Cost-Effective Soft-Error Protection for SRAM-Based Structures in GPGPUs

Jingweijia Tan, Zhi Li, Xin Fu
Department of Electrical Engineering and Computer Science
University of Kansas
Lawrence, KS 66045, USA
{jtan, zhili, xinfu}@ittc.ku.edu

ABSTRACT
The general-purpose computing on graphics processing units (GPGPUs) are increasingly used to accelerate parallel applications. This makes reliability a growing concern in GPUs as they are originally designed for graphics processing with relaxed requirements for execution correctness. With CMOS processing technologies continuously scaling down to the nano-scale, on-chip soft error rate (SER) has been predicted to increase exponentially. GPGPUs with hundreds of cores integrated into a single chip are prone to manifest high SER. This paper aims to enhance the GPGPU reliability in light of soft errors. We leverage the GPGPU microarchitecture characteristics, and propose energy-efficient protection mechanisms for two typical SRAM-based structures (i.e. instruction buffer and registers) which suffer high susceptibility. We develop Similarity-AWare Protection (SAWP) scheme that leverages the instruction similarity to provide the near-full ECC protection to the instruction buffer with quite little area and power overhead. Based on the observation that shared memory usually exhibits low utilization, we propose SHAred memory to Register Protection (SHARP) scheme, it intelligently leverages shared memory to hold the ECCs of registers. Experimental results show that our techniques have the strong capability of substantially improving the structure vulnerability, and significantly reducing the power consumption compared to the full ECC protection mechanism.

Categories and Subject Descriptors
B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; I.3.1 [**Computer Graphics**]: Hardware Architecture – *Graphics processors*

General Terms
Design, Reliability

Keywords
GPGPU, Reliability, Soft Error, SRAM, Energy Efficiency.

1. Introduction
Modern graphics processing units (GPUs) are composed of hundreds of on-chip cores, they support thousands of parallel threads and provide remarkably higher computational throughput than CPU. For example, NVIDA's GeForce 8800 [1] provides up to 197× higher throughput than Intel's Core2Duo processors on data intensive applications. The new programming models (e.g. NVIDIA CUDA™ [2], AMD Brook+ [3], and OpenCL [4]) further reduce the programmers' efforts in writing general-purpose applications using GPUs. With the increasing computing power and improved programmability, general-purpose computing on GPUs (GPGPUs) emerges as a highly attractive platform for a wide range of parallel applications. In fact, a recent trend observed in TOP500 supercomputers is the increasing adoption of GPGPUs to deliver high computational throughput [15].

This extensive usage of GPGPU makes reliability a critical concern. Current GPUs have quite limited capability in error detection and fault tolerance. Historically, GPUs are mainly designed for graphics processing, errors in those applications are effectively masked and 100% computation correctness is not required [5]. However, general-purpose applications such as scientific computing, financial application and medical data processing, require strict execution correctness. For example, in the GPGPU application computing a correlation function [25], 1% of value errors in any of the program output elements is treated as a silent data corruption (SDC) error and cannot be tolerated. From the device side, CMOS integrated circuits are facing high environmental susceptibility with the shrinking of feature sizes. Soft errors, also called transient faults or single-event upsets (SEUs), are failures caused by high-energy neutron or alpha particle strikes in integrated circuits. These failures may silently corrupt the data and lead to erroneous computation results. Soft error rate (SER) has been predicted to increase exponentially [6, 7]. GPGPUs with hundreds of cores integrated into a single chip are prone to manifest high SER [8]. For examples, eight soft errors were observed in a 72-hour run of testing program on 60 NVIDIA GeForce 8800GTS 512 [9]. It is also found that the silent data corruption (SDC) ratio in commodity GPUs with weak/no error protection is 16~33% [10], significantly higher than that in CPUs (<2.3%) with strong protection. If left unattended, this reliability challenge will soon become obstacle to future GPGPUs by either preventing them from scaling down to smaller feature sizes or resulting in the imprecise operation of these systems.

Existing soft-error reliability optimization mechanisms limit on CPU processors [11-14, 16-17] and largely ignore the emerging GPGPUs. In CPUs, the software-based redundancy, such as opportunistically triggering a redundant thread [16] when the main thread stalls for long-latency memory accesses, has been widely studied. It efficiently minimizes the performance degradation caused by the redundant execution since the main and redundant threads dynamically share the pipeline resources. However, every parallel thread in GPGPU has statically allocated resources, including the register files and on-chip shared memory, making such opportunistic redundant multi-threading infeasible. It has been found that the software-based replication in GPGPU leads to

CF' 13, May 14-16, 2013, Ischia, Italy.

high performance overhead if high fault coverage is desired [8]. Furthermore, methodologies that can be simply extended to GPGPUs fail to leverage the GPGPU microarchitecture characteristics in achieving the cost-effective fault-tolerance. For example, the error correction codes (ECC) table explored in [11] for CPU register protection may cause large area overhead when applied in GPGPU with thousands of registers. Therefore, it is imperative to develop a different set of GPGPU-aware reliability optimization techniques in the presence of soft errors.

In this paper, we explore reliable GPGPU microarchitecture designs to efficiently combat soft errors in light of small-scale processing technology. We focus our study on SRAM-based structures, and it is a necessary first step towards protecting the entire GPGPU processor. By using a reliability-aware architecture simulator for GPGPUs, we find that the instruction buffer and register files are the major SRAM-based structures exhibiting high soft-error vulnerability. Both structures are sizeable that keep the architectural state; they are likely to be the reliability hot-spots. We take advantage of the GPGPU microarchitecture characteristics, and develop two cost-effective protection mechanisms for the instruction buffer and registers, respectively.

The contributions of this work are:

(1) We observe that threads usually keep similar progress when executing in the GPGPU streaming multiprocessor, and majority instructions in the instruction buffer share the identical PC. Thus, one instruction's ECC could be used for multiple instructions. We propose Similary-AWare Protection (SAWP) to leverage the instruction similarity and protect the instruction buffer by implementing a small-size ECC table, which provides the near-full protection to the buffer with little area and power overhead.

(2) We observe that the shared memory usually keeps low utilization in many GPGPU applications (detailed description is shown in Section 3.3.1). Considering its unique characteristics (e.g. read/write-able, low access latency), the shared memory serves as the ideal candidate for registers fault tolerance. We propose SHAred memory to Register Protection (SHARP) that takes advantage of the under-utilized shared memory to intelligently hold the ECCs of a set of high vulnerable registers (i.e. registers with long lifetime), and substantially enhance the registers reliability with quite small area and power overhead.

(3) Experimental results show that both SAWP and SHARP have the strong capability in fault tolerance with little power consumption. SAWP reduces the soft-error vulnerability of instruction buffer by 68%, and SHARP reduces register vulnerability by 41% compared to the case without any protection scheme. Moreover, SAWP (SHARP) is able to reduce the power consumption by 17% (18%) compared to the full ECC protection mechanism.

The rest of this paper is organized as follows: Section 2 provides background on GPGPUs and soft errors. Section 3 presents our two techniques to cost effectively enhance the soft-error robustness of SRAM-based structures in GPU streaming multiprocessors. Section 4 describes experimental methodologies. Section 5 evaluates the proposed techniques. We discuss the related work in Section 6, and conclude the paper in Section 7.

2. Background

2.1. General-purpose computing on graphics processing units (GPGPUs) architecture

A typical GPU consists of a scalable number of in-order streaming multiprocessors (SM) that can access to multiple on-chip memory controllers via an on-chip interconnection network [2]. Figure 1 illustrates the SM microarchitecture [35]. It contains the fetch and decode unit, instruction buffer (I-Buffer), branch unit, register file (RF), streaming processors (SP), special functional units (SFU), load-store units, shared memory, and so on.

