Compiler Techniques to Reduce the Synchronization Overhead of GPU Redundant Multithreading

Manish Gupta[1] Daniel Lowell[2] John Kalamatianos[2] Steven Raasch[2]
Vilas Sridharan[3] Dean Tullsen[1] Rajesh Gupta[1]
[1]CSE Dept.
University of California San Diego
{manishg, tullsen, rgupta}@cs.ucsd.edu
[2]AMD Research [3]RAS Architecture
Advanced Micro Devices, Inc.
{firstname.lastname}@amd.com

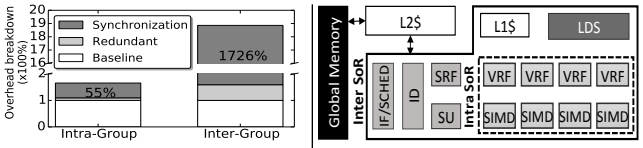
ABSTRACT

Redundant Multi-Threading (RMT) provides a potentially low cost mechanism to increase GPU reliability by replicating computation at the thread level. Prior work has shown that RMT's high performance overhead stems not only from executing redundant threads, but also from the synchronization overhead between the original and redundant threads. The overhead of inter-thread synchronization can be especially significant if the synchronization is implemented using global memory. This work presents novel compiler techniques using fingerprinting and cross-lane operations to reduce synchronization overhead for RMT on GPUs. Fingerprinting combines multiple synchronization events into one event by hashing, and cross-lane operations enable thread-level synchronization via register-level communication. This work shows that fingerprinting yields a 73.5% reduction in GPU RMT overhead while cross-lane operations reduce the overhead by 43% when compared to the state-of-the-art GPU RMT solutions on real hardware.

1. INTRODUCTION

Transient faults are a rising concern for mission-critical and large-scale systems. Transient faults may result in soft errors or Single-Event Upsets (SEUs) [8, 17]. Decreasing voltage levels and increasing numbers of transistors make future systems more susceptible to soft errors [11, 25]. Researchers have developed hardware and software techniques to detect and protect against soft errors [10, 22]. However, both hardware- and software-based protection schemes come at a cost. Hardware-based techniques increase area budget and time to market. The area overhead due to additional parity bits can be as high as 21% [7]. Moreover, hardware-based techniques cannot easily adapt to different workloads or market requirements. Software-based error detection techniques are flexible but come at a performance and energy cost. This work presents novel compiler techniques to reduce the cost of software-based error detection and make it viable for a commercial grade processor running a variety of workloads.

Redundant multithreading (RMT) is a transient error detection technique that uses redundant copies of a thread and compares results [21, 26, 29]. In the event of a mismatch, an error is flagged, and available error recovery schemes can be employed. To enable the comparison, both the redundant and the original threads must synchronize. For software-based RMT, synchronization is implemented using shared memory; therefore, a lock is first obtained on the shared memory followed by inter-thread communication.



(a) Synchronization overhead **(b)** GPU architecture

Figure 1: (a) Synchronization overhead is the major overhead of Intra- and Inter-Group GPU redundant multithreading. **(b)** Compute unit (CU). instruction fetch (IF), scheduler (SCHED), instruction decode (ID), scalar register file (SRF), scalar unit (SU), vector register file (VRF), Single instruction, multiple data (SIMD), local data share (LDS), L1 cache (L1$), shared L2 cache (L2$), Sphere of Replication (SoR).

We focus on GPUs because they provide an excellent platform for increase in parallel processing, and owing to their popularity in high-performance embedded computing [18]. Technology scaling and improving form factors have made GPUs accessible to more domains including the ones that demand absolute correctness, such as avionics and autonomous vehicles [19]. Specifically, we target GPU integrated into Accelerated Processing Unit (APU). An APU combines a low-power CPU with a high-performance GPU on a single SoC. APUs strike a good balance between form factor, power efficiency, and graphic capability required for portable embedded devices [6].

Two options for realizing RMT on a GPU are to execute the original-redundant thread pair (a) in lockstep within a single group (Intra-Group) or (b) independently in separate groups (Inter-Group) [29]. While the latter provides improved error coverage, the former offers lower performance overhead. Intra-Group RMT implements synchronization using low-latency local memory that is visible only to threads within a group. Inter-Group RMT implements synchronization using relatively slow global memory that is visible across the entire GPU. Similar to most software-based error detection schemes, these GPU RMT techniques also suffer from performance overheads, making them viable for only the most performance-tolerant safety-critical applications.

