SwapCodes: Error Codes for Hardware-Software Cooperative GPU Pipeline Error Detection

Michael B. Sullivan, Siva Kumar Sastry Hari, Brian Zimmer, Timothy Tsai, Stephen W. Keckler
{misullivan, shari, bzimmer, timothyt, skeckler}@nvidia.com
NVIDIA

***Abstract*—Intra-thread instruction duplication offers straightforward and effective pipeline error detection for data-intensive processors. However, software-enforced instruction duplication uses explicit checking instructions, roughly doubles program register usage, and doubles the arithmetic operation count per thread, potentially leading to severe slowdowns. This paper investigates *SwapCodes*, a family of software-hardware cooperative mechanisms to accelerate intra-thread duplication in GPUs. SwapCodes leverages the register file ECC hardware to detect pipeline errors without sacrificing the ability of ECC to detect and correct storage errors. By implicitly checking for pipeline errors on each register read, SwapCodes avoids the overheads of instruction checking without adding new hardware error checkers or buffers. We describe a family of SwapCodes implementations that successively eliminate the sources of inefficiency in intra-thread duplication with different complexities and error detection and correction trade-offs. We apply SwapCodes to protect a GPU-based processor against pipeline errors, and demonstrate that it is able to detect more than 99.3% of pipeline errors while improving performance and system efficiency relative to software-enforced duplication—the most performant SwapCodes organization incurs just 15% average slowdown over the un-duplicated program.**

I. Introduction

Datacenters and high performance computer (HPC) systems typically rely on commodity hardware with *error checking and correcting (ECC)* codes applied to large memory structures to increase hardware reliability. While storage ECC detects and often corrects errors in on-chip memory, it leaves coverage holes for errors that occur in pipeline structures such as datapath registers and arithmetic logic. Any thorough protection scheme must avoid such coverage holes. Efficient pipeline error handling is of particular concern for data-intensive processors, such as GPUs, due to the emerging importance of these devices for performance and reliability-conscious users and the large area that these chips devote to compute logic and pipeline registers.

A straightforward, general, and transparent way to provide pipeline error detection is to perform each instruction twice, eventually checking for agreement between the data produced by the original and shadow instructions. Intra-thread instruction duplication places the duplicated instruction pair within the same thread, using regular instructions and existing compiler mechanisms for scheduling and checking [1], [2], [3], [4]. It is applicable to any program without programmer intervention.

The major downside of intra-thread instruction duplication is high performance and energy overheads, stemming from three primary sources. First, software duplication detects pipeline errors using explicit checking instructions. In Section IV, we show that these checking instructions lead to as much as 35% dynamic instruction bloat. Second, instruction duplication increases the program register usage, potentially limiting the parallelism that is exposed in a data-intensive processor. Finally, it doubles the number of arithmetic instructions, which can severely degrade the performance of compute-bandwidth-limited programs.

The compute-class processors that are used for datacenters and HPC systems typically provide error detection [5] or correction [6], [7] for register file storage using ECC codes, leaving the datapath pipeline unprotected. Register file storage ECC cannot check for pipeline errors because encoding takes place after these errors strike, meaning that valid-yet-incorrect codewords are written back to the register file. This paper describes and evaluates a novel error code organization called *SwapCodes* that builds upon instruction duplication while allowing the register file error detection hardware to implicitly detect pipeline errors. SwapCodes is so-named because it swaps in the ECC check-bits from the duplicate instruction to ensure that a single pipeline error cannot affect both the data and check-bits of any codeword. By re-using the register file ECC bits and checking for errors with the ECC decoder, SwapCodes avoids the overheads of explicit instruction checking without adding new error checkers or hardware buffers.

High-reliability systems differ in their efficiency needs and the amount of chip area and design effort they can devote to pipeline error detection. Thus, we present a family of SwapCodes organizations that progressively target the three sources of performance overheads described above with differing design complexities. Our specific contributions when creating SwapCodes are:

- We describe Swap-ECC, a mostly-software approach to leverage intra-thread duplication and the register file error detecting hardware for efficient GPU pipeline error detection with modest hardware changes. We describe two algorithms (SEC-DED-DP and SEC-DP) that maintain storage correction for SEC-DED protected register files while completely avoiding pipeline error miscorrection. SEC-DED-DP works for any SEC-DED code while SEC-DP puts design constraints on the SEC-DED code to lower overheads.
- We describe Swap-Predict, an organization that extends Swap-ECC with selective ECC check-bit prediction to opportunistically improve efficiency. We also describe the innovations required to predict residue codes for the mixed-operand-width GPU MAD instruction.

- We present a detailed evaluation of SwapCodes performance using a modified compiler pass and running on an NVIDIA Tesla P100 compute-class GPU. The simple Swap-ECC organization roughly halves the slowdown of instruction duplication on average, and the most performant Swap-Predict variant incurs just 15% mean slowdown over the un-duplicated program.
- We use gate-level fault injection to quantify the resilience of SwapCodes and demonstrate that SwapCodes detects more than 98.8% of pipeline errors with a standard error-correcting register file and more than 99.3% with an equal-redundancy error-detecting residue code.

II. Background

A. Scope and Focus

We describe and evaluate SwapCodes for a data-intensive GPU-based processor; Figure 1 shows the GPU hardware that this work targets. GPU programming models run thousands of threads that each execute the same code. Code is executed in a unit called a *CUDA kernel*,[1] which is further subdivided into thread blocks called *cooperative thread arrays (CTAs)* and 32-wide bundles of threads called *warps*. These warps execute on compute cores called the GPU streaming multiprocessors (SMs). SMs execute using the single-instruction multiple-thread (SIMT) model, where each 32-thread warp runs in lockstep but control flow can diverge across threads. A hardware-implemented control-flow stack ensures that a warp eventually traverses the execution path of each constituent thread.

There is a limited amount of GPU state, and increasing the state-per-thread can limit the number of threads that concurrently execute, harming performance. This observation leads to two design principles that are respected by SwapCodes. First, SwapCodes does not reserve shadow state per thread (unlike software-enforced intra-thread instruction duplication) so as not to reduce the amount of exposed parallelism. Second, SwapCodes does not introduce new thread-local microarchitectural state (such as a buffer to temporarily hold results), since such state must be replicated for each executing thread, potentially using a large amount of area.

This paper considers only transient errors that affect a single computation; such errors are important to HPC systems as they pose a high risk of silent data corruption [8], [9]. The primary goal of SwapCodes is to protect the logic and pipeline state in the GPU SMs. The SM is important to protect, as it has a large amount of unprotected logic and state and it is heavily utilized for well-tuned applications. GPU-based error injection shows that pipeline errors in the SM cause silent data corruption at the program output up to 20% [10] or 40% [11] of the time. Recent neutron beam testing campaigns show the SDC rates in ECC-protected GPUs to be high for some reliability-conscious users and also demonstrate that software-enforced duplication reduces the overall GPU silent data corruption rate by roughly an order of magnitude [12], [4].

[1]This paper uses NVIDIA GPU terminology but SwapCodes could apply to any GPU with an ECC-protected register file.

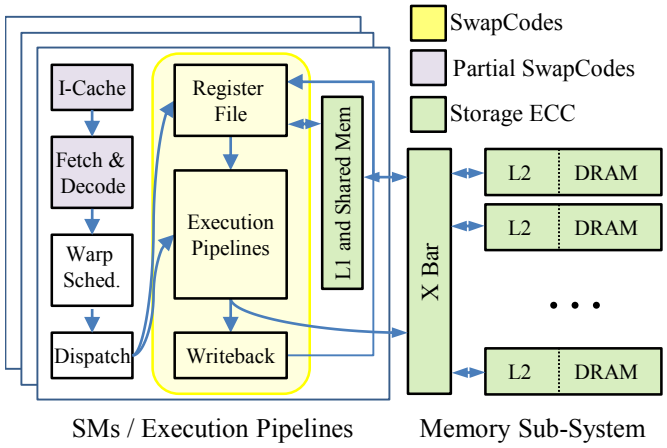


Fig. 1: The sphere of replication targeted by SwapCodes. The main focus is the SM pipeline datapath; structures in the control path get some incidental coverage but we do not focus on such errors. Structures in the memory sub-system must be protected by conventional means (such as storage ECC).

The instruction delivery and memory subsystems are not protected by SwapCodes. These subsystems deal mainly with data transmission (rather than transformation) at a coarse access granularity and consist largely of data buses and SRAM buffers. Thus, these structures can be effectively protected by more conventional techniques such as storage ECC and transmission bus parity [13], and they are not our focus.