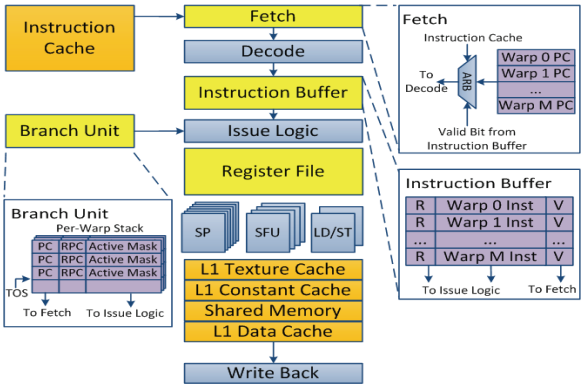


Figure 1. An overview of the SM microarchitecture

To facilitate GPGPU application development, several programming models have been developed by NVIDIA and AMD. In this paper, we study the NVIDIA CUDA programming model but the basic constructs will hold for most programming models. In CUDA, the GPU is treated as a co-processor that executes highly-parallel kernel functions launched by the CPU. The kernel is composed of a grid of light-weighted threads; a grid is divided into a set of blocks (referred as cooperative thread arrays (CTA) in CUDA); each block is composed of hundreds of threads. Threads are distributed to the SMs at the granularity of blocks, and threads within a single block communicate via the shared memory and synchronize at a barrier if desired. Per-block resources, such as registers, shared memory, and thread slots in an SM are not released until all the threads in the block finish execution.

Threads in the SM execute on the single-program multiple-data (SPMD) model. A number of individual threads (e.g. 32 threads) from the same block are grouped together, called warp. In the pipeline, threads within a warp execute the same instruction but with different data values. Each SM interleaves multiple warps (e.g. 32) on a cycle-by-cycle basis. The execution of a branch instruction in the warp may cause warp divergence when some threads jump while others fall through at the branch.

As Figure 1 shows, each warp has a dedicated slot in the fetch unit and I-Buffer. It also has own stack in the branch unit recording the reconvergence PC (RPC) and active mask (used to describe the active threads in the warp) to handle the warp divergence. At every cycle, the fetch unit selects the PC for a warp whose instruction slot is empty (i.e. the Valid bit is set as invalid in the instruction buffer), and fetches the instruction from the instruction cache. The instruction is decoded and written into the corresponding warp slot in the instruction buffer. It waits there and will not be ready for issue until its previous instruction completes. By checking the Ready bit in the instruction buffer, the issue logic chooses a ready warp instruction for the register access and execution. Once issued, the slot holding that issued instruction is marked as invalid in the I-Buffer. In the SM, all threads in a warp access the same-named registers (i.e. register vector) simultaneously, the register values are processed in parallel across the SP, SFU or load-store units. GPU is usually equipped with its own off-chip external memory (e.g. global memory) connected to the on-chip memory controllers. The off-chip memory access can last hundreds of cycles, and a long latency memory transaction from one thread would stall all threads within a warp. In other words, the warp cannot proceed until all the memory accesses from its threads complete. The load/store requests issued by different threads can get coalesced

into fewer memory requests according to the access pattern. Memory coalescing improves performance by reducing the requests for memory access.

2.2. Microarchitecture level soft-error vulnerability analysis

A key observation of soft error behavior at microarchitecture level is that a SEU may not affect processor states required for program's correct execution. At microarchitecture level, the overall hardware structure's soft error rate is decided by two factors [17]: the FIT rate (Failures in Time, which is the raw SER at circuit level) per bit, mainly determined by circuit design and processing technology, and the architecture vulnerability factor (AVF) [20]. A hardware structure's AVF refers to the probability that a transient fault in that hardware structure will result in incorrect program results. Therefore, the AVF, which can be used as a metric to estimate how vulnerable the hardware is to soft errors during program execution, is determined by the processor state bits required for architecturally correct execution (ACE). At the instruction level, an instruction is defined as ACE instruction if its computation result affects the program final output, and AVF is primarily determined by the quantity of ACE instructions per cycle and their residency time within the structure [20]. In this study, we use AVF as the major metric to estimate structure soft-error vulnerability.

3. Cost-effective soft-error protection for SRAM-based structures in streaming multiprocessors

In this section, we analyze the GPGPU microarchitecture vulnerability, and find that among various SRAM-based structures in the streaming multiprocessor, the instruction buffer and registers show great susceptibility to soft errors. We make two observations on GPGPU microarchitecture characteristics, and leverage them to propose a set of protection techniques for the two structures in Section 3.2 and 3.3., respectively.

3.1. Motivation: the reliability hot-spots in GPGPU microarchitecture

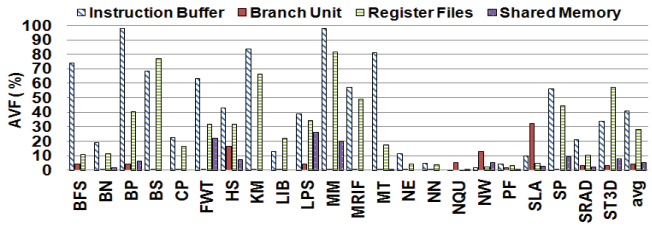


Figure 2. The AVF of the key GPGPU microarchitecture structures including instruction buffer, branch unit, register files, and shared memory

There have been various frameworks developed to estimate the CPU microarchitecture level soft error vulnerability [21, 22]. However, they are not applicable to the GPGPU microarchitecture that implements in-order SIMD pipeline and has significantly different architecture and data/control flow from general-purpose CPU processor. We develop a reliability-aware simulator for GPGPUs, it is built upon a cycle-accurate and open-source simulator, GPGPU-Sim [34]. We apply two major AVF calculation methodologies proposed in [20, 31] to identify the bits required for architecturally correct execution and their residency time in each structure to estimate the AVF. Using the framework, we profile the soft-error vulnerability of several key structures in SM (shown in Figure 2). Since the slots in the fetch unit and the instruction buffer have the same design: both hold the designated warp PC/instruction, the two structures manifest quite similar susceptibility to soft errors, the AVF of fetch unit is not presented in the figure. Moreover, Figure 2 does not show the AVF of L1 constant and texture caches because the studied workloads either do not or rarely use those two structures and their AVF is lower than 4%. The detailed experimental setups are illustrated in Section 4. In this study, we target on the reliability optimization on SRAM-based structures, improvement on combinational-logic based structures (e.g. streaming processor) is beyond the scope of this paper.

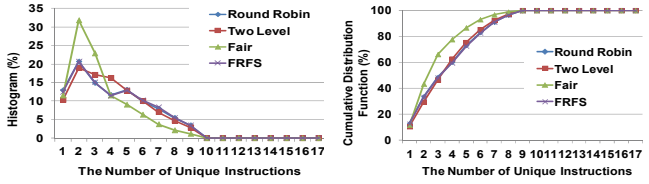
As Figure 2 shows, the instruction buffer and register files exhibit much higher AVF than other structures (this also matches the observation made in [39]), because they are highly utilized during the program execution while others are infrequently used. Take the branch unit as an example, only one entry of the warp stack is used when there is no branch divergence, and the stack entries are not fully utilized even when the warp diverges. As it shows, the AVF of the instruction buffer and register files can achieve up to 98% and 81%, respectively. Moreover, they are sizeable and occupy a large portion of the SM area: the instruction buffer has to hold all in-flight warps in the SM; the registers are much larger than those in traditional CPU processor as they have to support thousands of simultaneously active threads. For example, the registers size is reported to be 2MB in an NVIDIA Fermi GPU [26] and 6MB in AMD Cayman [27]. Therefore, the vulnerability of instruction buffer and registers significantly contributes to the stream multiprocessor SER robustness. In this paper, we focus on mitigating the two structures' vulnerability, it is the first and essential step to efficiently optimize the overall GPGPU reliability. Note that our observation and technique proposed for the instruction buffer is applicable to the fetch unit as well.