To understand the performance bottlenecks, we decompose the performance overhead of the state-of-the-art RMT schemes into redundant computation and synchronization overheads as shown in Figure 1a. For our benchmark suite

DAC '17, June 18-22, 2017, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/3061639.3062212

(discussed later), the total performance overhead for Intra- and Inter-Group RMT are 65.4% and 1785.9%, over the baseline execution time, respectively. The baseline execution time is the total kernel runtime of the original unmodified application. The synchronization overhead component alone is 55% for Intra- and 1726% for Inter-Group, dominating the performance penalty, and comprising 84.1% and 96.6% of the total overhead, respectively. Synchronization has shown to be a bottleneck not only for Intra- and Inter-Group GPU RMT but also for many CPU-based error detection mechanisms [26, 31]. Software-based redundancy techniques at instruction, process, and thread level are proven to be effective in detecting soft errors [9, 30, 31]. Error recovery overhead is orthogonal to detection; if recovery is implemented through incremental checkpoints, the overhead is separate from detection overhead. Moreover, error detection runs continuously while error recovery is employed only when an error is detected. In this work, we focus on reducing error detection overhead.

We propose and demonstrate reduction in the synchronization overhead for RMT by implementing novel compiler techniques, broadening the scope of RMT to performance-critical applications. We evaluate our work by running applications on real hardware.

1.1 Contributions

I. Quantitative analysis to break down RMT performance overhead into redundant computations vs. thread synchronization. We further decompose synchronization overhead for Inter-Group RMT into locking and communication. Our results show a reduction in RMT's locking overhead by 80.4% and communication overhead by 73.6%.

II. A new Intra-Group RMT optimization, Intra-Permute. Intra-Permute uses recently introduced cross-lane operations which have been independently shown to be an efficient way to share data between threads running in lockstep (lane) [23, 24]. We show that Intra-Permute reduces the overhead of Intra-Group RMT from 65% to 37%, a 43% reduction.

III. A new Inter-Group RMT optimization based on fingerprinting, Inter-Fingerprinting (Inter-FP). Inter-FP combines multiple synchronization events into a single event by hashing. We implement and contrast a light-weight XOR- and a robust CRC32-based hashing. We achieve a 73.5% reduction in Inter-Group overhead.

IV. We provide our techniques as a compiler transformation pass which can be used to enable/disable software-based error detection at compile time and applicable across different GPU architectures that support cross-lane operations.

2. BACKGROUND

The kernel launch is divided into a number of GPU threads also know as ***work-items***. A work-item is a single instance of the kernel. A set of 64 work-items which execute in lockstep on a single SIMD unit is termed a ***wavefront***. Further, a collection of wavefronts forms a ***workgroup***. A workgroup can have one or more wavefronts. Work-items within a workgroup can communicate through local scratch-pad memory.

The core of the AMD GCN architecture [7] is a Compute Unit (CU), shown in Figure 1b. Each CU consists of four 16-wide SIMD units. A SIMD unit can execute a wavefront, consisting of 64 work-items, in lockstep over four cycles.

2.1 GPU RMT

RMT is a reliability technique to detect faults in hardware through replicated computation rather than hardware duplication. Hardware protected through RMT is said to be within the Sphere of Replication (SoR). When data is brought into the SoR; e.g., from global memory, inputs are replicated and are used by the original and redundant work-items. The original and replicated computations run with no dependence on each other until the execution leaves the SoR. Any state change outside the SoR requires synchronization to enable output comparison. The outputs generated by the replicated copy are communicated to the original copy and compared for errors. Matching outputs indicate error-free execution, while a mismatch is a detected error. RMT is attractive for hardware devices where thread count is plentiful, and there is inherent latency tolerance, such as GPUs and the vector engine of APUs. Wadden et al. [29] published the first implementation of RMT that works on discrete GPUs. Their work presents Intra- and Inter-Group GPU RMT.

2.2 Intra-Group RMT (Intra-LDS)

Intra-Group RMT replicates work-items within a wavefront. The original and redundant work-items execute on the same SIMD unit in lockstep. The vector register file and SIMD units lie within the SoR, as shown by the dashed line in Figure 1b. The original work-item compares the communicated data with its local data to check for errors every time a store leaves SoR. The state-of-the-art Intra-Group RMT algorithm uses local memory (LDS) as the communication medium and is referred to as "*Intra-LDS*."

Figure 2b shows the code produced from Intra-LDS compiler transformation from the original code in Figure 2a. Every work-item is duplicated into work-item A (WIA) and work-item B (WIB) executing instructions ❶ (Compute Val) and ❷ (Compute Addr) in lockstep. To enable the error check for the store operation, the synchronization code is inserted where the communication is realized through a buffer in local memory, (*LDSBuf*). WIA and WIB diverge at branch B1. WIB's B1 branch shares its local value and address (data) by writing to *LDSBuf* at location V_p and A_p, respectively. WIA's B1 branch reads WIB's data and compares it with its local data, as shown by the function *errorChk*.

