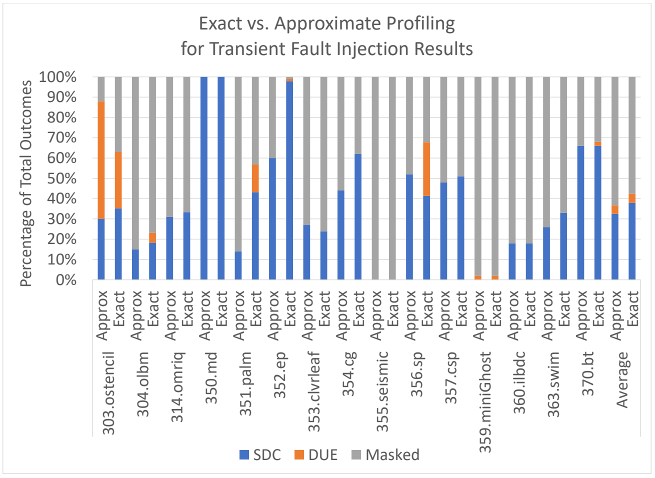


Fig. 2. Comparison of exact and approximate profiling for transient faults.

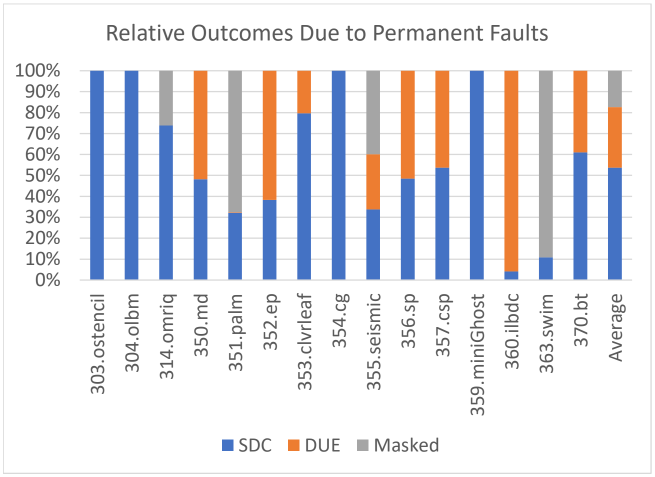


Fig. 3. Relative outcomes for permanent faults.

user will have to consider the trade-off between profiling accuracy and profiling time.

Permanent faults. NVBitFI supports permanent faults based on the fault model described above. Figure 3 shows the relative occurrence of SDC, DUE, and masked outcomes for permanent faults. For each program, 171 runs were conducted with one opcode out of the possible 171 opcodes selected for injection for each run. The outcome of each run is weighted based on the relative number of dynamic instructions for that opcode. For example, if injections into the FADD instruction results in an SDC and accounts for 10% of all program instructions and FMUL results in a DUE and accounts for 20% of all program instructions, then the DUE outcome would be weighted twice that of the SDC outcome. This weighting reflects the greater likelihood that the FMUL instruction is executed and activates the permanent fault.

Compared to transient faults, the permanent faults for our benchmark programs result in more SDC and DUE outcomes. Masked outcomes constitute 57.6% for transient faults but only 17.4% for permanent faults. This result is intuitive as permanent faults are activated multiple times and result in greater error propagation, leading to more SDCs and DUEs.

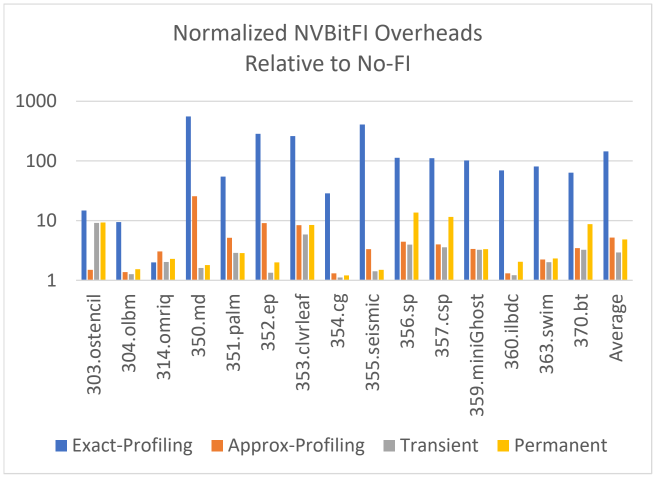


Fig. 4. Execution overheads.

