Precision-Aware Soft Error Protection for GPUs

David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti
Department of Electrical and Computer Engineering
University of Wisconsin–Madison
palframan@wisc.edu, nskim3@wisc.edu, mikko@engr.wisc.edu

Abstract

With the advent of general-purpose GPU computing, it is becoming increasingly desirable to protect GPUs from soft errors. For high computation throughout, GPUs must store a significant amount of state and have many execution units. The high power and area costs of full protection from soft errors make selective protection techniques attractive. Such approaches provide maximum error coverage within a fixed area or power limit, but typically treat all errors equally. We observe that for many floating-point-intensive GPGPU applications, small magnitude errors may have little effect on results, while large magnitude errors can be amplified to have a significant negative impact. We therefore propose a novel precision-aware protection approach for the GPU execution logic and register file to mitigate large magnitude errors. We also propose an architecture modification to optimize error coverage for integer computations. Our approach combines selective logic hardening, targeted checker circuits, and intelligent register file encoding for best error protection. We demonstrate that our approach can reduce the mean error magnitude by up to 87% compared to a traditional selective protection approach with the same overhead.

1. Introduction

Though technology scaling allows the fabrication of increasingly efficient and complex processors, it also introduces a number of reliability challenges that must be overcome. Among these, soft errors from particle strikes are a growing concern that must be addressed to guarantee reliable operation. A soft error occurs when an energetic particle strikes a vulnerable circuit node, depositing charge in its wake. The absorbed charge can result in bit flips in storage elements or transient pulses in combinational logic, either of which can lead to silent data corruption. Historically, soft errors were not a concern in GPUs, since they were used primarily for graphics. With the advent of general-purpose computing on GPUs (GPGPU), however, fault tolerance in these devices is becoming increasingly important [12, 21].

Because GPUs have many vulnerable execution units and storage elements, full protection from soft errors can incur non-trivial power and area overheads. To maintain reasonable overheads, it is often desirable to employ partial error protection in which the degree of fault tolerance can be traded off with power or area overhead. The simplest way to implement partial error protection is to protect only the elements or logic that are most vulnerable to soft errors. For instance, the architectural vulnerability factor (AVF) can be computed to determine which components are most cost-effective to protect [19]. Such partial protection techniques consider all errors to be equally bad, and simply attempt to mitigate as many errors as possible with a given hardware budget. Many GPGPU applications perform image processing tasks or other tasks that employ a large number of floating-point (FP) computations. Such applications are often tolerant of small errors but sensitive to larger magnitude errors. In this case, not all errors are equally bad, and we therefore propose an approach that is specifically tailored to protect against large *magnitude* errors.

In this paper, we present a precision-aware soft error protection scheme for the GPU execution logic and the register file that intelligently combines selective gate hardening, an inexpensive checker circuit, and precision-aware encoding to dramatically improve soft-error resilience with very low overhead. We extend the benefit to integer (INT) computations through a novel architecture modification in which integers are treated similarly to FP numbers. We demonstrate that precision-aware protection incurs low overhead and results in negligible error magnitudes in the event of a soft error, compared to approaches that do not target error magnitude. We show that our technique reduces error by up to 87% for 5% area overhead compared to a more traditional approach with the same area overhead. This paper includes the following contributions:

- Discussion and analysis of precision-aware soft error protection for GPU execution logic using gate hardening and significand checking;
- Presentation of a hardening/encoding protection approach for the register file to limit error magnitude;
- Discussion of architecture modifications to extend maximum protection to integer computations through a hybrid INT-FP representation.

The remainder of this paper is organized as follows. Section 2 discusses prior proposals for soft error mitigation and motivates precision-aware protection. Section 3 details our approach for selective protection in execution logic and the register file. Section 4 presents the benefits of our protection scheme compared to traditional approaches. Finally, Section 5 concludes the paper.