Limitations of Intra-LDS RMT. First, Intra-LDS RMT needs spare LDS memory for *LDSBuff*. LDS memory is limited and it could be a bottleneck for workloads which use LDS memory for performance optimizations, e.g. tiled matrix multiple. Second, the communication of data from WIB to WIA using *LDSBuf* is enclosed in between two special instructions, *memfence(acquire)* and *memfence(release)*. The *memfence* instruction is used to specify correct memory visibility and ordering [3]. Memory instructions within a work-item can be re-ordered as long as memory is in a consistent state for the work-item. Moreover, a value may live in L1 cache before it is committed to *LDSBuf*. However, if a memory location has to be consistent across different work-items within a workgroup *memfence()* instruction should be used. The semantics of *memfence* restricts re-ordering, and ensures correct memory visibility for different work-items within acquire-release boundaries. Third, WIA & WIB diverge twice per protected store, at branch B1 and B2. The branch B1 ensures that WIB writes to *LDSBuf* and WIA reads from *LDSBuf*. The branch B2 ensures that only WIA executes the error check and commits the protected store to memory. Divergence is a potential cause of loss in performance in GPUs, and should be avoided.

(a) Original code	(b) Intra-LDS compiler transformation		(c) Intra-Permute compiler transformation	
Work-item	**Work-item A (WIA)**	**Work-item B (WIB)**	**Work-item A (WIA)**	**Work-item B (WIB)**
R1=R2+R3; //Compute Val	1 R1=R2+R3;	1 R1=R2+R3;	1 R1=R2+R3;	1 R1=R2+R3;
R4=R4+4; //Compute Addr	2 R4=R4+4;	2 R4=R4+4;	2 R4=R4+4;	2 R4=R4+4;
STORE R1, [R4]; //SoR Exit	B1 memfence(acq);	B1 STORE R1, LDSBuf[Vp];	B1 R5=permute(R1, WIB);	B1 ....
REST	LOAD R5, LDSBuf[Vp];	STORE R4, LDSBuf[Ap];	R6=permute(R4, WIB);	....
	LOAD R6, LDSBuf[Ap];	memfence(rel);	errorChk(R1,R4,R5,R6);	....
	B2 errorChk(R1,R4,R5,R6);	B2	STORE R1, [R4];	....
	STORE R1, [R4];	....	**REST**	**REST**
	REST	**REST**		

Figure 2: Intra-Group RMT compiler transformation example: (a) Original code, (b) Intra-LDS, and (c) Intra-Permute.

2.3 Inter-Group RMT (Inter)

Inter-Group RMT replicates the entire workgroup. The original and redundant work-items within different workgroups don't ensure lockstep execution. Inter-SoR is a superset of Intra-SoR providing greater error coverage compared to Intra-Group RMT, as shown by the solid line in Figure 1b. Inter-SoR includes instruction fetch, decode, scheduler, scalar unit & register files, and local memory on top of Intra-SoR. Work-items within different workgroups cannot synchronize using LDS. Inter-Group RMT implements the synchronization operation through a global memory buffer every time execution leaves Inter-SoR, i.e. a global memory store. The state-of-the-art Inter-Group RMT is referred in this work as "*Inter*."

Limitations of Inter RMT. First, Inter RMT requires slower global memory to enable synchronization. Second, the absence of lockstep execution between the original and redundant work-item results in a leading and trailing work-item. Synchronization in between leading and trailing work-items involves additional locking overhead to ensure correct memory ordering on the shared global memory. The steps involved in synchronization are shown in flowchart Figure 3 in gray boxes (Figure 3 shows the improved Inter-FP algorithm, but can be used to visualize Inter synchronization). The synchronization overhead is composed of two major components: a) locking (L) and b) communication (C). The implementation of the locking component needs an additional global memory allocation to store the state of the lock for each work-item pair (lock buffer is not shown in the flowchart). The acquire lock block in the leading work-item reads the lock value from the lock buffer to check if it can write the data (value & address for Inter) to *GMBuf*. Similarly, the wait lock in the trailing work-item reads the lock value to confirm if the data is written by the leading work-item before the trailing work-item loads the data (value & address) and executes the error check. The implementation of locking further increases the global memory traffic resulting in increased synchronization overhead.

3. OUR APPROACH

In this section, we describe our optimizations for Intra- and Inter-Group GPU RMT. Our approach aims at removing limitations of existing GPU RMT algorithms.