3.2. SAWP: Similarity-AWare Protection for the instruction buffer

3.2.1. Instruction similarity in the instruction buffer

The observation we make on the instruction buffer (I-Buffer) is the instruction similarity: a large number of instructions in the I-Buffer share the same PC and hold the identical information. As described in Section 2.1, the warps in the SM are interleaved at cycle-by-cycle basis. At every cycle, an instruction is issued for a warp which is selected in a round robin (RR) manner among the warps with instructions ready to execute. In the CUDA programming model, all threads in a kernel execute the same code and the same instruction from different warps will keep the same register information after decoding [36, 37], therefore, all instructions in the I-Buffer can be represented by just two instructions in the ideal case that warps proceed normally without stalls. In order to improve the GPGPU throughput, various warp scheduling policies have been explored: First-Ready First-Served; Fair [28] which issues instruction for the warp with minimum number of instructions executed; and two-level round-robin warp scheduling that effectively hides long memory access latency and improves the SPs utilization [36]. Since every warp has a dedicated entry in the I-Buffer, there will be an empty instruction slot when the instruction is issued, and only the following instruction from the same warp will be placed into that slot. Different from the simultaneous multithreading architecture, there is no resource contention among warps in SM which helps to control the progress difference among warps. As a result, even though some instructions are stalled in the I-Buffer by the branch divergence,

off-memory access transactions, and barriers; while some are granted the higher issue priority under the impact of the warp scheduling policy, a large amount of instructions in the instruction buffer still share the same information.



(a) (b)

Figure 3. (a) The histogram and (b) the cumulative distribution function of the number of unique instructions

Figure 3 (a) plots the histogram of the number of unique instructions in the I-Buffer under the RR, two-level, First-Ready First-Serve, and FAIR policies. The I-Buffer size is set as 32. At every cycle, we collect the amount of unique instructions, and present the statistics as the probabilistic-based distribution at the Y-axis. In addition, Figure 3 (b) profiles the corresponding cumulative distribution function (CDF). The results are averaged across the studied benchmarks. As shown in Figure 3 (a) and (b), the case with only two unique instructions accounts for the largest fraction of the program execution time under various scheduling policies: 21% for RR, 19% for two-level, 21% for First-Ready First-Served, and 32% for FAIR. And the maximal number of unique instructions is limited to 12. It implies that a small number of instructions (e.g. 12) are sufficient to describe all the instructions sitting in the instruction buffer through the entire program execution, and this observation is not affected by the warp scheduling policy.

As a conventional and straightforward fault-tolerance mechanism, ECC provides the full protection to the vulnerable structure but leading to a large power and area overhead. When extending this technique to the I-Buffer, a full ECC table to all the instructions is not necessary since most instructions in it are the same. The ECC table size can be significantly reduced without losing any error coverage.

3.2.2. Concept of SAWP

In order to cost-effectively mitigate the soft error rate in the instruction buffer, we propose Similarity-AWare Protection (SAWP). It leverages the instruction similarity to implement a small-size ECC table for the I-Buffer with low area and power overhead.

Based on our observation in Section 3.2.1., an ECC table with 12 entries is sufficient for a 32-entry instruction buffer to obtain the full error detection. As Figure 3 (a) shows, the distribution for the PC quantity drops greatly as the number increases from 2 to 12. Specifically, the case with 12 unique PCs only appears in less than 1% of the execution time. Figure 3 (b) further confirms that an 11-entry ECC table already has the capability to capture 99% of the PCs in the instruction buffer. Moreover, the area overhead increases proportionally to the ECC table size. A sensitivity analysis is required to justify the effectiveness on achieving the best trade-off between reliability and power&area overhead when changing the table size. Based on the comprehensive analysis on various table size options, we find that the 4-entry ECC table outperforms other designs, and it is adopted in SAWP.

The SAWP supports two major operations for every instruction in the instruction buffer: ECC generation and entry allocation, and error detection/correction.

3.2.2.1. ECC generation and entry allocation

In the full-size ECC table, each instruction in the I-Buffer has an allocated ECC entry. When a new instruction arrives at the I-Buffer, its ECC is written to the designated entry directly. While in our small-size ECC table, one ECC entry may be shared by multiple instructions. When an instruction enters the instruction buffer, it needs to find out the certain entry holding its ECC. In SAWP, one existing bit is attached to each I-Buffer entry to describe whether the ECC table has the instruction's ECC, and a two-bit index is added to specify its corresponding ECC entry. In addition, a field is attached to each ECC entry holding the number of certain I-Buffer entry that is under its protection. The newly arriving instruction will compare with four instructions in the I-Buffer based on the warp IDs in the ECC table. A hit implies that one ECC record can be re-used by the new instruction. Correspondingly, the ECC entry number will be written into the I-Buffer with the new instruction, and its ECC existing bit is set as "1". On the other hand, if there is no match, the ECC generation and ECC entry allocation requests will be sent to the ECC generator and the ECC table. If there is an idle entry in the ECC table, it will be allocated to the new instruction. Otherwise, an occupied entry has to be replaced to accept the newly generated ECC. In one sentence, the ECC generation and entry allocation is mainly composed of two parts: (1) instruction comparison and ECC generation; (2) ECC entry allocation and replacement.

(1) Instruction comparison and ECC generation

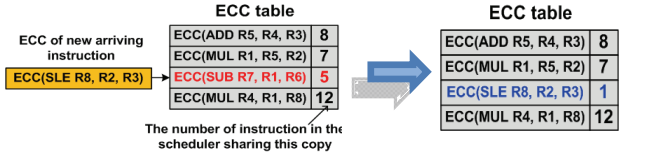
When a new instruction reaches the instruction buffer, the instruction comparison is triggered to determine if an ECC generation is required. Since the I-Buffer is vulnerable to soft errors, instructions in it can be erroneous and may affect the comparison correctness. In this study, we focus on the single-bit error model which has the first order impact on the failure rate in microprocessors [30]. One single-parity bit is used per I-Buffer entry to detect the erroneous instruction. The parity bit is checked during the instruction comparison. And the ECC generation request will be sent out only when there is a match with a fault-free instruction. It is possible that the new instruction receives miss while one matched instruction does exist among those four compared instructions but its bit is flipped due to the soft error. Similarly, the ECC table is vulnerable and the stored warp ID is likely to be erroneous and index to a different instruction for the comparison, leading to a miss as well. A new ECC entry will be allocated for the new instruction in both cases. It would reduce the error coverage because two identical ECC copies will appear in the ECC table, but it does not affect the error checking correctness.

(2) ECC entry allocation and replacement

At the cycle level, an ECC record may have only one corresponding instruction or even be shared by all the instructions in the I-Buffer. Because threads do not progress at 100% the same rate and instructions exhibit different residency latency in the instruction buffer. Intuitively, the ECC used by minimum number of instructions should be replaced by the newly generated ECC to achieve the best error coverage. However, it is possible that the new instruction belongs to a warp proceeding ahead/behind others, and its ECC is unlikely to be accessed in the following cycle; even worse, the previously evicted ECC may be generated again to serve the ECC entry allocation request from the next arriving instruction. Since the most recently inserted ECC only has one instruction in the I-Buffer, it becomes the one to be replaced. This results in a ping-pong effect which significantly reduces the number of instructions that can be protected and increases the power consumption due to the frequent ECC generation and write operations to the ECC table. Figure 4 shows an example of the ping-pong effect.

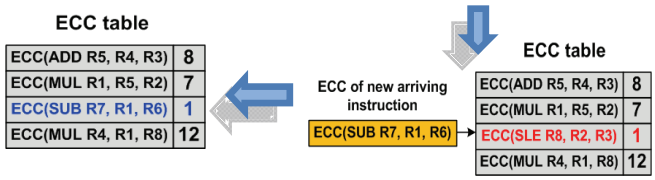
A simple solution to this effect is to use a threshold to control the entry replacement. When the amount of instructions sharing an

ECC exceeds the threshold, this ECC will not be evicted as it still provides the protection for numerous instructions. In this study, we set the threshold as four based on the detailed sensitivity analysis. Furthermore, an ECC will not be replaced when the warp that the new arriving instruction belongs to is far before/behind other warps. Because the identical instructions from other warps will not appear in a short time, the new ECC only protects a single instruction.