2. Background and Motivation

In this work, we focus on protecting two primary GPU components: the execution units and register file. In a modern GPU streaming multiprocessor (SM), the execution units are primarily comprised of logic required to perform FP operations. Since this logic is mostly combinational, it can pose more of a challenge to protect than an SRAM structure with a more regular layout. The simplest way to protect combinational logic is duplication through dual modular redundancy (DMR) or triple modular redundancy (TMR). Such solutions, however, incur prohibitively high area and power overheads of over 200% or 300%, respectively. Due to the high overhead of logic duplication, time redundancy is often proposed as an alternative solution with a lower area and power footprint [24]. This technique checks for erroneous transitions at the output of a circuit to determine if an incorrect value may have been latched. Time redundancy, however, may incur a performance penalty and is not able to detect long-duration transients, which may become more prevalent as technology scales [16]. For this reason, we consider only checker circuits and hardening approaches to soft error mitigation in this paper.

To protect against particle strikes, gate hardening resizes or modifies gates to make them more resilient. Partial gate hardening can be done to reduce costs, such that only the most vulnerable gates are modified [23, 28, 29, 36]. As an alternative or in addition to hardening logic gates against soft errors, a checker unit can be used to detect errors and initiate a corrective action. Checkers such as those using residue codes, however, can incur high overheads as they provide near-total protection [17]. Additionally, such codes are most well-suited for arithmetic logic, and not arbitrary combinational logic. Other proposals suggest approximate checker circuits to address these issues by partially protecting combinational logic [4, 30]. These techniques only consider error coverage, however, and have no concept of error magnitude. To mitigate large magnitude FP errors, prior work has observed that the exponent is fairly inexpensive to check [18]. Since exponent correctness is an essential step to reducing error magnitude, our design builds on such an exponent checker. We then enhance error coverage through precision-aware protection of the significand logic.

GPUs rely on large register files to store the program state for extensive multithreading. Modern GPU architectures can provide protection for the register file through error-correcting codes (ECC) [22]. The most commonly-used codes are capable of single error correction and double error detection (SECDED), and require that each SRAM entry be augmented with a number of parity bits. For our precision-aware protection approach, we use robust SRAM cells instead of standard cells for the most important bits. If all bits were protected in this manner, the cost would be significant, but we are only concerned with the bits that would result in the largest errors if flipped. Thus, precision-aware protection can incur less overhead than ECC while providing protection against multiple most-significant bit (MSB) errors.

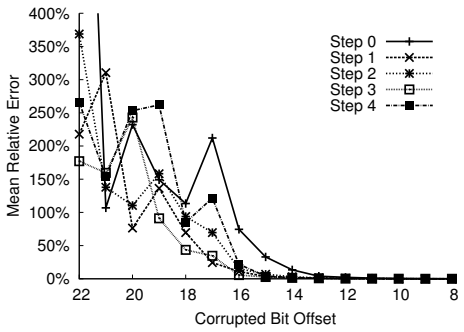


Figure 1. Mean error in the option prices computed by the Black-Scholes algorithm when different bits and computation steps are corrupted. Bit 22 is the highest order mantissa bit.

Since many applications can tolerate smaller or non-critical errors, some prior works attempt to avoid only critical errors. In the multimedia domain, for instance, Polian *et al.* analyze soft errors in a MPEG-2 motion estimator circuit, and identify errors as either critical or uncritical [27]. Similarly, other work suggests partitioning the cache such that only critical data is protected [14]. In both of these cases, the binary classification of errors as either critical or uncritical does not consider a finer gradient of error importance as we do in this work. Other error detection techniques for GPUs attempt to protect against all errors, which can be costly [12, 21].

As an example of a GPGPU application that can tolerate small magnitude errors, we look at Black-Scholes, a FP-intensive benchmark [26]. This benchmark computes solutions to the Black-Scholes partial differential equation, which predicts the price of stock options over time. To analyze the potential impact of a fault during different phases of this application, we divide the computation into steps comprising an equal number of operations. For each experiment, we choose a computation step to corrupt. For the intermediate value corresponding to the computation step, we then choose a bit offset in the mantissa to invert. This corrupted value is then used to complete the computation, and we note the error induced in the program output. To observe the impact of corruption at different bit offsets in the mantissa, we repeat this experiment at all computation steps and for all bit offsets. For every scenario, we compute the mean error across 4096 computations.