B. Error Correcting and Detecting Codes

An error checking and correcting code (ECC) detects and perhaps corrects errors using redundant check-bits that are generated algorithmically from the protected data. A data and check-bit pair is called an ECC word. A valid ECC word whose check-bits are consistent with its data is called a *codeword*, while an invalid pair is called a *non-codeword*. The process of generating a codeword from data is called *encoding*, and the process of detecting errors in a word and (possibly) restoring the original data is *decoding*.

The error detecting and correcting capabilities of a binary ECC code are often characterized by the maximum number of erroneous bits that the code is guaranteed to detect and correct. Compute-class GPU processors conventionally apply a single-bit-error-correcting and double-bit-error-detecting (SEC-DED) code to the register file [7], [14], which if used for detection alone can also function as a triple-bit-error-detecting (TED) code. An error that exceeds the correction capabilities of an ECC code, but is within the detection capabilities, will flag a detected-yet-uncorrected error (DUE). Errors that exceed the detection capabilities of a code can either happen to be be detected (resulting in a DUE) or they can lead to silent data corruption (SDC).

SwapCodes is a general scheme that works with any systematic register file error detecting or correcting code.[2] The pipeline error detection coverage of SwapCodes depends on the error-detecting strength of the underlying ECC code, and part of this

[2]A *systematic* code is one whose data and check-bit locations are fixed at design time; most practical ECC codes are systematic [15].

TABLE I: A qualitative comparison of pipeline error detection alternatives.

	High-Level Duplication	Thread Duplication	Instruction Duplication	Concurrent Check	SwapCodes
Granularity	Process/Kernel/Warp	Thread	Instruction	Operation	Instruction
Sphere of Rep.	Device	Pipeline	Pipeline	Arithmetic	Pipeline
S/W Changes	Program/Runtime	Runtime/Compiler	Compiler	None	Compiler
H/W Changes	None	None	None	Arithmetic	Control Logic
Transparent	No	No	Yes	Yes	**Yes**
Performance Hit	Medium–High	Medium–High	Medium–High	None–Low	**Low–Medium**
Major Issue	Data Duplication	Thread Usage	Performance	Complexity/Scope	**None**

work is to quantify its behavior using different register file error codes. The primary error-correcting code we consider is a Hsiao SEC-DED code [16], which is able to provide triple-bit error detection against pipeline errors using SwapCodes. Register files can also employ a detection-only error code that does not attempt error correction [5]. Therefore, we also consider alternate error-detection-only codes called low-cost residue codes [17], which have been applied to high-reliability register files in the past [18], [19], [20]. Residue codes operate by taking the remainder of the data value divided by some odd checking modulus as the error detecting check-bits. To avoid general division during encoding and decoding, low-cost residue codes rely on checking moduli in the form $A = \left[2^a - 1;\ a \in \mathbb{N}\right]$ [17].

ECC decoders are generally *self-checking checkers* [21] with internal redundancy to properly diagnose latent permanent errors (such as a stuck-at-0 error at the detected-error output). Respecting the synthesis constraints of these self-checking checkers complicates the backend design flow. Accordingly, SwapCodes is designed to re-use the existing register file ECC decoder such that it requires no new error checkers.

C. Pipeline Error Detection

Table I shows a qualitative overview of state-of-the-art techniques for pipeline error detection. High-level duplication schemes (column 1) check for agreement between the original program and a duplicate copy at the process [22], [23], kernel [3], CTA [3] or warp [24], [25], [26] granularity. While conceptually simple, these techniques halve the GPU throughput and they require significant system-level architecture changes. Those that operate at the program granularity modify system calls and I/O, and those that operate at finer granularities often must duplicate memory data, potentially complicating the runtime system and halving device memory capacity.

Inter-thread Duplication. Inter-thread duplication (column 2) replicates the threads within a warp, splitting each $>$16-thread warp in two [27], [24], [26] and checking for errors at the memory boundary. This approach seems attractive for GPUs, as it can leverage the naturally lockstepped execution within each warp for implicit synchronization between threads. Also, inter-thread duplication can use fast inter-thread GPU communication primitives (shuffle instructions) for checking [24], [26]. However, since it doubles the thread count, inter-thread duplication does not work for programs that use more than half the maximum thread-count for each CTA. Furthermore, since inter-thread duplication splits warps, it has difficulty with programs that use intra-warp communication instructions (such as shuffles), programs that use complex multithread instructions (such as the Tensor Core matrix-multiply-and-accumulate [28]), or that use shared memory for intra-warp communication assuming unsplit warps. Because inter-thread duplication does not work for all programs, it is not programmer transparent.

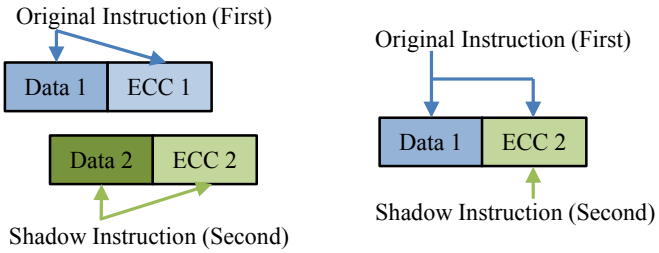
Intra-thread Duplication. Intra-thread instruction duplication (column 3) performs each instruction twice and checks for agreement between the original and shadow instructions. It has been successfully employed for both CPU [1], [2] and GPU-based programs [3], [4]. The most notable advantage of this approach is that it does not change the programming model and it can provide general and programmer-transparent pipeline error detection. However, software-enforced intra-thread duplication uses explicit checking instructions, increases program register usage, and doubles the number of arithmetic operations, potentially leading to significant slowdowns.

Concurrent Hardware Checkers. An alternative to duplication-based pipeline error detection is to employ specialized concurrent checkers (column 4) to vet operations as they execute [29], [30], [31], [32], [33], [18], [19], [34], [35], [36]. Such techniques provide low-latency error detection with less performance overheads than duplication, but they either suffer from limited scope (protecting only a simplified RISC pipeline) or the significant design complexity and costs of protecting each pipeline operation individually. This is problematic for GPUs, which execute a complex instruction set with many specialized instructions—for example, the authors of Argus-G propose concurrent checking for GPUs but note that production SMs would require un-described novel checkers [33].

SwapCodes. The SwapCodes organizations (column 5) seek to leverage the best elements of the pipeline duplication alternatives while avoiding their major limitations. SwapCodes is a hardware-software collaborative organization that leverages software-enforced intra-thread duplication and the existing register file ECC hardware to provide efficient pipeline error detection with modest hardware changes. Some SwapCodes organizations also leverage selective ECC check-bit prediction units to opportunistically improve efficiency while avoiding the complexity and practical concerns of a fully concurrently-checked datapath.

III. SwapCodes for Pipeline Error Detection

The core concept behind SwapCodes is to duplicate each instruction, pairing the data from the original instruction with the ECC check-bits from its shadow. Swapping the data and check-bits from the original and shadow codeword ensures that a single



(a) Intra-Thread Duplication **(b) Swap-ECC**

Fig. 2: Intra-thread duplication maintains a shadow register space for duplicate state. Swap-ECC stores the data of the original instruction with the ECC from the shadow instruction.

TABLE II: The Swap-ECC hardware and software changes.

Structure/Program	Swap-ECC Changes
Backend Compiler	Add an intra-thread duplication pass.
Backend Compiler	Swap-ECC-aware scheduling.
ISA Meta-Data	Add a 1b data write enable.
Register File	Add a data write enable and muxes for move propagation.
Error Reporting (Storage Correction)	Augmented error reporting to separate storage from pipeline errors.

pipeline error cannot affect both the data and check-bits of any register. This strategy allows the register file ECC hardware to detect execution errors in structures such as pipeline registers or arithmetic logic. We describe a family of SwapCodes organizations below that trade off performance and design complexity. Section III-A describes a basic mostly-software SwapCodes organization we call Swap-ECC. Section III-B describes two error reporting algorithms that Swap-ECC can use to maintain SEC-DED storage correction without risk of misdetecting pipeline errors. Section III-C describes Swap-Predict, organizations that optimize Swap-ECC with selective ECC check-bit prediction units to opportunistically improve efficiency.

A. Swap-ECC: SwapCodes with Intra-Thread Duplication

SwapCodes with intra-thread duplication employs swapped codewords within each thread to detect both register file storage and pipeline errors. Figure 2 illustrates this organization, which we call Swap-ECC, contrasting it with software-enforced instruction duplication. Intra-thread instruction duplication executes each instruction twice, maintaining a shadow register space for the duplicate state. Swap-ECC also executes each instruction twice, but it overwrites the ECC produced by the original instruction with the ECC from the shadow instruction.