(a) An incoming instruction will replace the "SUB" instruction ECC with the minimum number of instructions (i.e. 5)

(b) The "SUB" ECC is evicted, ***and its corresponding five instructions lose the ECC to compare with when issued***



(d) The "SUB" ECC is inserted again, ***the ECC is frequently generated and the table is frequently written***

(c) The "SUB" ECC is requested by the new arriving instruction, the previous inserted ECC (the "SLE" ECC)has to be evicted

Figure 4. The ping-pong effect during ECC entry allocation and replacement

3.2.2.2. Error Detection/Correction

While an instruction is issued for the register read, the error detection/correction is triggered. Its parity bit is first checked for the error detection. If the faulty instruction's ECC exists in the ECC table, both the instruction and its ECC will be sent to the ECC checker for error correction. To make sure that an instruction retrieves its ECC correctly, the gate-sizing technique is applied to protect the existing bit and the two-bit index against the soft errors. A detected erroneous instruction will be flushed and re-executed. A bit flip in the ECC field can be easily detected and corrected in the ECC checker, even the ECC table is attacked by soft errors, it does not affect the error correction for the issued instructions.

3.2.3. SAWP Implementation

Figure 5 introduces the implementation of the instruction comparison, ECC entry allocation and replacement, and the error detection/correction in SAWP architecture. Each entry in the ECC table consists of three components: the ECC, the ID of certain I-Buffer entry it protects, and a counter to record the number of instructions currently in the I-Buffer sharing this ECC. There are 32 entries in the I-Buffer, five bits are used in each ID filed and counter. As described in Section 3.2.2., the single parity bit, 2-bit index and existing bit is added to each I-Buffer entry. As Figure 5 shows, when a new coming instruction is writing into the I-Buffer, (a) it is compared with four instructions in the I-Buffer based on the IDs kept in the ECC table. (b) The result analyzer accepts the comparison result and determines the next step towards (c) or (d).

(c) When there is a match with a fault-free instruction, it writes the matched instruction's index into the I-Buffer to build up the link between the new instruction and its ECC, and the counter in the ECC entry increases by one. To make sure that the ID field in the ECC table keeps the latest instruction information, the ID of the I-Buffer entry that holds the new instruction will write into the ID filed in the corresponding ECC entry as well.

(d-1) When there is a miss, the warp progress is checked (the numbers of instruction executed in each warp is evaluated), and an ECC entry replacement request is assigned when the warp that the new instruction belongs to keeps the similar rate with others. (d-2) Meanwhile, the counters in the ECC table are read out and compared with the pre-defined threshold (i.e. 4) to select an entry for eviction, and the ECC generation request is sent to the ECC generator. Note that the counters do not need error protection, because a faulty counter only affects the entry eviction, but has no impact on the correctness of the SAWP architecture. (d-3) There may be few instructions in the instruction buffer still pointing to the evicted entry. Therefore, the entry number broadcasts to the I-Buffer, and the existing bit in the matched indices is reset to "0". (d-4) Upon the completion of the ECC generation and entry replacement, the instruction's index, the ID field and the counter of the new ECC entry are updated correspondingly. And the existing bit is set to "0" when the instruction's ECC is not qualified to replace any entry in the ECC table, which indicates that the instruction is not protected by SAWP.

Note that the instruction comparison, ECC generation, and ECC entry allocation perform in parallel with the instruction writing to the I-Buffer, it does not introduce any delay to the critical path. Those steps will not finish in one cycle, it is pipelined so that the incoming instruction does not need to wait for the completion of the previous instruction's operations.

As shown in Figure 5, at the time that an instruction is moving out of the instruction buffer and its existing bit is "1", the counter of the its ECC entry decreases by 1. The entry is evicted once the counter equals to zero. (e) If the instruction is faulty based on the single parity bit, it is dropped in the pipeline. It will be sent to the ECC checker with its ECC to retrieve the correct instruction, which will then be re-issued into the pipeline. In SAWP, the error detection and correction perform simultaneously with the pipeline. They are pipelined so that the instruction issued in the following cycles can start the error checking while the previous instruction is still under error correction. When an instruction is issued, it takes a few cycles before it starts to update the register/memory (the pipeline usually consists of 24 stages in the SM [34]). This provides enough slack for SAWP to verify the instruction correctness, and only the faulty instruction needs to be dropped, no further action is required to restore the registers/memory states.

3.2.4. Overhead analysis

As described in Section 3.2.3., SAWP requires a 4-entry ECC table. Each ECC entry contains 7-bit ECC (we assume a single bit error model in this study), 5-bit ID field, and 5-bit counter. Moreover, 4 bits are attached to each slot in the instruction buffer. SAWP also uses some combinational logics including the ECC generator and checker, and the result analyzer (it is a simple multiplexer). Compared to the case implementing full-size ECC table, SAWP reduces the area overhead up to 12% based on our gate-level estimation. More importantly, whenever an instruction is writing into the I-Buffer, its ECC has to be generated and written into the table in the full-size ECC table which causes high power consumption. SAWP effectively limits the ECC generation times by leveraging the instruction similarity, thus, the power consumption decreases substantially.

3.3. SHARP: SHAred memory to Registers Protection

3.3.1. The key observation on the shared memory

In the GPGPU SM, the per-block resources (e.g. shared memory, registers) will not be released until the block completes execution. They limit the maximum number of blocks that can be

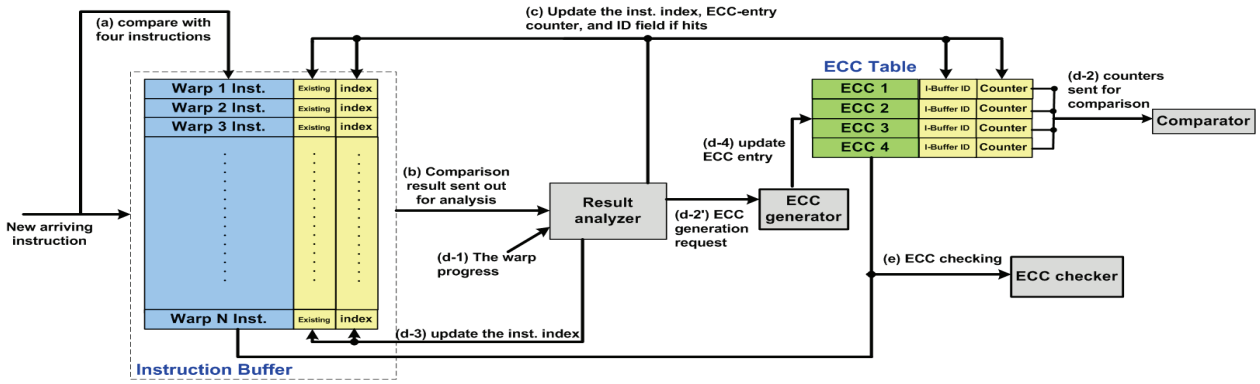


Figure 5. The implementation of SAWP

simultaneously assigned to an SM. Different per-block resources become the bottleneck during block allocation for kernels that have various resource requirements. Intuitively, the bottleneck structure is prone to be fully utilized and manifest high vulnerability. Interestingly, we observe the low utilization in the shared memory even it acts as the bottleneck resource. Shared memory is highly banked. The bank selected to hold a data value is determined by the data address, which leads to the unbalanced bank usage in a block. In other words, the number of blocks that each bank can support is different, and the minimum number finally limits the quantity of blocks the shared memory can support. Thus, even though shared memory becomes the resource bottleneck, most banks in it may be underutilized. Figure 6 presents the percentage of used entries in the shared memory for each benchmark (benchmark investigated in this study is listed in Section 4). On average, it is only 20%. In workloads whose block resource allocation is limited by the share memory (e.g. LPS, SP, SRAD), more than 50% entries are never be used during the entire execution time. For those used entries, they are written/read in a very short period and become free in majority of the time.