3.1 Intra-Permute RMT

Figure 2c shows the Intra-Permute compiler transformation. Intra-Permute requires no LDS allocations. Instead, the technique uses newly introduced cross-lane operations to enable register-level communication [23, 24]. Work-item A (WIA) and work-item B (WIB) execute instruction 1, and 2 in lockstep. WIA and WIB diverge at the branch instruction B1. WIA executes a cross-lane operation using *permute* instructions to communicate the value and address operand of the store. The *permute* instruction implements both push and pull semantics, Figure 2c uses the pull semantics. WIA pulls registers $R1$ & $R4$ from the vector register files of its neighbouring work-item (WIB) into its own registers $R5$ & $R6$, respectively. WIA checks its local value and address ($R1$ & $R4$) with WIB's permuted value and address ($R5$ & $R6$), as shown by the function *errorChk*. WIA also commits the protected store operation to memory after the error check. WIB skips the execution of the *permute* instruction, error check, and the execution of the protected store operation. For the rest of the code two work-items execute in lockstep until the next store operation. Intra-Permute only diverges once per protected store i.e. branch B1. Intra-Permute results in reduced branch divergence, exclusion of *memfence*, fewer total instructions, and the use of efficient register-level communication.

3.2 Inter-Fingerprinting RMT

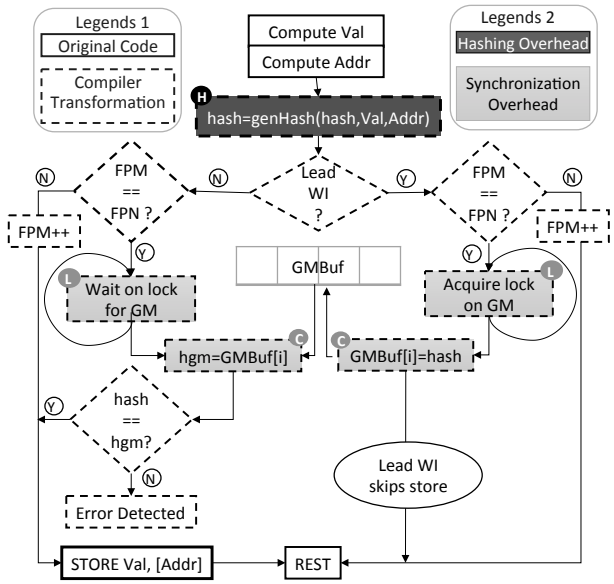


Figure 3: Inter-FP RMT. **FPN**: Fingerprinting threshold value. **FPM**: Number of stores hashed into the current hash.

Fingerprinting (FP) is a technique to amortize the cost of Inter-Group RMT synchronization (locking and communication) over several store operations. Inter-FP RMT reduces the number of synchronization points for the Inter-Group RMT algorithm.

The flowchart in Figure 3 shows steps involved in the Inter-FP algorithm. The solid box shows the original basic block present in the kernel code, and the dashed box shows basic blocks inserted by our Inter-FP compiler transformation. The flow chart shows that both the leading and trailing work-items independently compute a value (Compute Val), an address (Compute Addr), and a generated hash. The hash generation is independent of the core Inter-

Benchmark	Size	NK	NKC	GTC	DLS	DGS
matmul (MM)	4K	1	1	16.8M	512	1
simpleMatmul (SM)	4K	1	1	16.8M	1	1
FFT	20M	1	1	1.3M	32	16
SPMV	16K	1	1	263K	1-5	1
XSBench (XSB)	large	1	1	15M	0	7
CoMD	64^3	3	3	6.2M	0	338
miniAMR (mAR)	8^3	1	320	3.1M	10-30	8
snap-c (SnPC)	medium	1	101840	79.3K	0-4	4

Table 1: Benchmarks. **Size**: Problem input sizes **NK**: Num of Kernels, **NKC**: Number of Kernel Calls, **GTC**: Global Thread Count (baseline), **DLS**: Dynamic LDS Stores, **DGS**: Dynamic Global Memory Stores.

FP algorithm and is shown by the function *genHash(Val, Addr, hash)* (H). The *genHash* function takes as input a value (*Val*), an address (*Addr*), and the previous hash to generate the new hash. At this point, leading and trailing work-items check their local hash counts, *FPM*, with the fingerprinting threshold, *FPN*. If *FPM* is equal to *FPN*, the leading-trailing work-item pair synchronize to enable correct communication though *GMBuf* and error check. The synchronization consists of two major components – spin-locks to gain access to *GMBuf* and communication of data (hash value for Inter-FP) to *GMBuf*.

The leading work-item acquires the lock (L) on *GMBuf* before it communicates (C) the local *hash* value to *GMBuf*, and sets the lock (L) to ready state. The trailing work-item waits on the lock (L) to become ready before it reads (C) the leading work-item's *hash* value from *GMBuf*. The leading work-item skips the store and continues executing the rest of the program. Meanwhile, the trailing work-item compares it's local *hash* with the communicated *hash* value. A mismatch in *hash* is reported as an error. The trailing work-item then commits the protected store operation to memory. On exit of a kernel, if there is an outstanding hash which has not been compared, the leading and trailing work-items must synchronize one last time.