Table II shows the hardware and software changes needed for Swap-ECC support. Swap-ECC requires the compiler to perform intra-thread instruction duplication. The register file must support the ability to write only the ECC check-bits back for a shadow instruction. The ISA must be extended with a 1b meta-data flag to identify shadow instructions for masked write-back. Finally, end-to-end move propagation (described later) may require some registers and multiplexers to propagate the ECC check-bits.

(1) Un-Duplicated Code

```
ADD R1,R1,R2
```

(3) Swap-ECC

```
ADD      R3,R1,R2 //orig.
ADD.ECC R3,R1,R2 //shad.
```

(2) Intra-Thread Duplication

```
    ADD R1,R1,R2 //orig.
    ADD R3,R3,R4 //shad.
    ISETP.EQ P1,R1,R3
@P1 BRA,U `(.L_1) //check
    BPT.TRAP 0x1
.L_1:
```

Fig. 3: An example of intra-thread duplication and Swap-ECC.

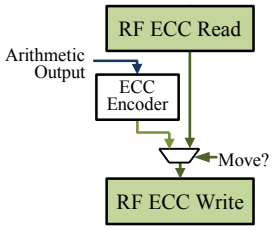


Fig. 4: Swap-ECC need not duplicate move instructions if the full swapped ECC codeword is propagated back to the register file.

Scheduling. Swap-ECC is able to use compiler-defined scheduling without any additional hardware or explicit synchronization. The original and shadow Swap-ECC instructions write different parts to the same register and have a write-after-write dependence. This dependence is meaningful because it ensures that a following instruction cannot read a register until both its data and ECC are produced. The dependence also prevents the GPU register allocator from inserting instructions with the same source and destination registers; this is because the source and destination registers are shared between the original and shadow instruction with Swap-ECC, making single-register accumulation impossible. Careful compiler design is required to ensure that dead code elimination does not remove the apparently-dead original instruction.

Figure 3 shows code generated for executing and checking a single add instruction with intra-thread duplication and Swap-ECC. Intra-thread duplication maintains shadow register state, inserting code to check for equivalence before memory or control-flow instructions. This example assumes that some non-duplicated instruction, such as a memory store, uses the destination register in following code. Swap-ECC replicates the add instruction (without single-register accumulation), writing only the ECC check-bits with the shadow instruction. Following uses of the Swap-ECC destination register (R3) must wait for the shadow instruction to complete due to the write-after-write dependency. Checking then occurs for R3 during any register file read without explicit code.

Register Bypassing. Because GPUs focus on thread-level parallelism, CPU optimizations such as register bypassing are less critical or even unnecessary. With many simultaneously executing threads there is little or no performance benefit to register bypassing—there are always threads ready to be executed and peak throughput can be maintained despite the register writeback latency. Accordingly, this work assumes that the GPU pipeline does not employ register bypassing. This is consistent with GPGPUSim, which is inspired by the NVIDIA Fermi architecture [37]. We discuss the implications

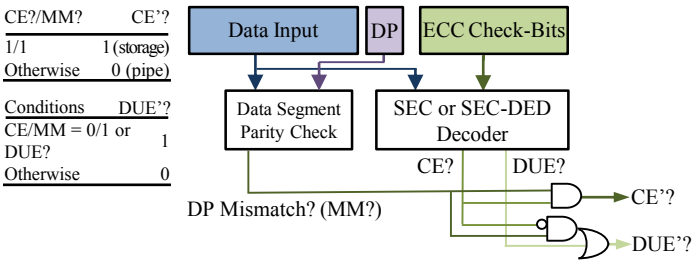


Fig. 5: SEC-DED-DP and SEC-DP use a data-parity check to avoid miscorrecting single-bit compute errors.

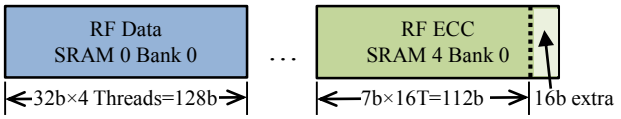


Fig. 6: GPU vector register files are implemented SRAM arrays. One possible register file organization uses 128b-wide SRAMs; such an organization may have room for 40b SEC-DED-DP codewords due to internal fragmentation in the ECC SRAM.

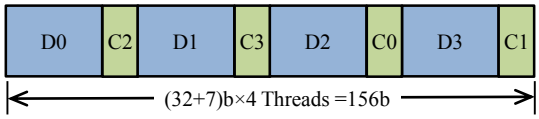


Fig. 7: Codeword layout within the register file can space the data and ECC check-bits so as to avoid any problematic SEC-DP double-bit storage errors. D: data-bits; C: check-bits.

for theoretical GPUs with register bypassing in Section VI.

Debugability and Interrupts. As Figure 2b illustrates, Swap-ECC writes the full destination register with the original instruction but only the ECC with the shadow. This design maintains the assembly-level debugability of Swap-ECC and it simplifies the handling of GPU-based interrupts. Because the original instruction produces a valid output codeword during error-free operation, an intervening interrupt can read the value in the register without triggering a false-positive DUE. The most obvious example of a GPU-based interrupt that reads a register at arbitrary points is assembly-mode `Cuda-GDB`.

Move Propagation. One small optimization we employ for Swap-ECC is to propagate the full swapped ECC codeword back to the register file upon register moves, as shown in Figure 4, rather than decoding the moved register and encoding it for writeback after it flows through the pipeline. Such move propagation is a reasonable design choice as it may also reduce register movement power; if it is employed in GPUs already then Swap-ECC enjoys its benefits without hardware changes. This optimization avoids the need to duplicate register moves; a generalization of this concept (Swap-Predict) for other operations is presented in Section III-C.

B. Maintaining Storage Error Correction with Swap-ECC

SwapCodes works without any changes to the ECC decoder or error-reporting subsystem when employed with an error-detecting register file. Retaining the ability to correct storage errors without risking miscorrection for pipeline errors is more challenging, however. No issues arise if the original instruction is hit by a pipeline error, but if the ECC-producing shadow instruction suffers from a single-bit pipeline error, it will erroneously miscorrect the same bit in the actually-error-free data unless further steps are taken. We describe two algorithms below (SEC-DED-DP and SEC-DP) that maintain storage correction for SEC-DED protected register files while completely avoiding pipeline error miscorrection. SEC-DED-DP works for any SEC-DED code while SEC-DP optimizes parity-bit based SEC-DED to possibly lower overheads. We envision that the designer will select between SEC-DED-DP and SEC-DP based on the SEC-DED code in use and the register file design, and we describe their trade-offs below.

SEC-DED with Data Parity (SEC-DED-DP). *SEC-DED with Data Parity (SEC-DED-DP)* adds an extra *data-parity bit* to the code, generating this parity bit from only the data-bits (excluding the check-bits). Crucially, the data-parity bit is *not* swapped, but rather it is generated by the original instruction. SEC-DED-DP is able to distinguish between a single-bit storage error, which is corrected, and a single-bit compute error, which is flagged as a DUE. The intuition behind SEC-DED-DP is that a single-bit error in the shadow instruction will manifest as a miscorrection-causing error in the ECC check-bits of the swapped codeword, but the data segment will remain untouched. In contrast, a single-bit storage error will corrupt the data in a manner that is detectable via the data-parity bit. SEC-DED-DP allows data error correction *only* upon a mismatch between the data parity bit and the data segment; otherwise it raises a DUE.

This approach maintains the storage correction capabilities of the un-augmented SEC-DED code, while providing triple-bit error detection against compute errors and completely avoiding the risk of miscorrecting compute errors. SEC-DED-DP can augment any underlying SEC-DED ECC without code changes or changes to the decoder hardware. Rather, it augments the error reporting procedure as shown in Figure 5. The *CE?* signal indicates that *data* error correction was attempted[3] and *DUE?* indicates a detectable-uncorrectable error.

GPU vector register files are implemented with SRAMs [38], and a reasonable organization is to use a separate SRAM to hold the check-bits for many threads. Figure 6 shows a register file that uses 128b-wide SRAMs for both data and ECC. This register file organization already has 16b of internal fragmentation per 16 threads, so the data parity bit might be stored without introducing additional redundancy. Using a separate SRAM for check-bits may be desirable in GPUs that do not always require ECC protection, as it allows the ECC SRAM to be power gated. If the GPU register file is not organized in this way (e.g., if 156-bit wide SRAMs are used to store both data and ECC), then SEC-DED-DP requires an extra bit per register, an increase of 2.6%. Alternatively, a small code change can provide SEC-DED levels of protection without any added redundancy. We describe such a scheme below.

[3]Note that CE? should not raise if the decoder corrects an error in the ECC check-bits and not the data; this case only arises for storage errors (not pipeline errors) and the decoder will operate as intended.