Figure 1 shows the result of this experiment. Here, bit offset 22 is the MSB of the mantissa, since 32-bit FP val-

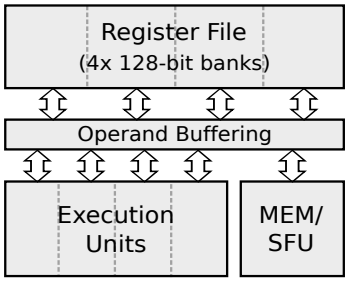


Figure 2. Modern GPU SIMT cluster. Each cluster contains four execution units.

ues were used. The Y axis shows the mean relative error in the calculated option price across all computations. Note that the maximum injected error is only 50% because the actual MSB of the fraction is implicit in the FP representation. Despite this limit, we observe upwards of 1000% error when bit 22 is corrupted. Even faults injected into bit 18, corresponding to an error of about 3%, can be inflated by the Black-Scholes algorithm to result in 100% error. Faults injected below bit 12, however, have a negligible impact on the computed option prices. Thus, when considering partial soft error protection for FP logic, it makes sense to consider error magnitude. This is particularly true for semi-error tolerant applications such as Black-Scholes and many other graphics and GPGPU applications.

3. Precision-Aware Protection

In this section, we discuss our proposed SM enhancements that provide cost-effective protection against large magnitude errors resulting from particle strikes. We employ targeted logic hardening which can be complemented or enhanced with low-cost checker circuits. We begin by considering execution logic protection. We present a standard partial logic hardening approach that we use as our baseline, and then introduce our precision-aware hardening technique. We also show how to achieve similar coverage using a significand checker circuit and discuss how to best use the same error protection for both FP and INT computations. Finally, we discuss how these ideas can be applied to the register file.

3.1. Execution Unit Protection

In this work, we assume a GPU architecture that utilizes a fused multiply-add (FMA) FP unit as the basic execution block. As shown in Figure 2, each SM contains multiple execution units divided into SIMT clusters of four execution units each [7]. The execution units in the cluster share four register file banks. The FMA execution unit is shown in more detail in Figure 3. Such a unit has the ability to perform both FP and INT operations and is employed in various existing designs [1, 8, 15]. When used for FP com-

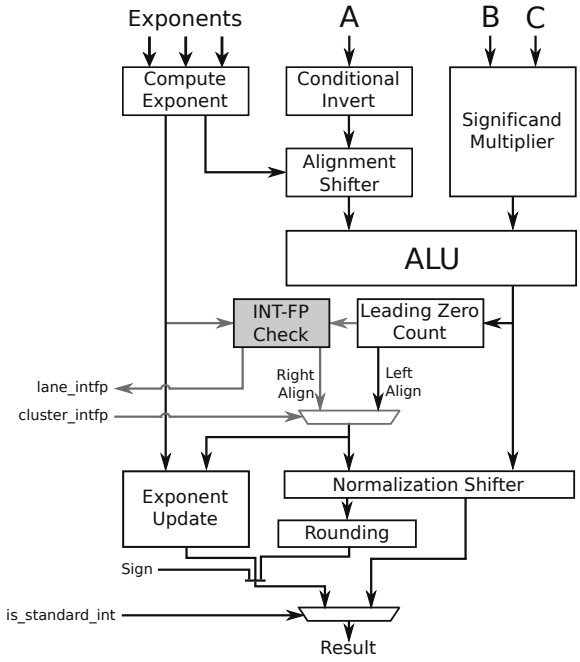


Figure 3. Fused multiply-add execution unit capable of INT and FP operations. In this work, we add the shaded logic to enable conversion to the INT-FP format discussed in Section 3.1.4.