Inter vs. Inter-FP (Synchronization Overhead). Inter-FP cuts synchronization overhead in two ways. First, Inter-FP removes the number of times work-items enter synchronization, reducing the overhead due to locking. Second, performance improvement comes from the hashing and reducing the number of bits communicated via *GMBuf*. While Inter communicates both value and address for every protected store, Inter-FP hashes value and address operands and communicates the hash. This results in fewer memory instructions even for a fingerprinting threshold of one. The benefits of hashing further increases with a larger fingerprinting threshold. Hence, Inter-FP reduces the synchronization overhead by reducing both the locking (L) and communication (C) components. However, there's an added computation overhead of generating the hash value (H).

Hashing. We implement XOR and CRC32 for the *getHash* function. XOR provides a light-weight means of catching some, but not all, single bit flips per hash. The Hamming distance conventionally for XOR is 2, catching single bit errors. However, there's a low probability of two single bit errors in the same bit position in two hashed data items getting masked. In order to cover multi-bit errors we also implement computationally intensive CRC32-based hashing. We use the IEEE 802.3 CRC32 polynomial function which has a Hamming Distance of six for 268-bit data length, which allows us to catch all single bit errors within five hashes of 32-bit length [15, 20]. CRC32 has an extremely low probability of collision i.e. 2^{-32} [15]. Any error in hashing also has the same level of protection as the rest of the kernel code because the hashing computations are also duplicated by the redundant work-item. We will refer to XOR and CRC32 Inter-FP variants as FPNXOR and FPNCRC32, where FPN is shorthand for fingerprinting with threshold N.

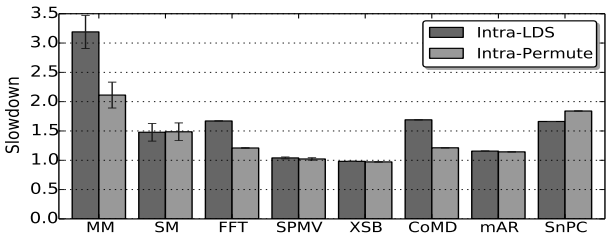


Figure 4: Performance overhead of Intra-LDS and Intra-Permute normalized to the runtime of original kernel on AMD GCN3. On average Intra-Permute performs 43% better relative to Intra-LDS.

4. EXPERIMENTAL SETUP & RESULTS

We run benchmarks transformed with our RMT techniques on AMD FX-8800P (GCN3) APU. The transformations are implemented in an open-source HCC compiler, an LLVM-based HSA compiler [2]. The software environment includes HSA runtime [5] and HSA kfd 1.6 Linux drivers [4]. We use a modified version of CodeXL [1] v2.2 to measure the kernel runtimes, power, and energy usage. We use four embedded-focused applications (MM, SM, FFT, and SPMV) and four ProxyApps [12] (XSBench, CoMD, miniAMR, and snap-c) shown in Table 1.

Performance: Intra-LDS vs. Intra-Permute Figure 4 shows the performance overhead for Intra-LDS and Intra-Permute RMT variants normalized to the runtime of the original kernel. Intra-Permute runs faster than Intra-LDS for all the benchmarks except snap-c. The opportunity to reduce the communication cost with Intra-Permute depends on the number of dynamic local (DLS) and global memory stores (DGS) present in a benchmark -- see Table 1. SM, SPMV, XSB, and mAR, the benchmarks with fewer dynamic stores, don't gain much from Intra-Permute. However, MM, FFT, and CoMD with more dynamic stores show a substantial reduction in slowdown with Intra-Permute by reducing the communication overhead associated with each dynamic store. snap-c shows a decrease in performance with Intra-Permute relative to Intra-LDS. Intra-Permute increases the register pressure and changes the instruction mix to have more VALU instructions. These may result in reduced parallelism and a slowdown if the original kernel is dominated by VALU instructions. On average Intra-LDS and Intra-Permute run 1.65x and 1.37x longer than baseline, respectively. Intra-Permute reduces the average RMT performance overhead from 65% to 37%, which is a 43% reduction in overhead.

Performance: Inter vs. Inter-FP. In Figure 5, the first bar for each benchmark shows the slowdown of Inter RMT. For example, while SPMV shows a 2x slowdown, FFT shows 126x slowdown of the baseline runtime. The major takeaway from the Inter performance results are *a)* the slowdown for Inter-Group RMT is 23 times worse than Intra-LDS RMT because of the use of global memory as a communication medium and the explicit locking mechanism to coordinate synchronization between work-items, and *b)* the performance overhead varies widely between different benchmarks.