In summary, although shared memory is the software-managed cache for memory reuse, it is lightly utilized in many applications [29, 38]. First, memory reuse is limited by the nature of applications. For example, computation-intensive applications have little memory reuse. Second, shared memory needs to be synchronized to ensure access order among threads, and it has the bank conflict problem while addressing data. These increase the difficulty for program developers in efficiently using shared memory as the on-chip chip for global memory. Based on our analysis on a large set of widely-used GPGPU benchmarks, few benchmarks heavily use shared-memory.

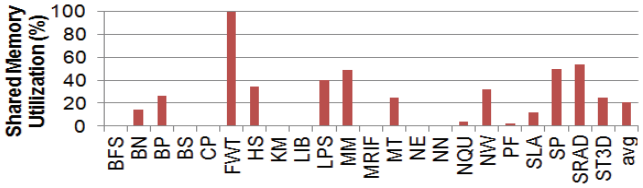


Figure 6. Shared memory utilization in percentage

3.3.2. The concept of SHARP

In order to improve registers SER robustness, a simple and direct approach is to implement the ECC for each register vector. When registers are not the resource bottleneck, each thread in the SM will be allocated more than sufficient amounts of registers, and some of them will be idle through the entire thread execution. Those idle register vectors can be used to store the ECCs of other active register vectors to reduce the SER. We name this method as register-to-register protection or RTRP as an abbreviation. RTRP could largely enhance register reliability when there are numerous free registers. However, it loses the benefits when all the available register vectors are used during the kernel execution. Even worse, the vulnerability of registers increases dramatically in benchmarks requiring heavy register utilizations. A technique is highly desired to optimize registers SER robustness when they face the severe vulnerability challenge. As we discussed in Section 3.3.1., shared memory contains many always-idle entries and exhibits low AVF, moreover, it is read/write-able and its access latency is comparable to the register access latency. Shared memory is a good candidate to keep the ECCs and provide the protection to fully-utilized registers. In this paper, we propose SHAred memory to Register Protection (SHARP) to intelligently store the ECCs of a set of register vectors into shared memory and efficiently mitigate register vulnerability.

3.3.2.1. Register selection for error protection

The register file AVF is the averaged ratio of each register's lifetime to the workload execution time [31], one can develop the 100% fault-tolerant registers by recording/checking every register vector's ECC once it is written/read. At every cycle, there are many register reads and writes, but shared memory usually serves no more than 16 access requests per cycle, thus, it is not feasible to perform the ECC protection for each single register vector through its lifetime. SHARP selectively protects a set of register vectors and meanwhile, maximizes the benefits by using the limited shared memory resources. [11] observes that the long-lived registers are the major contributors to the overall registers AVF. In other words, the register vulnerability will drop significantly if the long-lived registers are fault-free. When an off-chip memory access occurs in a thread, all threads within the warp have to stall until it finishes, which tremendously extends the lifetime of registers belonging to that warp. In SHARP, the ECC recording/checking is triggered when a warp starts/completes the long memory operations, and it is performed at the warp level, which means that all register vectors assigned to that warp will experience the ECC recording/checking process.

3.3.2.2. ECC mapping mechanism to share memory

We propose a novel ECC mapping strategy to achieve the fast ECC access in shared memory. In this study, we assume a single bit error model in each 32-bit register. Every register in a register vector requires 7 ECC bits, and one 32-bit shared memory entry can only record 4 registers' ECCs as one bit in the entry will be used to describe the status (busy or free) of those saved ECCs. Furthermore, a valid bit is attached to each entry to mark if the entry currently keeps a real value. As can be seen, one register vector requires 8 shared memory entries for its ECC.

In the shared memory, each bank can serve only one entry access per cycle, and 8 ECC access requests for one register vector may take 8 cycles to finish if they are all mapped to different

entries within the same bank. The access time decreases dramatically if the 8 requests distribute to different banks, which mainly depends on the mapping policy. Figure 7 illustrates the SHARP mapping mechanism by presenting an example of placing ECC of 4 registers into share memory. Note that they belong to the same register vector. As our default GPGPU configuration in Section 4 describes, the registers size is 64KB and each register vector number is 9-bit long. Every 4 consecutive registers in a register vector are gathered into one group, which is identified by attaching 3 bits to the least significant bit of the vector number (total 12 bits). To evenly distribute groups into each bank, the lowest 4 bits of the group identification number are used to index the shared memory bank (shared memory has 16 banks). And the remaining bits are used to locate the entry within the bank. Note that entry conflict occurs when two register groups map to the same entry. Basically, it would not happen within the warp based on our mapping mechanism, as long as the shared memory is large enough to save the register vectors' ECCs for just one warp, which is usually the case in current GPGPU microarchitecture design. The conflicts could only exist at inter-warp level.

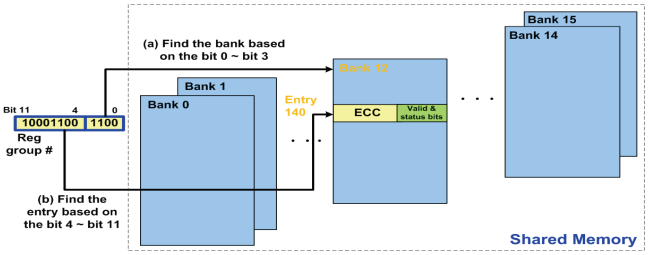


Figure 7. ECC mapping to shared memory

3.3.3. The implementation of SHARP

Figure 8 describes the implementation of SHARP. A request queue is attached to the register files, it is composed of a FIFO buffer and a warp ID table. The buffer keeps a number of requests for the warp-level ECC access, and the table records warp IDs whose registers are currently protected in the shared memory.

When a warp starts/completes an off-chip memory access (the coalesced intra-warp memory operations is treated as one access), an ECC recording/checking request with the warp ID are sent to the queue. When an ECC recording request arrives at the head of the buffer, the request queue reads its warp ID, performs the inter-warp ECC conflict examination to check if its mapped entries in the shared memory have already been occupied by another warp. It starts the recording if no conflict, the warp ID is also saved into the warp ID table; otherwise, the request is simply dropped. When the buffer head is an ECC checking request, the request queue enables the checking as long as its warp ID exists in the table.

Registers are highly banked (e.g. 16 banks) and each bank is equipped with a read port. The register vectors are interleaved across the banks to increase the likelihood that all the operands for an instruction can be fetched simultaneously. By running a large number of benchmarks, we observe that no more than 4 banks are accessed concurrently during 99% of the execution time. It implies that the read ports of a large number of register banks are idle and they could be used to read register contents during the ECC generating/checking without affecting the normal thread execution. Note that the ECC related operations are assigned a lower priority compared to the operands read required by threads for the same bank. It is the case in the shared memory as well. A waiting queue is attached to temporarily hold the register group numbers and values that are waiting for the available write/read port in shared memory banks. The register read stalls if the waiting queue is full and resumes once it has free slots.

To perform an ECC recording, the register contents are sent to the ECC generator to produce ECCs which are further buffered in the waiting queue for write operations, meanwhile, the corresponding register group numbers are saved in the waiting queue for ECC mapping. Note that a real value is written to a shared memory entry normally and its valid bit is set as "1", a future ECC write/read to this entry will fail/miss to guarantee the program execution correctness. When the valid bit of the shared memory entry is "0" and the ECC recording completes successfully, the status bits in the entries are set as busy.

An ECC checking is enabled when the memory access completes, the register group numbers and contents move into the waiting queue and wait for their ECCs from the shared memory if exist (i.e. the valid bit of the shared memory entry is "0" and the status bit is busy) to perform the error checking in the ECC checker. The register group number and correct value will be sent back to the registers when an error is detected. Afterwards, the obtained off-chip memory values are written back to the registers, and the request queue is updated to keep the up-to-date warp ID information.