SEC with Data Parity (SEC-DP). An alternative to adding an extra data-parity bit for SEC-DED-DP is to downgrade the register file error correcting code to a SEC code (using only 6 bits of ECC per 32b register) and then append the data parity bit to fit within the original redundancy of SEC-DED ECC. We call this scheme SEC-DP; its error reporting logic is the same as SEC-DED-DP (Figure 5). It is well known that adding a *full* parity bit to a SEC code (with parity generated across both the data and check-bits) provides SEC-DED protection [15]. Accordingly, the SEC-DP data-parity bit can achieve *almost* double-bit error detection, and we describe how its double-bit coverage holes can be closed through careful register file codeword layout.

The double-bit errors that are missed by SEC-DP are those that affect a data-bit and an ECC check-bit—if such an error occurs, SEC-DP will miscorrect an error-free data bit, potentially causing silent data corruption. Fortunately, the problematic data-bit and check-bit error pattern does *not* reduce the SEC-DP pipeline error coverage, since by construction a single pipeline error affects *only* the data or the ECC check-bits. The problematic pattern can be avoided for storage errors through careful codeword layout within the register file. Because GPU vector register files store multiple codewords to the same SRAM, it is possible to physically separate the data and check-bits such that a single event is highly unlikely to affect a data and ECC bit within any codeword, as shown in Figure 7. Therefore, SEC-DP offers an alternative Swap-ECC organization that uses careful code design and register file layout to achieve the same pipeline error coverage as SEC-DED-DP while fitting in the redundancy of SEC-DED ECC.

C. Swap-Predict: Swap-ECC with Check-bit Prediction

Swap-ECC provides a natural organization to leverage specialized ECC prediction units in the datapath to opportunistically avoid shadow instructions for common operations. Figure 8 shows this organization, which we call Swap-Predict. Swap-Predict introduces a reduced-width ECC prediction pipeline alongside the regular datapath. The name Swap-Predict is inspired by check-bit prediction techniques (such as parity prediction [39]) that generate the correct check-bits following a logical transformation for concurrent checking without wholesale duplication. Check-bit prediction is not speculative, unlike similarly-named microarchitectural prediction structures.

GPUs execute a complex instruction set with many variants, including specialized graphics instructions that can be repurposed for compute workloads [40]. The major advantage of Swap-Predict over prior concurrently-checked datapath organizations is that Swap-Predict need not protect all operations. Rather, Swap-Predict opportunistically accelerates common operations while relying on Swap-ECC to check difficult-to-predict or rarely-used instructions. Also, unlike prior concurrent checking organizations, Swap-Predict requires no new error checkers.

Using binary instrumentation, we observe that the most-often used arithmetic instructions include fixed-point addition and fixed-point multiply-add (MAD). Fixed-point addition and multiply can be predicted using ECC-specific circuits.

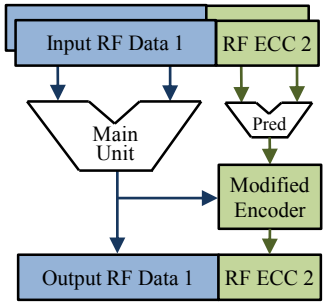


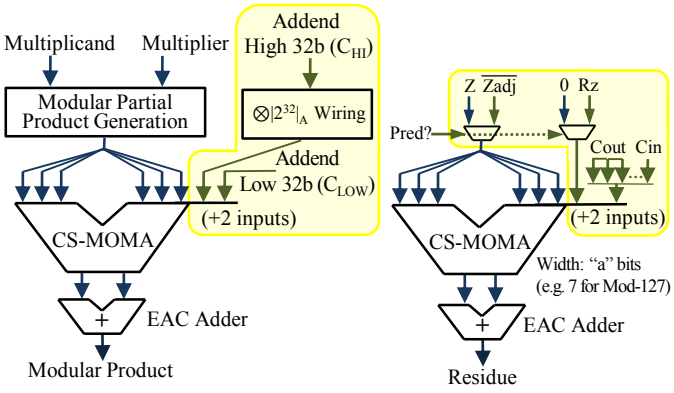
Fig. 8: Swap-Predict selectively avoids the need for duplication using specialized ECC prediction units for common operations. These units form the correct check-bits for the result of an operation even upon an error in the arithmetic pipeline.

Low-cost residues are the most popular and best-studied code for arithmetic prediction, and residue predictors are used both in prior papers [18], [19], [32], [33] and mainframe computers [29], [30], [32]. Accordingly, we focus on Swap-Predict with residue codes in this work but discuss the possibility of using SEC-DED codes later in Section VI. Prior residue checking approaches assume a single uniform operand length across all inputs and outputs, but GPU MAD can mix 32b and 64b inputs and it produces 64b outputs. To demonstrate the full potential of Swap-Predict for GPU datapath protection, we present a case study of residue prediction and describe the innovations required to apply it to GPU MAD below.

Case Study: GPU Residue Code Prediction. Residue codes are well suited for check-bit prediction because they are closed under modular arithmetic and can be added and multiplied directly [17]. Low-cost residues use a modulus that is one less than a power-of-two ($A = \lceil 2^a - 1;\ a \in \mathbb{N} \rceil$) to craft efficient arithmetic units from the following building blocks. A carry-save multi-operand modular adder (CS-MOMA) adds many inputs, propagating each carry-out as the carry-in to the next computation and outputting the result in the redundant carry-save format. A logarithmic-delay CS-MOMA uses a reduction tree of constant-delay end-around-carry carry-save adders [41]. An end-around-carry carry-propagate adder (EAC adder) adds two numbers, incrementing the end result if there is a carry-out. An EAC adder can be crafted from a parallel prefix adder using an additional level to internally re-propagate the carry-out signal [42]. The low-cost residue of an N-bit number can be generated by adding $\frac{N}{a}$ non-overlapping bit-slices of the number with a CS-MOMA and an EAC adder [17], [43]. Residue addition can be performed with an a-bit EAC adder. Residue multiplication uses a modified partial product generation algorithm, an a-wide, a-deep CS-MOMA, and an a-bit EAC adder [41], [42].

GPUs heavily use multiply-add (MAD), which has sometimes-wider-than-32b inputs—a full 32b MAD multiplies two 32b operands and adds a 64b addend. Problematically, instead of having an input residue for the full 64b addend ($|C|_A$), the register file gives residues for the two 32b halves ($|C_{HI}|_A$ and $|C_{LOW}|_A$).[4] This makes existing residue

[4]We use the established notation [17] $|x|_A$ to denote x modulo A.



(a) Modified MAD **(b) Modified Encoder**

Fig. 9: A residue arithmetic unit for fixed-point multiply-add with mixed-width operands, and a modified encoder to generate 32b residues with a sometimes-wider-than-32b datapath. CS-MOMA: a carry-save multi-operand modular adder tree; EAC Adder: an end-around-carry carry-propagate adder. $\overline{\text{Zadj}}$ is the bitwise inverse of Zadj.

TABLE III: Handling Cin and Cout in Figure 9b.

Cout	Cin	Signal	Adjustment
0	0	000…0	+0
0	1	000…1	+1
1	0	111…0	−1
1	1	111…1	−0

prediction algorithms inapplicable to the operation. To solve this problem, we derive the proper full input residue from the two partial residues using Equation 1, where $\oplus$ and $\otimes$ denote low-cost residue addition and multiplication, respectively.

$$C = C_{HI} \times 2^{32} + C_{LOW}$$
$$|C|_A = |C_{HI}|_A \otimes \left|2^{32}\right|_A \oplus |C_{LOW}|_A \quad (1)$$

Correction from partial addend residues to the full value of $|C|_A$ entails the residue multiplication of $|C_{HI}|_A$ with $\left|2^{32}\right|_A$ and addition with $|C_{LOW}|_A$. Fortunately, $\left|2^{32}\right|_A$ is a perfect power-of-two for all low-cost residues, making this computation trivial—the low-cost residues with moduli 3, 7, 15, 31, 63, 127, and 255 have corresponding correction factors of 1, 4, 1, 4, 4, 16, and 1. Thus, no correction is needed for the full residue when A is 3, 15, or 255, and the correction for other moduli can be implemented with wiring. Figure 9a shows a modular multiply-add unit with addend correction highlighted in yellow.

Residue arithmetic produces the residue of the full output, but it does not split this residue into the constituent 32b words that are written back to the register file. We address this issue by modifying the residue encoder as shown in Figure 9b. With these modifications, the encoder now serves a dual purpose—it either encodes the instruction output without check-bit prediction (Pred? = 0), or it *recodes* the output into 32b codewords that are written back to the register file (Pred?=1). For predicting operations with 64b outputs, Zadj is set to the 32b output segment that is *not* being written back at a given time; it is subtracted out to adjust the full predicted residue (Rz). A second level of adjustment supports carry-out and carry-in bits. Low-cost residues are encoded with a double zero, such that the proper adjustment can be formed as shown in Table III by adding a residue whose bottom bit is set to the carry-in with every other bit set to the carry-out signal.