In Figure 5, bars 2-5 and 6-9 for each benchmark show the slowdown for FP2-8XOR and FP2-8CRC32 RMT variants, respectively. From Inter-FP performance analysis we

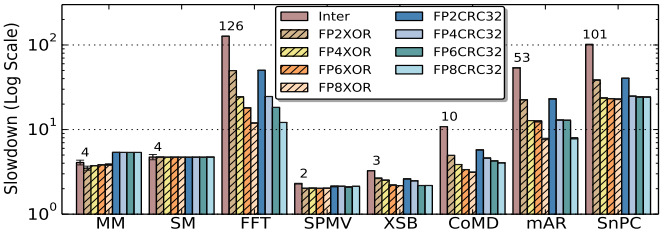
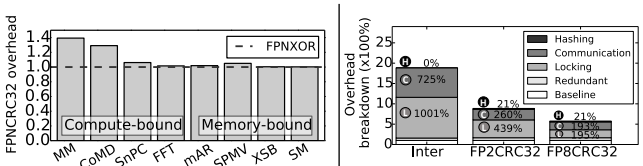


Figure 5: Performance overhead of Inter, FPNCRC32, and FPNXOR RMT normalized to the runtime of the original kernel. Performance of Inter-FP depends on the number of dynamic global stores (DGS) -- see Table 1.



(a) CRC vs XOR hashing **(b)** Overhead breakdown

Figure 6: **(a)** FPNCRC32 runtime normalized to FPNXOR. **(b)** Breakdown of Inter and Inter-FP RMT overheads. Inter-FP reduces both locking (L) and communication (C) component of synchronization overhead at a small cost of hashing (H).

can conclude: *a)* performance improvements by increasing fingerprinting threshold (FPN) depends on the number of dynamic global memory stores (DGS) available in a benchmark (see Table 1), and *b)* computationally intensive FPNCRC32 only shows marginal increase in slowdown relative to FPNXOR. The discussion below compares the performance of Inter-FP with Inter unless otherwise mentioned explicitly.

Inter-FP improves performance by reducing the number of synchronization events performed to do the error check for each DGS. For benchmarks MM, SM, and SPMV with only one DGS, Inter-FP doesn't have an opportunity to reduce the number of synchronization events. However, Inter-FP can still save communication bandwidth by communicating one hash instead of both value and address, as done with Inter. MM (only FPNXOR), SM, and SPMV show a performance improvement because of the reduced communication bandwidth through hashing. Benchmarks with one DGS show no performance benefit with increasing the fingerprinting threshold (FPN). However, benchmarks FFT, XSB, CoMD, MAR, and SnPC with more than one DGS do show an increase in performance with the increase in FPN. FFT, with 16 DGS, shows 60.7%, 80.7%, 85.8%, and 90.5% reduction in slowdown when 2, 4, 6, and 8 global memory stores are skipped before synchronization using FPNXOR. XSB and CoMD with Inter-FP variants show similar behavior to FFT. mAR shows the reduction in slowdown with Inter-FP RMT with FP2 and FP4. However, there is no reduction in slowdown from FP4 to FP6. This is because the number of DGS per work-item is eight, which means for both FP4 and FP6 there are two synchronization events per kernel call. The average performance overhead reduction with FPNCRC32 and FPNXOR are 73.5% and 74.25%, respectively.

Hashing XOR vs. CRC32. Figure 6a shows performance overhead of FPNCRC32 RMT variants normalized to the runtime of FPNXOR RMT kernel. We arranged all the compute-bound benchmarks to the left and all the memory-bound benchmarks to the right. We observe that compute-bound benchmarks have a higher overhead for CRC32 computations than memory-bound benchmarks. For memory-bound benchmarks, the overhead of CRC32 is hidden underneath memory latency. MM and SM at the opposite ends of the compute- and memory-bound spectrum show a clear contrast in CRC32 computation overhead. MM is an optimized implementation of matrix multiply which uses LDS memory to reduce the global memory traffic. The additional CRC32 computations worsen the already compute-bound MM kernel performance. MM with FPNCRC32 shows a performance degradation of 39.3% relative to FPNXOR RMT variant. SM shows only 0.2% performance degradation relative to FPNXOR. The average overhead of CRC32 computations is 10.4% for all the benchmarks, while the overhead of CRC32 for memory-bound benchmarks (mAR, SPMV, XSB, and SM) is 1.8%.