putations, the significands of operands B and C are multiplied while the exponent computation unit sums their exponents and compares the result with the exponent of operand A. Based on the exponent comparison, the shifter aligns the radix points of the product and operand A before they are passed to the ALU, which contains the adder. Once the addition operation has been performed by the ALU, a leading zero count is performed to determine if normalization is required. The normalization shifter properly left-aligns the significand while the exponent is updated to reflect the normalization. Rounding and renormalization are also performed if required. To support INT operations, the ALU includes logic for bitwise operations as well as an adder, which is used for both FP and INT addition. The significand multiplier is used for INT multiplication and the alignment shifters can be used to perform INT shift operations. For INT operations, certain blocks such as the exponent logic and rounding stage are not used.

Since any exponent error will significantly change a computed FP value, our baseline protects the computed exponent using the low-cost exponent monitoring hardware proposed in [18]. When adapted for the FMA unit, this approach duplicates the "Compute Exponent" block in Figure 3. The computed (unadjusted) exponent is then compared to the result exponent plus or minus a small margin to account for possible normalization. With the exponent protected, we can focus on how to best protect the signif-

Algorithm 1 Traditional Selective Hardening

```
1: for v ← 1, num_vectors do
2:     Apply random input vector to circuit;
3:     for g ← 1, num_gates do
4:         Flip output of gate g;
5:         if output mantissa changed then
6:             gate_sensitized_count[g]++;
7:         end if
8:         Revert output of gate g;
9:     end for
10: end for
11: for g ← 1, num_gates do
12:     gate_score[g] = gate_sensitized_count[g]
13:         *drain_area[g]*timing_derating[g]
14:         /hardening_cost[g];
15: end for
16:
```

Algorithm 2 Precision-Aware Selective Hardening

```
1: for v ← 1, num_vectors do
2:     Apply random input vector to circuit;
3:     for g ← 1, num_gates do
4:         Flip output of gate g;
5:         if output mantissa changed then
6:             gate_score[g] += abs((correct_output
7:                 -faulty_output)/correct_output);
8:         end if
9:         Revert output of gate g;
10:    end for
11: end for
12: for g ← 1, num_gates do
13:     gate_score[g] = gate_score[g]
14:         *drain_area[g]*timing_derating[g]
15:         /hardening_cost[g];
16: end for
```

icand. This problem is worthy of consideration, since the significand logic consumes more resources than the exponent logic. One approach to reduce overhead is to use selective gate hardening. The next section discusses a baseline selective hardening algorithm that we enhance to make precision-aware.

3.1.1. Traditional Selective Hardening

To avoid high area and power overheads, selective gate hardening entails choosing a subset of gates in a circuit to protect. Individual gates can be hardened by creating a duplicate that shares inputs and outputs, or by resizing the gate by increasing the width of its component transistors. Both of these approaches increase the gate's driving strength, reducing the amplitude of transient pulses created by particle strikes [32]. If the gate is sized such that no deposited charge will create a pulse with an amplitude greater than Vdd/2, the gate can be considered immune to soft errors [36]. Other approaches examine the possibility of partially hardening gates for increased efficiency [31]. For simplicity, we fully harden all selected gates in this work, though our approach could be modified for partial hardening.

A number of different algorithms have been proposed for selecting gates to harden considering a variety of factors [23, 28, 29]. First and foremost, a gate is only vulnerable if there is a sensitized path from the gate to the circuit output. If there is no such path due to the circuit inputs applied, soft errors affecting the gate will be logically masked. Since analytical computation of logical masking probabilities is complicated by reconvergent fanouts [6], we implement a statistical approach similar to that in [36] for our baseline selective hardening algorithm. To statistically determine the probability of there being a logically sensitized path from any given gate to the output, we functionally simulate the circuit with a number of random input vectors. Because FP operands often have similar exponents, we fix the upper half of the input exponent bits and allow the other inputs bits to vary randomly. This also ensures that all inputs generated in this manner are valid FP numbers. We use random inputs here instead of application traces to make our solution more general.