IV. SwapCodes Evaluation

Our goal is to evaluate the error coverage, performance, power benefits, and hardware overheads of SwapCodes in a GPU-based system. We describe our prototypes and methods of investigation and then present their experimental results below.

A. Experimental Methodology

Modified GPU Runtime and Compiler. We modify NVIDIA's production backend compiler (`ptxas`) to perform software-enforced intra-thread instruction duplication. Duplication is performed by doubling each arithmetic operation with a shadow register space for duplicate instructions. The values produced by the original and shadow instruction streams are checked for equivalence before any control-flow instruction, memory operation, or non-duplication-eligible instruction (e.g., atomic operations). This compiler pass is similar to the *Base-DRDV* algorithm from [4], and it is heavily inspired by similar work on CPU-based systems [1], [2].

We develop specialized backend compilers that emit the code that a SwapCodes-enabled GPU would run. For instance, the Swap-ECC backend compiler does not have checking code, and it respects the behavior and instruction dependencies described in Section III-A. Swap-Predict builds upon the Swap-ECC compiler, but it does not need to duplicate predicted instructions. We do not expect the SwapCodes hardware changes to dictate the system clock or require any additional pipeline stages. Therefore, we directly observe the performance and power of the SwapCodes variants by running the SwapCodes proxy programs on GPU hardware. We separate out several Swap-Predict organizations in our evaluation to show the effect of different ECC predictors.

Data Collection Methodology. We use the Rodinia 2.3 benchmark suite [44], a DOE miniapp ("SNAP" [45]), and matrix multiplication from the CUDA SDK [46] for evaluation.[5] To measure the performance of each resilience scheme, we recompile each workload with the appropriate backend compiler and measure performance directly on a 16GB PCIe-attached NVIDIA Tesla P100 GPU [7]. We obtain the GPU run time by first executing several warm-up runs and then extracting and analyzing a GPU execution trace (from `nvprof --print-gpu-trace`) with the kernel launch times and durations.[6] We exclude the CPU time and the time spent copying data between the GPU and host, because this should not change across the duplication approaches.

We use `nvprof --system-profiling on` to measure the power usage of each technique. `nvprof` averages power readings over roughly 50 millisecond windows across the full application—given that kernel run times are typically shorter than this window and many benchmarks utilize the

[5]Streamcluster and srad v1 are omitted due to compilation failures, and nn due to a runtime failure.

[6]We observe little run-to-run variability in GPU run time estimates so we do not present confidence intervals for performance.

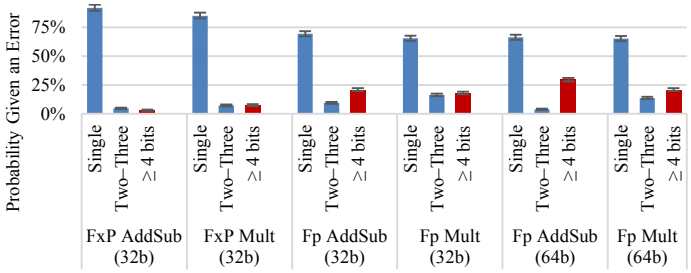


Fig. 10: The severity and pattern of unmasked transient errors in different arithmetic units, with 95% confidence intervals. Three error patterns are shown, in increasing order of error coding complexity. Errors with ≥4 bits (highlighted in red) are the only category with SDC risk using SwapCodes with SEC-DED ECC. FxP: fixed-point; Fp: floating-point.

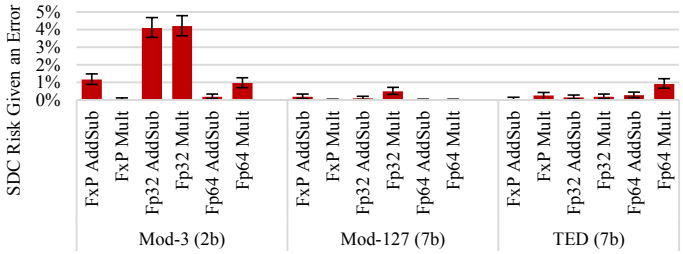


Fig. 11: The pipeline error coverage of all SwapCodes variants using different error detecting codes, with 95% confidence intervals.

GPU for only a small part of the application, most of the power readings contain little or no GPU activity. We estimate the active GPU power using the 90th percentile power reading (which is likely to be a mostly-active power estimation window) and make rough energy estimates assuming that the GPU power stays constant at this active power during each kernel execution.

GPU Binary Instrumentation. We use SASSI-like [47] GPU binary instrumentation tools to observe program behavior and extract arithmetic values for more accurate error injection. Our binary instrumentation tools include:

- A duplicated code-mix profiler that accepts metadata from our modified backend compiler to classify each instruction and to examine the amount of checking code in the software-only baseline.
- An arithmetic value tracer to extract realistic data inputs for gate-level error injection.

Hardware Models and Error Injection. We estimate the hardware overheads of SwapCodes (especially Swap-Predict) using Verilog designs synthesized with the Synopsys toolchain [48] with a 16nm industrial technology library. Automatic register retiming is used to target a two-stage pipeline with a 2GHz clock (assuming 50% margin for uncertainty and unmodeled control circuitry), which is an efficient operating point for the multiply-add unit. Double-precision floating-point units are synthesized with a similar two-stage pipeline, but with a halved 1GHz clock.

We perform gate-level error injection on the synthesized arithmetic units to investigate the error coverage of SwapCodes. The Hamartia framework [49] is used to flip the output of a single gate or flip-flop per injection to estimate the effects of transient errors in logic and pipeline state. We inject an unmasked single-event error into each unit with 10,000 input pairs—that is, for every input pair, we randomly inject single-event errors until one corrupts the unit output. We do not inject errors in the SM control circuitry but note that the majority of the datapath area is devoted to the arithmetic units and prior work has shown control errors to generally result in detectable crashes, not silent data corruption [50], [51].

The severity of arithmetic errors depends on the data inputs to each unit. Accordingly, we use binary instrumentation to extract representative input streams for error injection. The tool traces the inputs of the different arithmetic operations that we consider; to bound the trace size, we trace only the Rodinia 2.3 programs, target the 2048 lowest-numbered threads on the GPU, and halt after 100,000 instructions. We then draw 10,000 random input pairs across the traces of all workloads for error injection into each operation.

B. Pipeline Error Detection Coverage

To quantify the behavior of SwapCodes using different error codes, we inject gate-level errors into six pipelined arithmetic units synthesized with a 16nm industry technology library. Figure 10 shows the error patterns we observe at the output of the arithmetic units. The errors are classified into three patterns in increasing order of SwapCodes detection complexity using a SEC-DED protected register file. SwapCodes can provide triple-bit error detection against pipeline errors using this ECC code. Errors with greater than three bits in error are not guaranteed to be detected, and such errors appear at the output of some units—especially 64b floating point units—roughly 25% of the time. Despite the lack of detection guarantees, we quantify the SDC risk of SwapCodes using several ECC codes below and find it to be small.

Figure 10 shows that the majority of unmasked transient errors affect only a single bit. This makes intuitive sense—any error that affects an output register will affect a single bit, as will any error in a sub-structure that shifts or buffers an internal signal without transformation. Buffers are common for the least-significant bits in the final pipeline stage, since these arithmetic units proceed in a least-to-most significant order and the least-significant output bits are often completely determined before the final stage. Also, shifters are relatively prevalent in the floating-point arithmetic units, which re-normalize the output of the operation to be IEEE-754 compliant using a series of shifters and incrementers [41].