Inter-FP performance breakdown. The stacked bar (H) in Figure 6b shows the hashing overhead for Inter, FP2CRC32, and FP8CRC32, respectively. The hashing overhead for Inter RMT is zero. The hashing overhead for FP2CRC32 and FP8CRC32 is 21% over the baseline. This is because FP variants do the same number of hashing computations regardless of number of synchronization events. The stacked bar (C) shows the communication overhead for Inter, FP2CRC32, and FP8CRC32 which are 725.7%, 260.2%, and 193.0% over the baseline run, respectively. FP2CRC32 and FP8CRC32 reduce the communication overhead by 63.8% and 73.6% relative to Inter RMT, respectively. The stacked bar (L) shows the locking overhead for Inter, FP2CRC32, and FP8CRC32 which are 1001.2%, 439.4%, and 195.0% over the baseline run, respectively. FP2CRC32 and FP8CRC32 reduce the locking overhead by 56.1% and 80.4% relative to Inter RMT, respectively. The second stacked bar from the bottom shows the overhead of redundant computation for Inter, FP2CRC32 and FP8CRC32 RMT variants. Redundant computation overhead for all the three variants remains approximately at 60% over the baseline run.

Our results show that Inter-FP reduces the locking and communication overhead up to 80.4% and 73.6% of the state-of-the-art Inter RMT. The combined effect results in synchronization overhead reduction of 78% over Inter RMT.

Energy Usage (EU). EU is the energy used by the GPU. EU is defined as the area under the GPU power consumption over time curve. We estimate EU by periodic sampling of power consumption at a 50 ms interval using CodeXL [1]. The average EU for our benchmark suite with Intra-LDS and Intra-Permute RMT increase by 1.30x and 1.25x of the baseline EU, as shown by the first two bars in Figure 7. Intra-Permute results in energy overhead reduction of 16% relative to Intra-LDS. The average power for Intra-LDS and Intra-Permute RMT increase by 1.17x and 1.18x over the baseline average power. There is a 5% increase in average power for Intra-Permute relative to Intra-LDS. Intra-Permute enables faster intra-thread communication and thus more vector ALU instructions are executed per clock, resulting in slightly higher average power. Hence, the reduced execution time is the prime contributor to lower energy usage.

The next five bars in Figure 7 show the EU for the Inter and Inter-FP variants normalized to the baseline EU. The average EU for our benchmark suite with Inter, FPNXOR and FPNCRC32 are 10x, 4.2x and 4.4x of the baseline, respectively. FPNXOR and FPNCRC32 result in energy overhead reduction of 64.4% and 62.2% relative to Inter. The average power with Inter, FPNXOR, and FPNCRC32 are 2.0x, 1.76x, and 1.83x of the baseline power, respectively. The EU overhead of CRC32 relative to XOR is consistently higher across most benchmarks, unlike their relative per-

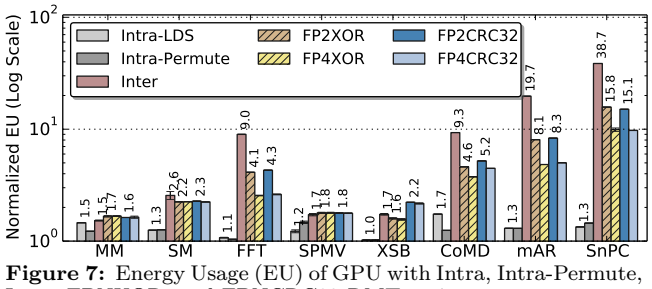


Figure 7: Energy Usage (EU) of GPU with Intra, Intra-Permute, Inter, FPNXOR, and FPNCRC32 RMT variants.

formance overhead. The performance overhead of CRC32's extra computations can hide behind the memory latency, but it shows up in average power and EU.

5. RELATED WORK

Redundancy-based error detection techniques are proposed at many levels, including instruction [9, 22] and process [26]. Instruction-level redundancy techniques duplicate instructions and synchronize at memory operations, branches, & function calls. Process-level redundancy techniques synchronize at a coarser granularity, such as system calls. Didehban et al. present compiler techniques to increase coverage for non-duplicated instructions and provide near zero silent data corruption [9]. The work on increased converge and efficacy of software-based redundancy techniques is complimentary to our work that provides the reduction in performance overhead.

With the rise of multi-core CPUs, many ways to provide error detection through replicated computations are proposed for both hardware [27, 28] and software [13, 16]. Smolens, et al. [27] present a hardware-based solution for error detection for CPUs. The work demonstrates the reduction in communication bandwidth for threads executing on two processors in lockstep. They evaluate their work through simulations. In contrast, we propose a software-based fingerprinting solution for GPUs and handle the challenges of non-lockstep execution. We demonstrate reduction in both communication bandwidth and locking overhead. The software-based implementation allows users to plug in any hashing scheme. Moreover, we show all results by running RMT-ized binaries on real hardware.