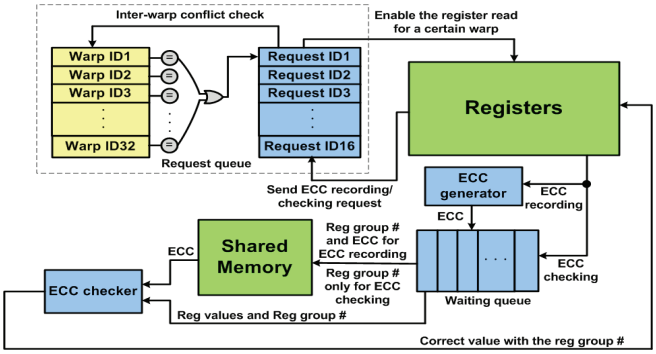


Figure 8. The detailed design of SHARP

The request queue and waiting queue are also vulnerable to soft errors. A faulty request may lead to different warp registers mapping to the share memory. And an error in the waiting queue causes wrong data written/read during the ECC recording/checking. We apply the gate-sizing technique to protect the two structures. Note that a bit flip in the ECC field can be easily corrected in the ECC checker, although shared memory may be attacked by the soft errors, it does not affect the correctness of SHARP.

3.3.4. Overhead analysis

Both registers and shared memory are highly banked, the warp-level register read and ECC write/read usually complete in several cycles, moreover, it takes a few cycles to finish an ECC recording/checking operation. Since the ECC recording is not in the SM critical path, it does not introduce any extra delay. Even though the stalled warp will not proceed during the ECC checking, SHARP introduces negligible performance penalty as other ready warps can still be issued to hide that ECC checking latency.

It is possible that a warp is stalled by a sequence of long latency memory operations that cannot be coalesced, and the completion of one operation only updates one or very few registers. Repeatedly performing the ECC recording and checking per memory access for that warp results in severe power overhead. To achieve the good trade-off between reliability and power, only one pair of ECC recording and checking requests is issued at the occurrence of the first memory access and at the completion of the last memory access, respectively. When one access finishes, the updated register group number are directly sent to the shared memory, their ECC status bits in the mapped entries are simply reset to free, and they will be skipped during the ECC checking.

The buffer size in the request queue determines the number of ECC requests can be accepted. A large buffer is able to keep all the requests but may significantly delay a warp ECC checking and increase its stall time as requests are performed at the FIFO order. We set the buffer size as 16 in this study, and each entry contains 1 bit describing request type, and 5-bit warp ID. In addition, the warp ID table has 32 entries, each with 5 bit ID. In SHARP, as long as a warp ECC recording is performed, its ECC checking needs to be executed to delete the corresponding warp ID saved in the request queue. Otherwise, the request queue will not accept the following ECC related requests issued by the same warp due to the conflict checking. The buffer may be full when a checking request is issued, a latest inserted recording request will be dropped in that case. In this study, the waiting queue contains 8 140-bit (12-bit register group number and 4X32-bit content) entries which is large enough to accept data from the registers/ECC generator and provide data to shared memory/ECC checker at every cycle. Overall, the attached buffers and combinational logics introduce 5% hardware overhead to the register files.

4. Experimental Methodology

We use the developed GPGPU reliability-aware simulator based on GPGPU-Sim to obtain the GPGPU reliability, performance, and power statistics. Our baseline GPGPU configuration is set as follows: there are 28 SMs in the GPU, SM pipeline width is 32, warp size is 32, each SM supports 1024 threads and 8 blocks at most, each SM contains 16K 32-bit registers, 16KB shared memory, 8KB constant cache, and 64KB texture cache, the warp scheduler applies the round robin scheduling policy, the immediate post-dominator reconvergence [18] is used to handle the branch divergences; the GPU includes 8 DRAM controllers, each controller has a 32-entry input buffer, and applies out-of-order first-ready first-come first-serve scheduling policy; the interconnect topologies is Mesh, and the dimension order routing algorithm is used in the interconnect. We collect a large set of available GPGPU workloads from Nvidia CUDA SDK [23], Rodinia Benchmark [24], Parboil Benchmark [25] and some third party applications. The workloads show significant diversity according to their kernel characteristics, divergence characteristics, memory access patterns, and so on.

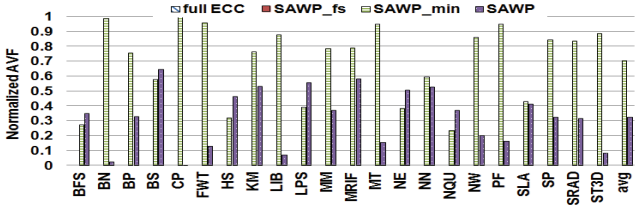
We use AVF as the basic metrics to estimate the structure soft error vulnerability. To estimate the power consumption, we use HSPICE to build the power model for the combinational logics related to ECC generation and checking, and modify the energy analysis tool CACTI [19] to calculate the power of SRAM-based structures such as the I-Buffer, registers, ECC table, the added request queue and waiting queue, and so on. We collect the statistics for instruction comparison, ECC generation and checking via the microarchitecture simulation, they are combined with the developed power model to obtain the dynamic and static power of the investigated structures. Our work is based on the 40nm processing technology which is applied in recently delivered GPGPUs.

5. Evaluation

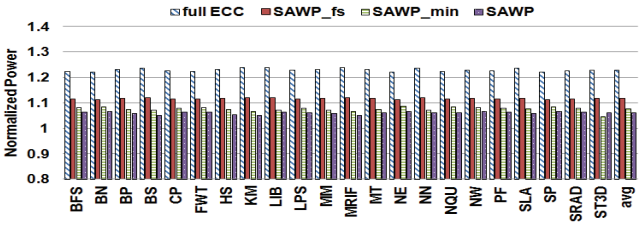
5.1. Effectiveness of SAWP

We compare SAWP with three schemes as follows: full-size ECC table (full_ECC) which assigns an ECC entry for each instruction; full-size SAWP (SAWP_fs) which applies a 12-entry ECC table in SAWP to cover all errors in the instruction buffer; and SAWP_min which applies 4-entry ECC table but simply evicting the ECC record with minimum number of instructions during the ECC entry allocation stage in SAWP. Figure 9 (a) and (b) show the instruction buffer AVF and power results, respectively, across the studied benchmarks while the four schemes are applied. The Round Robin warp scheduling policy is used in all the four techniques. Results are averaged across the SMs in each benchmark, and normalized to the baseline case without any soft error protection mechanism.

As Figure 9 (a) demonstrates, full_ECC and SAWP_fs achieves the 0% AVF since they provide the protection for every instruction. Although SAWP does not fully protect the instruction buffer, it shows the strong capability of reducing the vulnerability. On average across the benchmarks, the instruction buffer AVF decreases 68% under SAWP compared to the baseline case. Especially, as shown in Figure 2, the I-Buffer suffers extremely high AVF (e.g. above 65%) in FWT and MT when no protection scheme is triggered, while our SAWP enhances the reliability up to 90%. Note that the AVF reduction under SAWP varies across benchmarks. Because the instruction similarity in the instruction buffer differs in various workloads, it greatly affects the quantity of instructions can be protected by the ECC table. Take FWT as an example, all instructions in the I-Buffer are identical during 33% of the execution time; To the contrary, instructions exhibit weak similarity characteristic in BS, as a result, the vulnerability reduction between these two benchmarks differs significantly.



(a) Instruction buffer AVF (full_ECC and SAWP_fs covers all errors, the AVF is 0% for those two mechanisms)



(b) Normalized instruction buffer power

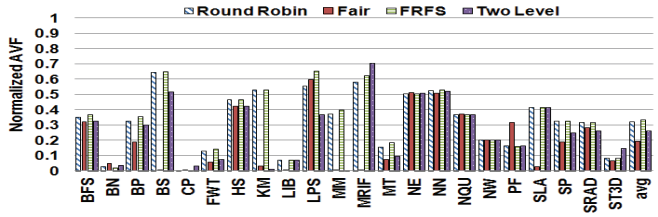
Figure 9. The effectiveness of full_ECC, SAWP_fs, SAWP_min, and SAWP

In addition, SAWP_min is able to reduce the instruction buffer AVF by 30%. It underperforms SAWP because the quantity of instructions under the protection decreases during the frequent entry eviction and allocation which downsizes the SAWP_min efficiency in fault tolerance. One may notice that the SWAP_min achieves a little lower AVF than SAWP in several benchmarks (e.g. BFS, LPS). Because they include a large number of branch divergences, there are increasing unique instructions in the I-Buffer, the frequent eviction in the ECC table can help to capture more unique instructions and provide the protection.