As illustrated in Algorithm 1, we iterate through all gates in the circuit for each applied input vector. For each gate, we flip its output to the incorrect value, and note if this action changes a primary output of the circuit. If the circuit output is corrupted, we increment a counter associated with the gate. The gate output is then reset to the correct value before moving on to the next gate. Using this methodology, we can compute the percentage of input vectors for which there is sensitized path to the circuit output from each gate. This percentage corresponds to the probability that a gate will be vulnerable to a particle strike. We then assign each gate a score based on this probability such that the ranking will indicate the best gate to harden for the lowest cost. Since gates with larger drain areas are more likely to be affected by a particle strike, we multiply each gate's score by its drain area divided by the area cost to harden it [9]. We also apply a weight to reflect the effect of timing window masking, which occurs when the transient is not present at the circuit output during the clock edge, so it is not latched. Particle strikes that upset flip-flops are much less likely to be timing window masked at the next stage, since the erroneous value is held longer. The $timing\_derating$ weight accounts for this difference and favors protecting flip-flops, which we can harden by replacing with a robust flip-flop design [20]. Finally, we create a ranking of gates based on their final score, and choose the highest-scoring to protect first.

3.1.2. Precision-Aware Hardening Algorithm

The selective hardening methodology previously described treats all errors equally, regardless of error magnitude. Given a tight protection budget, however, it may be difficult to appreciably reduce a circuit's SER. It is possible, however, to significantly reduce the magnitude of errors that do occur by modifying the gate ranking algorithm. Algorithm 2 shows our proposed approach.

For our modified selective hardening algorithm, we take into account not only the presence of an error at the circuit output, but also the error magnitude relative to the correct FP value. As in the traditional hardening algorithm, all gates are flipped for each random input vector applied to the circuit, and a score is tracked for each gate. Instead of simply incrementing a counter when a gate's inversion would affect the output, each gate's score is now the summation of the relative output error created by flipping it for all input vectors. To determine the error magnitude, the correct and corrupted FP *values* output by the circuit are used. Using this methodology, a gate's score incorporates both the probability of a sensitized path to the output and the relative magnitude of the error that it can potentially create. We include only the error induced in the significand, regardless of whether or not the exponent was corrupted. This way, we account for the presence of the previously-discussed exponent monitor and avoid superfluous protection. As in the baseline approach, we apply weights to account for gate drain area, timing window masking, and the hardening cost.

3.1.3. Significand MSB Checker

As an alternative or complement to precision-aware gate hardening, we also explore the use of a low-cost checker circuit to detect large errors that corrupt the MSBs of the significand. Figure 4 shows how the checker circuit is employed along with the FMA unit. The circuit performs a redundant add and/or multiply operation for only the MSBs of the significand calculation. These bits are compared to the corresponding bits from the full circuit, thereby guaranteeing their correctness. This approach is potentially cheaper than selective gate hardening because the critical path is shorter for the checker circuit than the main circuit. The checker logic therefore does not require complex mechanisms such as parallel carry computation and booth encoding to meet the timing constraint. The looser timing constraint also allows the use of minimum sized gates, reducing the area and power overheads.

The checker circuit takes as input the MSBs of the mantissa of each operand. The implicit one is added internally as required for each check computation. For the multiplier portion, these truncated fractions are simply multiplied. The add computation requires somewhat higher complexity, since the mantissa datapath is not independent from the exponent part. To account for this, we tap some of the

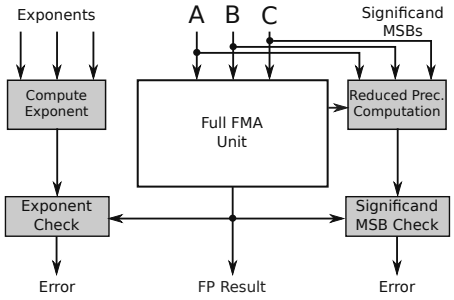


Figure 4. A significand checker circuit computes only the significand MSBs for comparison. For the addition part of the checker, we require the inversion and alignment signals from the FMA.

exponent-related signals in the full FMA unit. This information allows proper alignment of truncated inputs and inversion of the smaller operand for subtraction. As in a standard FP computation, the result is normalized before comparison with significand MSBs from the full unit's output.