Wadden et al. published a software-based GPU RMT [29]. Kalra et al. ported the core RMT algorithms introduced by Wadden et al. to an HSA compiler and evaluated their work on an HSA-compliant GCN2 APU device [14]. Wadden et al. used a discrete GPU and OpenCL^{TM} programming environment to implement Intra- and Inter-Group GPU RMT. They also mention the possibility of using a swizzle instruction. They made custom changes to the shader compiler (compiler backend) to fit their architecture and showed that hardware optimizations can benefit Intra-Group RMT. Their approach observes a performance slowdown for some benchmarks due to casting and packing/unpacking of communicated values via 32-bit registers. In contrast, we provide an intermediate level compiler transformation using more efficient/newly introduced 32- or 64-bit cross-lane operations which not only provides portability across different GPU architectures but also results in exclusion of *memfence* instructions and reduced branch divergence.

6. CONCLUSION

We demonstrate techniques that reduce synchronization overhead for GPU RMT using cross-lane operations and fingerprinting. We implement two very diverse hashing schemes and show that hashing overhead is not always a bottleneck. We provide our techniques as compiler transformations which can be enabled at compile time and are portable across different GPU architectures. The use of such flexible software-based error-detection to protect against transient faults significantly reduces the required hardware complexity, enabling these techniques to work on existing systems. The reduced hardware complexity translates to area savings and/or performance boost.

Acknowledgements

We would like to sincerely thank Joe Greathouse, Mauricio Breternitz, and Tony Tye for their feedback. This research was supported by the U.S. Department of Energy (DoE) under the FF2 program. This work is also supported by the NSF Expedition in Computing grant CCF-1029783. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the DoE or NSF. AMD, the AMD Arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] AMD CodeXL gpuopen.com/compute-product/codexl.
[2] *HCC https://github.com/RadeonOpenCompute/hcc.*
[3] *HSA http://www.hsafoundation.com/standards.*
[4] *HSA Kernel Driver https://github.com/HSAFoundation/HSA-Drivers-Linux-AMD.*
[5] *HSA Runtime https://github.com/HSAFoundation/HSA-Runtime-AMD.*
[6] AMD. A beginner's guide to computational computing on the amd embedded g-series apu with opencl. White Paper.
[7] AMD. Graphics Core Next Architecture, Generation 3.
[8] F. Cappello et al. Toward Exascale Resilience. *Int. J. High Perform. Comput. Appl.*, 2009.
[9] M. Didehban et al. nZDC: A Compiler Technique for Near Zero Silent Data Corruption. DAC, 2016.
[10] D. Ernst et al. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. MICRO, 2003.
[11] J. Henkel et al. Reliable On-chip Systems in the Nano-era: Lessons Learnt and Future Trends. DAC, 2013.
[12] M. Heroux et al. ASC Co-design Proxy App Strategy. 2016.
[13] J. S. Hu et al. Compiler-Directed Instruction Duplication for Soft Error Detection. DATE, 2005.
[14] C. Kalra et al. Performance Evaluation of Compiler-based SW RMT in an HSA Environment. SELSE, 2015.
[15] Koopman et al. 32-Bit Cyclic Redundancy Codes for Internet Applications. DSN, 2002.
[16] F. Kriebel et al. ageOpt-RMT: Compiler-driven Variation-aware Aging Optimization for RMT. DAC, 2016.
[17] S. E. Michalak et al. Predicting the number of fatal soft errors in Los Alamos National Laboratory's ASC Q supercomputer. *IEEE Transactions*, 2005.
[18] D. Mor. GPGPU for Embedded Systems. White Paper.
[19] NVIDIA. https://blogs.nvidia.com/blog/2016/08/22/parker-for-self-driving-cars. 2016.
[20] W. W. Peterson et al. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 1961.
[21] S. K. Reinhardt et al. Transient Fault Detection via Simultaneous Multithreading. In *ISCA*, 2000.
[22] G. A. Reis et al. SWIFT: Software Implemented Fault Tolerance. CGO, 2005.
[23] B. Sander. AMD gcn assembly: Cross-lane operations. 2016.
[24] M. Schott. NVIDIA reading between the threads. 2016.
[25] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. DSN 2002.
[26] A. Shye et al. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Trans.*, 2009.
[27] J. C. Smolens et al. Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth. ASPLOS, 2004.
[28] T. N. Vijaykumar et al. Transient-Fault Recovery Using Simultaneous Multithreading. ISCA, 2002.
[29] J. Wadden et al. Real-world design and evaluation of compiler-managed gpu rmt. ISCA, 2014.
[30] C. Wang et al. Compiler-Managed Software-based RMT for Transient Fault Detection. CGO, 2007.
[31] Y. Zhang et al. Runtime Asynchronous Fault Tolerance via Speculation. CGO, 2012.