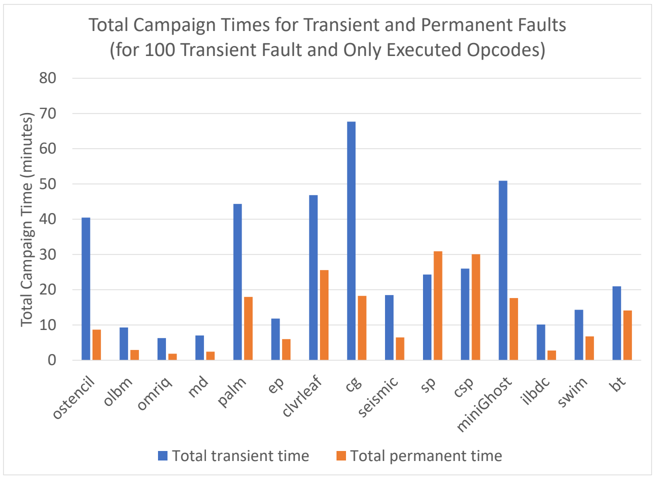


Fig. 5. Total campaign times (assuming 100 transient faults).

C. Benchmark Performance Overheads

Figure 4 shows the overheads for profiling and injection, relative to the runtime of an uninstrumented program. Profiling need only be performed once per application to characterize the set of injection sites. The overhead for exact profiling can be quite large (as much as 558$\times$ for 350.md), especially if the instrumentation causes GPU registers to be spilled to memory. On average, our programs demonstrate an exact profiling overhead that is 28$\times$ more than approximate profiling. If the exact profiling time is unacceptable, approximate profiling offers an attractive alternative.

In contrast to profiling, injection experiments must be performed many times to obtain statistical confidence. For these programs, injection times can vary depending on the amount of instrumentation added to a program and the number of times that instrumentation is executed. The injection times in Figure 4 are the median from the set of 100 injection experiments for each program and fault type. For our programs, transient fault injection slows down program execution by about 2.9$\times$, while permanent fault injection slows down program execution by about 4.8$\times$ on average.

In a testing campaign, transient error experiments must be run enough times to be statistically significant, while permanent fault injection experiments must be run once for each opcode. Furthermore, permanent fault experiments can be skipped for unused opcodes, further simplifying the campaign. Figure 5 illustrates aggregate campaign times for transient campaigns with 100 faults and permanent campaigns that leverage a profile to identify unused opcodes. The number of executed opcodes for our programs ranges from 16 to 41 opcodes per program (out of the total possible 171 opcodes). For these applications, the transient campaigns typically take about twice the time as the permanent campaigns, although the transient campaign can take as much a 5$\times$ longer or be slightly faster.

V. Summary and Future Directions

We have presented the NVBitFI tool for dynamic fault injection into GPUs. The tool is built using the NVBit dynamic binary instrumentation framework and therefore offers a convenient way to conduct a fault injection campaign into GPUs without having to know many details about the GPU. Furthermore, no source code for the target program is required, which not only simplifies the user's job but also allows faults to be injected into dynamic libraries for which no source code is available. Porting the tool to non-Nvidia GPUs would require (a) a binary instrumentation tool on the GPU and (b) porting the handlers from CUDA to something like OpenCL or employing a translator (e.g., HIP [26]) to perform the translation.

Using the NVBit framework, NVBitFI can limit instrumentation needed for fault injection to the dynamic instance of the target kernel. Non-target instances of the same static kernel execute unmodified, thus minimizing the performance overhead of the injection code.

NVBitFI currently supports a transient and a simple permanent fault model. We are considering extensions of the fault model, including the following:

- **Intermittent faults.** The permanent fault model corrupts the destination register of every dynamic instruction of a particular opcode. An intermittent fault model would inject into only a subset of those instructions. The subset can be specified as a random, bursty process.
- **More complex fault models.** Our current fault models can be extended to provide additional flexibility in specifying fault parameters, including (1) corrupting multiple registers, (2) supporting corruption functions beyond the current set of XOR, random, and zero functions, (3) conditioning error effects on the specific opcode, and (4) allowing a permanent fault to affect multiple opcodes.
- **Fault dictionary.** A fault dictionary based on microarchitectural simulation or an analytical model is a specific example of a more complex fault models. A fault dictionary might be useful when a complex fault model is not easily characterized by a set of parameters.

References

[1] "TOP500 LIST - NOVEMBER 2020," https://www.top500.org/lists/top500/list/2020/11/, accessed: 2021-03-27.