Figure 11 shows the SDC risk when using SwapCodes to detect an unmasked pipeline error with various error codes. SDC risk denotes the probability that a pipeline error in a duplication-eligible instruction goes undiagnosed—thus, a system with no protection has 100% SDC risk and a system using software-enforced duplication has 0% SDC risk. For operations with 64b outputs, we consider an error detected if *either* constituent output register produces a DUE. SwapCodes generally has little SDC risk with all considered codes; even a Mod-3 residue code pushes the SDC risk to $<5\%$, meaning a >20x pipeline error rate reduction despite only having 2b

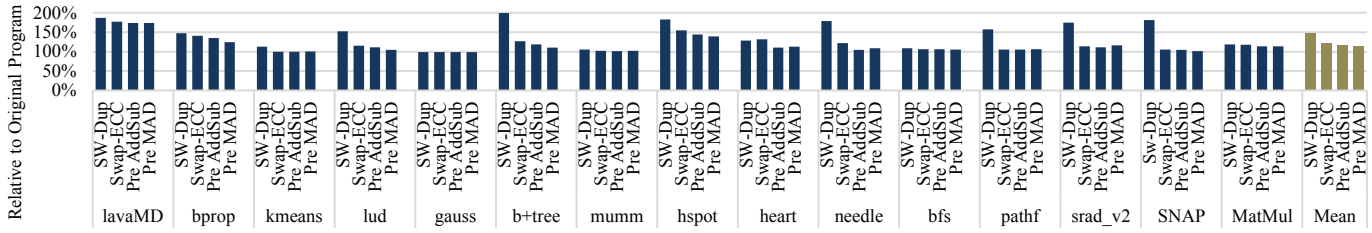


Fig. 12: The performance of intra-thread instruction duplication and SwapCodes running on an NVIDIA Tesla P100 GPU. The order of Rodinia programs is the same as in Figure 13. SW-Dup: Software-enforced intra-thread instruction duplication.

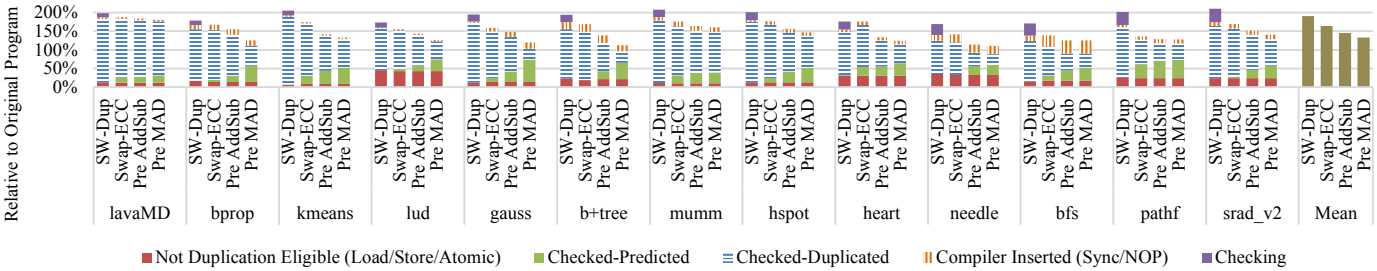


Fig. 13: The dynamic instruction bloat of intra-thread instruction duplication and the SwapCodes variants, measured through binary instrumentation. Programs are ordered by increasing checking bloat. SW-Dup: Software-enforced intra-thread instruction duplication.

redundancy per register. Mod-127 residue codes have the strongest error detection capabilities, with a worst-case 95% confidence interval SDC risk of just 0.7%. SEC-DED protected register files provide triple-bit pipeline error detection using SwapCodes, and the TED code shows the next-best performance with an upper bound of 1.20% SDC risk. Neither the effects of a pipeline error nor the behavior of the register file ECC decoder depend on which SwapCodes variant is used. Thus the results from Figure 11 hold for both Swap-ECC and Swap-Predict.

C. Performance, Code Properties, and Power

Performance and Code Properties. Figure 12 shows the performance of the SwapCodes variants and intra-thread instruction duplication. Two Swap-Predict organizations are shown—one with fixed-point add/subtract prediction ("Pre AddSub"), and a more aggressive variant that predicts both fixed-point add/subtract and multiply/MAD ("Pre MAD"). We also use a GPU binary instrumentation tool to further investigate the code properties of each technique. Figure 13 quantitatively evaluates the dynamic instruction bloat required for each Rodinia 2.3 benchmark. Instructions are classified into those that are not duplicated by construction (bottom), those that are predicted and not duplicated (second to bottom), those that are duplicated (third/horizontal stripes), compiler-inserted control instructions (fourth/vertical stripes), and checking instructions (top).

The arithmetic mean slowdown for intra-thread instruction duplication is 49%, with a worst-case slowdown of 99% (b+tree). Swap-ECC reduces the mean slowdown to 21% (worst-case 78%, lavaMD) by eliminating explicit checking code, avoiding the need for shadow register space, and propagating moved registers without a duplicate instruction. On average, Swap-ECC reduces the total dynamic instruction bloat from 91% to 63%. Eliminating checking code saves the 11–35% redundant instructions (relative to the un-duplicated program) that are required for software-enforced intra-thread duplication. The Rodinia 2.3 programs are sorted in both figures according to their checking code bloat; many programs that require the most checking code (such as srad_v2, pathf, and needle) see large performance improvements over intra-thread duplication. Swap-ECC shows a small number of predicted (and not duplicated) instructions. These are the moves that are not duplicated due to end-to-end move propagation.

The two Swap-Predict variants avoid the need to duplicate some arithmetic instructions, reducing the mean slowdown to 16% (Pre AddSub, worst-case 74% for lavaMD) and 15% (Pre MAD, worst-case 74% for lavaMD). Through prediction, Swap-Predict reduces the total dynamic instruction bloat to 45% (Pre AddSub) and 33% (Pre MAD). MAD prediction reduces the average number of duplicated instructions significantly, but Figure 12 shows that it only reduces the average slowdown by 1% relative to AddSub prediction. In many cases Swap-Predict with AddSub prediction already approaches the same performance as the un-duplicated program, meaning that the compiler can effectively schedule the additional MAD operations and they affect the performance little. The largest performance improvements from MAD prediction come from hotspot, backprop, and b+tree, all of which progressively benefit from more aggressive check-bit prediction.

The worst case slowdown of SwapCodes is for lavaMD, where even Swap-Predict (Pre-MAD) runs 74% slower than the un-duplicated program. Figure 13 shows little checking code in this program, so Swap-ECC offers little benefit over software-enforced duplication. Discussion in Section VI shows lavaMD to be floating-point MAD limited and it describes future work to accelerate programs like it.

Power. Figure 14 estimates the GPU power and energy used by Swap-ECC for the two high-utilization workloads with 95% confidence intervals across 66 program runs. Both intra-thread duplication and SwapCodes have some power overheads, in the worst case showing 15% increased GPU power. There

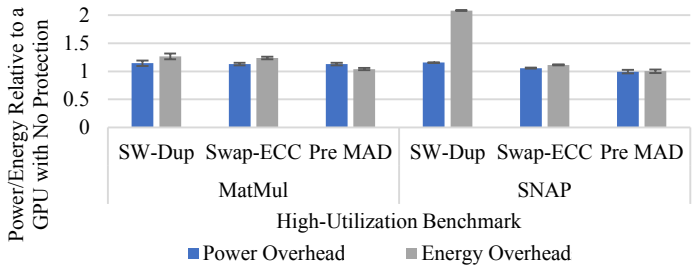


Fig. 14: The estimated GPU power and energy overheads of software duplication and SwapCodes for the two benchmarks with the highest GPU utilization.

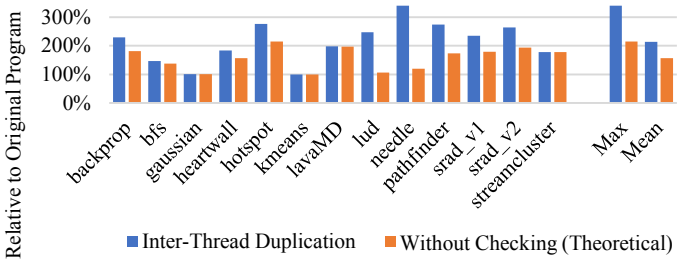


Fig. 15: The performance of inter-thread instruction duplication running on an NVIDIA Tesla P100 GPU.

is a relatively small power difference between intra-thread duplication and SwapCodes, meaning that the energy benefits of the SwapCodes variants are directly proportional to their performance improvements. The more notable example of this is SNAP, which suffers from $>80\%$ performance degradation using intra-thread duplication, leading to >2x energy increases. In contrast, the slowdown of SNAP with Swap-ECC is 6%, leading to a worst-case energy increase of only 11%, and Swap-Predict with MAD prediction has nominal performance and energy overheads.

D. Hardware Overheads

The hardware-software collaborative nature of SwapCodes makes its hardware modifications modest and its overheads manageable. Swap-ECC requires a single-bit ECC write enable and a 1b ISA field to control masked ECC register writes. GPU register files use banked SRAM arrays with a per-lane write-enable for control divergence [38], and masked ECC-only register writes can be added to the GPU register file with little additional complexity. GPUs also have a mutable ISA that changes each generation, relying on just-in-time compilation to maintain compatibility.