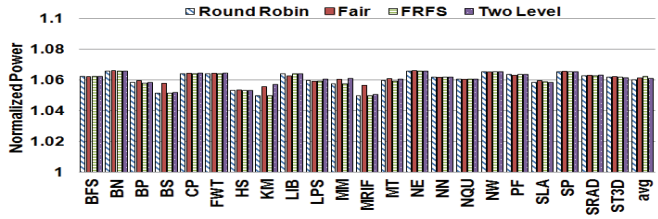
As shown in Figure 9 (b), on average, the SAWP_fs is able to reduce the I-Buffer power by 11% when compared to full_ECC whose power consumption is 23% higher than the baseline case. Because full_ECC requires ECC generation for each single instruction, and the full-size ECC table causes a higher area overhead. While SAWP_fs effectively reduces the ECC generation frequency since multiple instructions can share just one ECC record. SAWP_min outperforms SAWP_fs and further reduces the power consumption by 4%, it uses smaller ECC table and performs ECC generation and checking less frequently by scarifying the error coverage to some degree. SAWP achieves the lowest power

consumption in all the investigated protection mechanisms. It reduces the power consumption by 17% compared to full_ECC. In a conclusion, when compared to SAWP_min, SAWP intelligently chooses the appropriate instructions for soft error protection and achieve the win-win senario: reducing the ECC generation frequency and ECC table access times to save power, and meanwhile increase the error coverage.

Various techniques have been proposed on warp scheduling to improve the GPGPU throughput, such as FAIR, First-Ready First-Served, and two-level round-robin. Figure 10 (a) and (b) show the normalized instruction buffer AVF and normalized power under SAWP when those optimization schemes are enabled. The results of the case when the Round Robin policy is applied are shown in the figure as well for comparison purpose. As Figure 10 (a) shows, they introduce the positive effect to the I-Buffer vulnerability. Especially, the AVF reduces 81% under the impact of FAIR. FAIR issues instructions from various warps in a fair manner, it maintains the uniform progress among warps which enhances the instruction similarity and correspondingly, the error coverage. Furthermore, as shown in Figure 10 (b), the power consumption has little change under various warp scheduling policies. Therefore, the capability of SAWP on improving the instruction buffer vulnerability with little power consumption is not affected (and even enhanced) by the GPGPU performance oriented techniques.



(a) Normalized instruction buffer AVF



(b) Normalized instruction buffer power

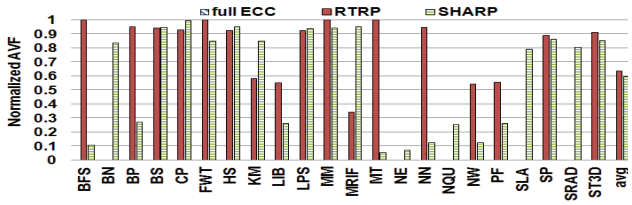
Figure 10. The effectiveness of SAWP when FAIR, First-Ready First-Served (FRFS), and Two-level round-robin (Two Level) are triggered

5.2. Effectiveness of SHARP

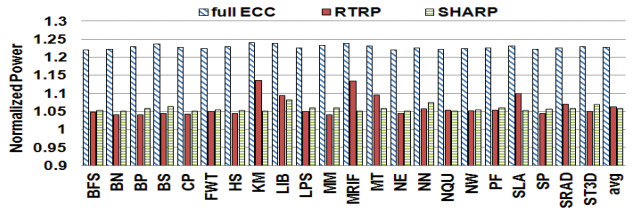
We compare SHARP with two register vulnerability optimization mechanisms: full ECC protection (Full_ECC) which introduces an additional table to keep the ECC for every single register, this technique has been applied in Nvidia Fermi GPUs [26]; register-to-register protection (RTRP) which leverages the free registers to keep the ECCs. Figure 11 (a-b) shows the normalized register AVF and power results when using the three techniques with a set of studied benchmarks. The results are normalized to the case without any protection mechanism.

As Figure 11 (a) shows, Full_ECC provides the 100% protection to registers, and the AVF is zero. Both RTRP and SHARP improve the register reliability significantly. On average, RTRP and SHARP reduce AVF 37% and 41%, respectively, compared to the baseline case. The effectiveness of RTRP in fault tolerance is largely affected by the register utilization. For example, more than 90% registers are used in BP, there are few free registers to perform the ECC protection. As a result, RTRP gains limited vulnerability mitigation (around 5% AVF reduction). Worse, the registers suffer severe vulnerability challenge in the two benchmarks due to the heavy usage, and an efficient error protection mechanism is even more emergent. In BP, the shared memory is idle 80% of the entire execution time, and there are numerous off-chip memory transactions, SHARP intelligently leverages those characteristics and successfully achieves the high error coverage (AVF decreases by 75%).

As shown in Figure 11 (b), on average, SHARP outperforms full_ECC by reducing the register power consumption from 124% to 106%. This is caused by the tremendous power and area overhead introduced in full_ECC. Full_ECC requires a hard structure to buffer ECCs, although a 7-bit ECC is able to protect the 32-bit register, the extra area overhead to the registers jumps to 25%. More importantly, no matter the registers have short or long lifetime, full_ECC treats them the same and perform the ECC recording/checking whenever a register is accessed, it substantially increases the power consumption especially in benchmarks requiring numerous registers. As Figure 11 (b) shows, on average, the power consumption in RTRP is 2% higher than SHARP. Because the RTRP trigger frequency is determined by the RF utilization. It consumes large power on ECC generation and checking when RF is lightly used (e.g. KM).



(a) Normalized registers AVF
(full_ECC covers all errors, the AVF is 0%)



(b) Normalized registers power

Figure 11. The effectiveness of Full_ECC, RTRP, and SHARP

Note that both SAWP and SHARP outperform other mechanisms compared in this study (e.g. Full_ECC) regarding to the trade-offs between power and reliability. We use a ratio between the power overhead and AVF reduction to describe how efficiently the technique can trade the extra power in gaining reliability optimization, and a lower value implies a better technique. SAWP trades 6% power overhead to gain 68% AVF reduction, the ratio between those two factors (i.e. 6%/68%=0.09) is much lower than that of Full_ECC, which is 23%/100%=0.23 (Full-ECC requires 23% power overhead to achieve 100% AVF reduction). It indicates that SAWP is able to achieve higher AVF reduction compared to full_ECC given the same amount of power budget. In addition, although SHARP has fewer opportunities to trigger the ECC protection in computation-intensive benchmarks (e.g. BS, CP, MM), it also achieves better trade-off (i.e. 0.14) between power and reliability than that (i.e. 0.24) of full-ECC.

6. Related Work

There have been various studies on protecting vulnerability hot-spots in CPUs via software/hardware-based redundancy. Montesinos et al. [11] explore ParShield which selectively protects

a subset of the registers by generating, storing, and checking the ECCs of only the registers with long lifetime. Blome et al. [12] proposes a register value cache that holds duplicates of live register values. Feng et al. [13] leverage the symptom based detection and selective instruction duplication to minimize user-visible failures induced by soft errors. Slick [14] avoids the redundancy for results predictable instructions to improve the performance whiling running redundant multithreading. However, they mainly target on CPUs and largely ignore the GPGPU architecture.