[2] "GREEN500 LIST - NOVEMBER 2020," https://www.top500.org/lists/green500/list/2020/11/, accessed: 2021-03-27.

[3] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," in *International Symposium on Microarchitecture (MICRO)*, 2003, pp. 29–40.

[4] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An Architecture-level Fault Injection Tool for GPU Application Resilience Evaluation," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2017.

[5] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding Error Propagation in GPGPU Applications," in *International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2016.

[6] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "GPU-Qin: A Methodology for Evaluating the Error Resilience of GPGPU Applications," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014.

[7] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight Silent Data Corruption Error Detector for GPGPU," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2011.

[8] "NVIDIA CUDA Binary Utilities," https://docs.nvidia.com/cuda/cuda-binary-utilities/index.html, accessed: 2021-03-27.

[9] O. Villa, M. Stephenson, D. Nellans, and S. W. Keckler, "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs," in *International Symposium on Microarchitecture (MICRO)*, 2019.

[10] "Parallel Thread Execution ISA Version 7.1," https://docs.nvidia.com/cuda/parallel-thread-execution/, accessed: 2021-03-27.

[11] M. Stephenson, S. K. S. Hari, Y. Lee, E. Ebrahimi, D. R. Johnson, D. Nellans, M. O'Conner, and S. W. Keckler, "Flexible Software Profiling of GPU Architectures," in *International Symposium on Computer Architecture (ISCA)*, 2015.

[12] "CUDA-GDB," https://developer.nvidia.com/cuda-gdb, accessed: 2021-03-27.

[13] A. Vallero, D. Gizopoulos, and S. D. Carlo, "SIFI: AMD Southern Islands GPU Microarchitectural Level Fault Injector," in *International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2017.

[14] R. Ubal, B. Jang, P. Mistry, D. Schaa, and D. Kaeli, "Multi2Sim: A Simulation Framework for CPU-GPU Computing," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012.

[15] S. Tselonis and D. Gizopoulos, "GUFI: A Framework for GPUs Reliability Assessment," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2016.

[16] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing Soft-error Vulnerability on GPGPU Microarchitecture," in *International Symposium on Workload Characterization (IISWC)*, 2011.

[17] "CUDA Parallel Computing Platform," https://developer.nvidia.com/cuda-zone, accessed: 2021-03-27.

[18] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA Workloads Using a Detailed GPU Simulator," in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[19] M. Wilkening, V. Sridharan, S. Li, F. Previlon, S. Gurumurthi, and D. R. Kaeli, "Calculating Architectural Vulnerability Factors for Spatial Multi-Bit Transient Faults," in *International Symposium on Microarchitecture (MICRO)*, 2014.

[20] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. H. Loh, "Architectural Vulnerability Modeling and Analysis of Integrated Graphics Processors," in *Workshop on Silicon Errors in Logic–System Effects (SELSE)*, 2013.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, , R. Sen, K. L. Sewell, A. Muahammad, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[22] S. Jha, S. Banerjee, T. Tsai, S. K. S. Hari, M. B. Sullivan, Z. T. Kalbarczyk, S. W. Keckler, and R. K. Iyer, "ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection," in *Dependable Systems and Networks (DSN)*, 2019.

[23] G. Juckeland, W. C. Brantley, S. Chandrasekaran, B. M. Chapman, S. Che, M. E. Colgrove, H. Feng, A. Grund, R. Henschel, W. mei W. Hwu, H. Li, M. S. Müller, W. E. Nagel, M. Perminov, P. Shelepugin, K. Skadron, J. A. Stratton, A. Titov, K. Wang, G. M. van Waveren, B. Whitney, S. Wienke, R. Xu, and K. Kumaran, "SpecACCEL: A Standard Application Suite for Measuring Hardware Accelerator Performance," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2014.

[24] B. Nie, L. Yang, A. Jog, and E. Smirni, "Fault Site Pruning for Practical Reliability Analysis of GPGPU Applications," in *International Symposium on Microarchitecture (MICRO)*, 2018.

[25] B. Nie, A. Jog, and E. Smirni, "Characterizing Accuracy-Aware Resilience of GPGPU Applications," in *IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID)*, 2020.

[26] "Whitepaper: Introducing AMD CDNA Architecture - The All-New AMD GPU Architecture for the Modern Era of HPC & AI," https://www.amd.com/system/files/documents/amd-cdna-whitepaper.pdf/, accessed: 2021-03-27.