To estimate the logic overheads of the SwapCodes variants, we synthesize a Verilog model of the necessary hardware components. Table IV gives the circuit area estimates (in NAND2 gate-equivalents) and the relative overhead of the SwapCodes logic. Overheads are given relative to the original hardware structure that is augmented or predicted.

End-to-end move propagation (Figure 4) is a reasonable way to perform moves in a GPU system. If moves are not performed this way, the mechanism requires some pipeline registers and multiplexers to propagate the ECC through the datapath. The modified error reporting procedures for Swap-ECC with storage error correction also introduce a small amount of supplementary logic. The overheads of these components are small relative to the SEC-DED decoder—taken together, they use roughly 50% the area of the decoder.

TABLE IV: The logic overheads of SwapCodes.

Unit	Bits	Pipe Stages	Flip-Flops	Area (NAND2)	Area Overhead
Original Data Path					
Add	32	1	96	715	-
MAD	32+64	2	513	9941	-
SECDED Dec.	7	0	0	296	-
Mod-3 Enc.	2	1	34	587	-
Mod-127 Enc.	7	1	39	392	-
Swap-ECC Modifications (Overheads Relative to SEC-DED Decoder)					
Move-Propagate	7	0	14	81	+27.39%
SEC-(DED)-DP	2	0	0	67	+22.65%
Swap-Predict Residue Code Prediction Circuitry					
Add	2	1	6	42	+5.91%
Add	7	1	21	154	+21.57%
MAD	2	2	12	98	+0.98%
MAD	7	2	49	584	+5.87%
Mod-3 Enc.	2	1	71	1016	+108.84%
Mod-127 Enc.	7	1	81	861	+119.86%

Swap-Predict requires error-code-specific predictors and modified encoder circuits; we implement the changes required for residue code prediction with the "Pre MAD" organization. The Swap-Predict circuits are efficient, representing only a fractional increase in the datapath area. The relatively large and heavily utilized multiply-add unit can be predicted especially efficiently, with $<1\%$ area overheads for Mod-3 MAD prediction. The modified residue encoder shows the largest relative hardware overheads, but its absolute size is small such that this overhead should be minor relative to the datapath as a whole.

There is some non-overlapped latency from the move prediction muxes and Swap-Predict encoder changes. The $\sim$1.5 GHz clock of a compute-class GPU in 16nm technology is much longer than a cell-delay, and we do not expect the SwapCodes hardware to have any negative impact on the operating frequency of a GPU SM. All of our circuits, including the modified encoders, fit easily within the aggressive 250ps clock period we use with a 50% timing margin.

V. Comparison to Inter-Thread Duplication

While inter-thread duplication is not a transparent and general error detection mechanism, it has been used for GPUs in the past [24], [26], so we evaluate it as an alternative baseline for those benchmarks where it will run without programmer intervention.[7]

Methodology. To implement inter-thread duplication, we double the number of executing threads using a CUDA runtime library wrapper that modifies each kernel configuration and then use a modified backend compiler pass. The modified compiler adjusts thread-indexing special-register reads to make it appear as if the original and shadow threads have the same index (and therefore execute the same code). It also inserts shuffle-based

[7]We find that inter-thread duplication works for all Rodinia 2.3 programs used in this paper. It fails on matrix multiply due to the number of threads per CTA, and it fails on SNAP due to the program's use of shuffle instructions.

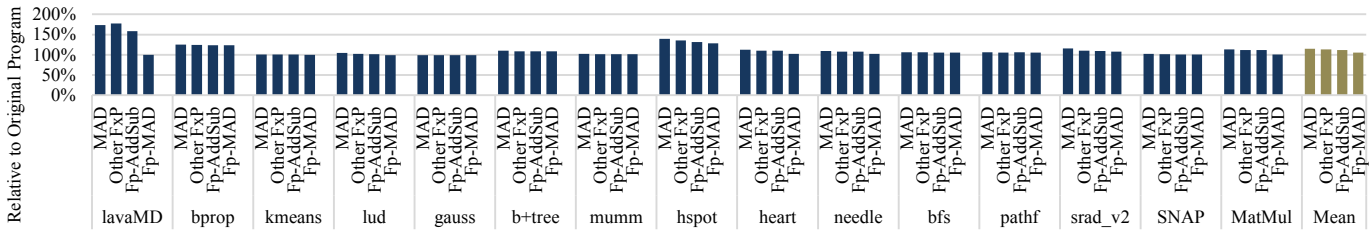


Fig. 16: Swap-Predict performance on an NVIDIA Tesla P100 GPU with plausible future check-bit predictors. The first bar ("MAD") is the most aggressive organization that is fully evaluated in Section IV-C. Other fixed-point predictors ("Other FxP"), and floating-point predictors ("Fp-AddSub", "Fp-MAD") follow.

checking instructions for the addresses and values at atomic memory operations and global memory stores. This duplication strategy is similar to the *Intra-Group-LDS FAST* configuration from [24] and the *Intra-Permute* configuration from [26], which is the most aggressive organization that they consider.

Performance. Figure 15 shows the performance of software-managed inter-thread duplication as well as a theoretical variant that eliminates checking instructions. In general, both the maximum (241% vs 99%) and average (113% vs 49%) slowdowns of inter-thread duplication are worse than our intra-thread duplication baseline. Eliminating all checking instructions still suffers worse maximum (114% vs 99%) and average (57% vs 49%) slowdowns than intra-thread duplication, ruling out the possibility that the performance of our inter-thread prototype is limited by its checking implementation. This shows that the intra-thread duplication baseline used by this paper is competitive with prior work.

VI. Discussion and Future Work

Other Residue Predictors. Section IV-C shows Swap-Predict to provide compelling average performance improvements by avoiding the need to duplicate the most common fixed-point operations. The worst-case Swap-Predict slowdown remains at 74% for lavaMD, however, since this benchmark is floating-point compute bound. Further innovation is possible if check-bit prediction were implemented for floating-point arithmetic. There is some prior work in this area using residue codes [52], [53], [54], but substantial design effort remains for SwapCodes adoption. Figure 16 projects Swap-Predict performance with residue arithmetic units for fixed-point logic and shift operations (which are predictable [55]) and floating-point operations. The average and worst-case overheads improve substantially with floating-point prediction ("Fp-MAD") to 5% and 28%, respectively, motivating future work into check-bit predictors for a wider variety of instructions.

GPUs with Register Bypassing. This work assumes that the GPU pipeline does not employ register bypassing. In a theoretical GPU with register bypassing, either bypassing should occur in a Swap-ECC-aware ECC-protected buffer or constraints should be enforced to ensure full Swap-ECC error coverage. One possible place for GPU register bypassing is in the operand collector, which is a staging area near the register file [37], [56]. If the operand collector stores ECC-protected data and it is augmented with Swap-ECC-aware write coalescing, then following reads could bypass the register file without issue. Alternatively, if bypassing passes non-ECC-protected data, it should be disabled for inputs to shadow instructions (and non-duplicated instructions, such as loads or stores). This limits the propagation of pipeline errors, maintaining SwapCodes error coverage. The inputs to original instructions may be bypassed without issue, because the same inputs will be checked by the non-bypassed SwapCodes shadow.

Error Recovery. Swap-ECC provides pipeline error detection, and existing recovery schemes (e.g., checkpoint-restart) work without change. Swap-ECC detects errors during register reads, containing pipeline errors and not allowing them to leak to memory. Such strict error containment may simplify the design of an efficient higher-level recovery scheme [57].

Alternative Optimizations. SInRG [4] has several hardware and software optimizations for intra-thread instruction duplication on GPUs. We leave direct comparison for future work, but expect Swap-ECC to perform roughly as well as *HW-Sig-SRIV* (the most aggressive SInRG organization for many benchmarks). HW-Sig-SRIV optimizes instruction duplication, but sacrifices error containment (allowing errors to leak to memory before detection) and it uses new hardware checkers and buffers. In contrast, Swap-ECC provides complete error containment without new checkers or buffers. Swap-Predict maintains these qualitative advantages with superior performance.

Swap-Predict with SEC-DED ECC. Our Swap-Predict evaluation uses residue codes for their strong arithmetic error coverage and efficient hardware implementations. It is also possible to design SEC-DED prediction units [34], [35], [36], but operations other than addition/subtraction tend to be expensive to predict. Our analyses indicate that Swap-Predict with SEC-DED and addition/subtraction prediction would be viable, offering 16% average slowdown and sacrificing little error coverage relative to a residue code.

VII. Conclusion

SwapCodes combines software-enforced intra-thread instruction duplication with a novel error coding scheme that leverages the register file ECC hardware to safely detect pipeline errors without sacrificing storage error correction. We describe SwapCodes variants that differ in complexity and overheads. Swap-ECC avoids the need for shadow storage or checking instructions with intra-thread duplication, reducing average performance overheads to 21%. Swap-Predict adds selective ECC prediction units to Swap-ECC to avoid the need to duplicate common operations, resulting in 15% average slowdown for the most performant organization we evaluate.