Both hardware- and software-based redundancy has been proposed to optimize GPGPU vulnerability. Sheaffer et al. [5] explore the concept of the sphere of replication on GPGPU processors, and present a hardware redundancy-based approach to create a reliable GPU with no performance loss. Dimitrov [8] investigate three software approaches to perform redundant execution for GPGPU reliability. Checkpointing is a widely used protection mechanism in CPU processors, it has been applied to enhance GPGPU robustness as well. Maruyama et al. [9] propose a high-performance software framework to enhance GPU with DRAM fault tolerance. It leverages light-weight data coding for error detection and checkpointing for recovery. Solano-Quinde et al. [32] propose an application-level checkpoint scheme for GPGPU systems, and explore the computation-communication overlapping to reduce the checkpoint overhead. In our study, we develop the cost-effective protection schemes based on our two key observations on the GPGPU microarchitecture structures (i.e. instruction buffer and shared memory). Recently, Nathan et al. [33] develop Argus-G, it implements control flow, dataflow and computation checkers in the GPGPU stream multiprocessor for low cost error detection. Yim et al. [10] use a fault injection tool to analyze the error resiliency of GPGPU platforms, and strategically place customized error detectors in the source code of GPU applications to tolerate errors. Both the two schemes are orthogonal to our techniques.

7. Conclusions

With their strong computing power and improved programmability, GPGPUs emerge as highly-efficient devices for a wide range of parallel applications. Meanwhile, GPGPU with hundreds of cores integrated in a single chip are highly vulnerable to the soft error strikes. This work aims to protect GPGPU microarchitecture against soft errors. We find that two SRAM-based structures (i.e. instruction buffer and registers) are prone to be the reliability hot-spots in the GPGPUs, and take advantage of the GPGPU microarchitecture characteristics to explore the cost-effective protection techniques for them. We propose the similarity-aware protection (SAWP) scheme which leverages the instruction similarity to provide the near-full protection for the instruction buffer with little power and area overhead. We further find that the shared memory are significantly under-utilized, and propose shared memory to registers protection (SHARP) which leverages the idle shared memory to keep the registers ECCs. Experimental results show that both SAWP and SHARP have the strong capability in fault tolerance and meanwhile achieving the low power consumption. SAWP reduces instruction buffer AVF by 68%, and SHARP reduces register AVF by 41% when compared to the case without any protection scheme. Moreover, SAWP (SHARP) is able to achieve similar AVF as the full ECC protection mechanism with 17% (18%) reduction on power.

Acknowledgement

This work is supported by the National Science Foundation under Award No. EPS - 0903806 and matching support from the State of Kansas through the Kansas Board of Regents.

References

[1] GeForce 8800 & NVIDIA CUDA: A New Architecture for Computing on the GPU, NVIDA Corporation, 2006.
[2] NVIDIA CUDA™ Programming Guide Version 2.3.1, Nvidia Corporation, 2009.
[3] Advanced Micro Devices, Inc. AMD Brook+. http://ati.amd.com/technology/streamcomputing/AMD-Brookplus.pdf.
[4] Khronos. Opencl – the open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/
[5] J. Sheaffer, D. Luebke, and K. Skadron, A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors, In Proceedings of Graphics Hardware 2007.
[6] N. Wang and S. Patel, ReStore: Symptom Based Soft Error Detection in Microprocessors, In Proceedings of DSN, 2005.
[7] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, Techniques to reduce the soft error rate of a high-performance microprocessor, In Proceedings of ISCA, 2004.
[8] M. Dimitrov, M. Mantor, and H. Zhou, Understanding Software Approaches for GPGPU Reliability, In Proceedings of GPGPU-2, 2009.
[9] N. Maruyama, A. Nukada, and S. Matsuoka, A High performance Fault-Tolerant Software Framework for Memory on Commodity GPUs, In Proceedings of IPDPS, 2010.
[10] K. Yim and R. Iyer, Hauberk: Lightweight silent data corruption error detectors for GPGPU, In Proceedings of IPDPS, 2011.
[11] P. Montesinos, W. Liu, and J. Torrellas, Using register lifetime predictions to protect register files against soft errors, In Proceedings of DSN, 2007.
[12] J. A. Blome, S. Gupta, S. Feng, S. Mahlke, and D. Bradley, Cost efficient soft error protection for embedded microprocessors, In Proceedings of CASES, 2006.
[13] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, Shoestring: probabilistic soft error reliability on the cheap, In Proceedings of ASPLOS, 2010.
[14] A. Parashar, A. Sivasubramaniam, S. Curumurthi, SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading, In Proceedings of ASPLOS, 2006.
[15] http://www.top500.org/blog/2009/05/20/top_trends_high_performance_computing
[16] M. A. Gomaa and T. N. Vijaykumar, Opportunistic transient-fault detection, In Proceedings of ISCA, 2005.
[17] N. Soundararajan, A. Parashar, A. Sivasubramaniam, Mechanisms for Bounding Vulnerabilities of Processor Structures, In Proceedings of ISCA, 2007.
[18] S. S.Muchnick. Advanced Compiler Design and Implementation. Morgan Kaufmanns, 1997.
[19] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi. Cacti 5.1. HP Labs, Tech. Rep. 2008.
[20] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor, In Proceedings of MICRO, 2003.
[21] X. Fu, T. Li, and J. Fortes, Sim-SODA: A Unified Framework for Architectural Level Software Reliability Analysis, Workshop on Modeling, Benchmarking and Simulation, 2006.
[22] M. Li, P. Ramachandran, S. Adve, V. Adve, and Y. Zhou, SWAT: An Error Resilient System, In Proceedings of SELSE, 2008.
[23] http://www.nvidia.com/object/cuda_sdks.html
[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing, In Proceedings of IISWC, 2009.
[25] Parboil Benchmark suite. URL: http://impact.crhc.illinois.edu/ parboil.php.
[26] D. Kanter. Inside Fermi: Nvidia's HPC Push, 2009.http://www.realworldtech.com/page.cfm?ArticleID=RWT093009110932.
[27] D. Kanter. AMD's Cayman GPU architecture, 2010.http://realworldtech.com/page.cfm?ArticleID=RWT121410213827.
[28] N. B. Lakshminarayana, H. Kim, Effect of Instruction Fetch and Memory Scheduling on GPU Performance, Workshop on Language, Compiler, and Architecture Support for GPGPU, 2010.
[29] Y. Yang, P. Xiang, J. Kong, and H. Zhou, A GPGPU Compiler for Memory Optimization and Parallelism Management, In Proceedings of PLDI, 2010.
[30] S.S.Mukherjee, J. Emer, and S.K.Reinhardt, The Soft Error Problem: An Architectural Perspective, In Proceedings of HPCA, 2005.
[31] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. Mukherjee, R. Rangan, Computing Architectural Vulnerability Factors for Address-Based Structures, In Proceedings of ISCA, 2005.
[32] L. Solano-Quinde, B. Bode, and A. Somani, Coarse Grain Computation-Communication Overlap for Efficient Application-Level Checkpointing for GPUs, In Proceedings of International Conference on Electro/Information Technology (EIT), 2010.
[33] R. Nathan, D. J. Sorin, Argus-G: A Low-Cost Error Detection Scheme for GPGPUs, Workshop on Resilient Architectures (WRA), 2010.
[34] A. Bakhoda, G.L. Yuan, W. W. L. Fung, H. Wong, Tor M. Aamodt, Analyzing CUDA Workloads Using a Detailed GPU Simulator, In Proceedings of ISPASS, 2009.
[35] W. W. L. Fung and T. Aamodt. Thread Block Compaction for Efficient SIMT Control Flow. In Proceedings of HPCA, 2011.
[36] V. Narasiman, C. J. Lee, M. Shebanow, R. Miftakhutdinov, O. Mutlu, and Y. N. Patt. Improving GPU Performance via Large Warps and Two-Level Warp Scheduling. In Proceedings of MICRO, 2011.
[37] W. W. L. Fung, I. Sham, G. Yuan, and T. M. Aamodt, Dynamic Warp Formation and Scheduling for Efficient GPU Control Flow, In Proceedings of MICRO, 2007.
[38] Y. Yang, P. Xiang, M. Mantor, N. Rubin, and H. Zhou, Shared Memory Multiplexing: A Novel Way to Improve GPGPU Performance, In Proceedings of PACT, 2012.
[39] N. Farazmand, R. Ubal, D. Kaeli, Statistical Fault Injection-Based AVF Analysis of a GPU Architecture, In Proceedings of SELSE, 2012.