REFERENCES

[1] N. Oh, P. Shirvani, and E. McCluskey, "Error Detection by Duplicated Instructions in Super-scalar Processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

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

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

[4] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing Software-Directed Instruction Replication for GPU Error Detection," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2018.

[5] Advanced Micro Devices (AMD), Inc., "BIOS and Kernel Developer's Guide (BKDG) for AoMD Family 15h Models 60h–6Fh Processors," Tech. Rep. 50742 Rev 3.01, 2015.

[6] Sun Microsystems, Inc., "T2 Core Microarchitecture Specification." http://www.oracle.com/technetwork/systems/opensparc/t2-06-opensparct2-core-microarch-1537749.html.

[7] "NVIDIA Tesla P100—The Most Advanced Data Center Accelerator Ever Built." http://www.nvidia.com/object/pascal-architecture-whitepaper.html, 2016.

[8] J. Caffrey, "The Resiliency Challenge Presented by Soft Failure Incidents," *IBM Systems Journal*, vol. 47, pp. 641–652, 2008.

[9] 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, no. 2, pp. 129–173, 2014.

[10] 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 *Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249–258, 2017.

[11] B. Fang, *Error resilience evaluation on GPGPU applications*. PhD Thesis, University of British Columbia, 2014.

[12] D. Oliveira, L. Pilla, T. Santini, and P. Rech, "Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units," *IEEE Transactions on Computers*, vol. 65, no. 3, pp. 791–804, 2015.

[13] H. Cho, E. Cheng, T. Shepherd, C. Y. Cher, and S. Mitra, "System-level Effects of Soft Errors in Uncore Components," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 36, no. 9, pp. 1–14, 2017.

[14] 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.

[15] E. Fujiwara, *Code Design for Dependable Systems: Theory and Practical Applications*. John Wiley & Sons, 2006.

[16] M. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[17] A. Avizienis, "Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design," *IEEE Transactions on Computers*, vol. C-20, pp. 1322–1331, 1971.

[18] A. Avizienis, G. C. Gilley, F. P. Mathur, D. A. Rennels, J. A. Rohr, and D. K. Rubin, "The STAR (Self Testing And Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," *IEEE Transactions on Computers*, vol. C-20, pp. 1312–1321, 1971.

[19] D. Kinniment, I. L. Sayers, and E. G. Chester, "Design of a Reliable and Self-testing VLSI Datapath Using Residue Coding Techniques," *Computers and Digital Techniques, IEE Proceedings E*, vol. 133, no. 3, pp. 169–179, 1986.

[20] H. Naeimi, "An End-to-End ECC-based Resiliency Approach for Microprocessors," in *Workshop on Silicon Errors in Logic–System Effects (SELSE)*, 2011.

[21] J. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications*. New York, NY: Elsevier, 1978.

[22] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, 2009.

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

[24] 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.

[25] D. Sabena, M. Sonza Reorda, L. Sterpone, P. Rech, and L. Carro, "On the Evaluation of Soft-errors Detection Techniques for GPGPUs," in *Proceedings of the International Design and Test Symposium (IDT)*, pp. 1–6, December 2013.

[26] 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, 2017.

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

[28] NVIDIA Corporation, "NVIDIA Tesla V100 GPU Architecture," Tech. Rep. WP-08608-001_v1.150742 Rev 3.01, 2017.

[29] M. Hsiao, W. Carter, J. Thomas, and W. Stringfellow, "Reliability, Availability, and Serviceability of IBM Computer Systems: A Quarter Century of Progress," *IBM Journal of Research and Development*, vol. 25, pp. 453–468, 1981.

[30] D. K. Pradhan, ed., *Fault-Tolerant Computing: Theory and Technique*, vol. I. Old Tappan, NJ: Prentice Hall Inc., 1986.

[31] W. Clarke, L. Alves, T. Dell, H. Elfering, J. Kubala, C. Lin, M. Mueller, and K. Werner, "IBM System z10 design for RAS," *IBM Journal of Research and Development*, vol. 53, pp. 120–130, 2009.

[32] A. Meixner, M. E. Bauer, and D. J. Sorin, "Argus: Low-cost, Comprehensive Error Detection in Simple Cores," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 210–222, December 2007.

[33] R. Nathan and D. Sorin, "Argus-G: Comprehensive, Low-Cost Error Detection for GPGPU Cores," *Computer Architecture Letters*, vol. 14, no. 1, 2015.

[34] I. D. Elliott and I. L. Sayers, "Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction," *IEE Proceedings E - Computers and Digital Techniques*, vol. 137, no. 1, pp. 88–102, 1990.

[35] M. H. Sulaiman, S. I. M. Salim, A. Jaafar, and M. M. Ibrahim, "A Survey of Fault-tolerant Processor Based on Error Correction Code," in *IEEE Student Conference on Research and Development (SCOReD)*, pp. 1–6, 2014.

[36] W. F. Heida, *Towards a fault tolerant RISC-V softcore*. PhD thesis, Delft University of Technology, 2016.

[37] T. M. Aamodt, W. W. Fung, and T. H. Hetherington, *GPGPU-Sim 3.x Manual*. Revision 1.2.

[38] M. Gebhart, S. W. Keckler, B. Khailany, R. Krashinsky, and W. J. Dally, "Unifying Primary Cache, Scratch, and Register File Memories in a Throughput Processor," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pp. 96–106, 2012.

[39] M. Nicolaidis, "Carry Checking/Parity Prediction Adders and ALUs," *IEEE Transactions on VLSI Systems (TVLSI)*, vol. 11, no. 1, pp. 121–128, 2003.

[40] NVIDIA Corporation, "Instruction Set Reference: Maxwell and Pascal Instruction Set." http://docs.nvidia.com/cuda/cuda-binary-utilities/index.html#maxwell-pascal, 2016.

[41] B. Parhami, *Computer Arithmetic: Algorithms and Hardware Designs*. New York, NY: Oxford University Press, second ed., 2010.

[42] R. Zimmermann, "Efficient VLSI Implementation of Modulo $(2^n \pm 1)$ Addition and Multiplication," in *Proceedings of the IEEE Symposium on Computer Arithmetic*, pp. 158–167, 1999.

[43] S. Piestrak, "Design of Residue Generators and Multioperand Modular Adders Using Carry-Save Adders," *IEEE Transactions on Computers*, vol. 43, no. 1, pp. 68–77, 1994.

[44] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A Benchmark Suite for Heterogeneous Computing," in *Proceedings of the International Symposium on Workload Characterization*, pp. 44–54, 2009.
[45] Los Alamos National Laboratory, "SNAP: SN (Discrete Ordinates) Proxy Application." https://github.com/lanl/SNAP, 2013.
[46] NVIDIA Corporation, "CUDA 8.0 SDK Code Samples," 2017.
[47] M. Stephenson, S. K. Sastry 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 42Nd Annual International Symposium on Computer Architecture*, pp. 185–197, ACM, 2015.
[48] Synopsys Inc., "Design Compiler J-2014.09," August 2014.
[49] 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)*, 2018.
[50] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design," in *Proceedings of the Design Automation Conference (DAC)*, p. 101, 2013.
[51] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "MeRLiN: Exploiting Dynamic Instruction Behavior for Fast and Accurate Microarchitecture Level Reliability Assessment," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pp. 241–254, 2017.
[52] J.-C. Lo, "Reliable Floating-Point Arithmetic Algorithms for Error-Coded Operands," *IEEE Transactions on Computers*, vol. 43, no. 4, pp. 400–412, 1994.
[53] S. Elsayed, H. Fahmy, and M. Khairy, "Residue Codes for Error Correction in a Combined Decimal/Binary Redundant Floating Point Adder," in *Proceedings of the Asilomar Conference on Signals and Systems*, pp. 1444–1447, 2012.
[54] D. Lipetz and E. Schwarz, "Self Checking in Current Floating-Point Units," in *Proceedings of the IEEE Symposium on Computer Arithmetic*, pp. 73–76, 2011.
[55] T. R. N. Rao, "Error-Checking Logic for Arithmetic-Type Operations of a Processor," *IEEE Transactions on Computers*, vol. C-17, no. 9, pp. 845–849, 1968.
[56] S. Liu, J. E. Lindholm, M. Y. Siu, B. W. Coon, and S. F. Oberman, "Operand Collector Architecture," Nov 2010. Patent #US7834881B2.
[57] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D. W. Kim, D. H. Yoon, L. Kaplan, and M. Erez, "Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 1–11, 2012.