A mismatch with the full result is interpreted as a significantly large precision error, and the current instruction is flagged for replay. Only one replay is allowed per instruction in case the mismatch was a false positive. Because our check computation is performed only with the MSBs of the input mantissas, the checker output is not guaranteed to match the MSBs of the full result, even in the absence of any soft errors. This potential for false positives exists because the checker circuit cannot compute the carry-in signals to the truncated computation that exist in the full adder and multiplier. The checker logic implicitly assumes these carry-in signals to be zero, thus computing an incorrect result if they actually are non-zero. Unlike prior work using narrow addends, our approach does not allow false negatives in the bits being checked, so our main concern is false positives [5]. To limit the number of false positives, we exploit the fact that carries are less likely to propagate across a large number of bits. In the checker, we compute more significand bits than are compared to the output of the full circuit. The lower-order MSBs that we do not compare serve as padding that may be incorrect due to the lack of carry-in data. To determine the optimal number of padding bits, we performed an experiment for the benchmarks studied in which we swept this parameter. For a configuration similar to the NVIDIA QuadroFX5800, we find the using 6 padding bits results in negligible performance impact due to false positives.

3.1.4. Integer Computation Extension

We observe that because the execution units are shared between INT and FP operations, the same soft error protection that benefits FP computations can also provide some

degree of protection for INT operations. Integer values, however, are stored right-aligned (unnormalized) such that the MSB of the stored value may be stored at any bit position, depending on the value magnitude. Due to this formatting, much of the benefit of our proposed protection techniques is lost when considering INT values, since the upper bits handled by the protected logic may be all zeros or all ones. To better exploit our precision-aware protection approach with INT values, we propose introducing the capability for the hardware to automatically store and process INT values in a FP-like format when possible. We will refer to this special INT format as INT-FP. This conversion is not always possible for larger magnitude INTs, since the FP mantissa can only hold up to 23 bits in the 32-bit representation. INT values with magnitudes too large to convert still benefit from precision-aware protection, however. In these cases, the INT value is large enough that its MSBs will be processed by the protected logic.

As shown in Figure 5, our INT-FP format uses a representation similar to standard FP values. The INT-FP format differs from standard FP format in two ways. First, we store the mantissa in two's complement form. Doing so greatly simplifies bitwise operations, since otherwise multiple two's complement operations may be needed to convert from and back to a magnitude-only representation. As in standard FP format, the MSB of the significand is implicit and is not stored. In our case, however, the implicit bit can be a one or a zero, depending on the sign of the value being stored. This implicit bit will always be the inverse of the sign bit. Second, some LSBs of a stored INT-FP value may not contain valid data. Unlike FP numbers, which are fractional, INT-FP numbers cannot be fractional, and thus all bits to the right of the radix point must be zero. Any soft errors that corrupt these bits will be masked, since the hardware can simply discard them. We store INT values in INT-FP format only in the execution core, and rely on extra state bits to indicate the presence of this special format in the register file.

When an INT value is read from memory during a load instruction, we do not immediately convert it to INT-FP format. We convert only the results of computation operations so that we can use the execution unit's normalization shifter in the conversion. When addresses or INT data are sent to the memory unit, the memory unit converts them back to standard INT form if they were flagged as INT-FP. This conversion is accomplished with a simple right shift. The execution unit in Figure 3 shows the additional logic required for handling the INT-FP case. After an INT or INT-FP computation is performed, an INT-FP check block determines if it is possible to store the result in INT-FP format. Based on the projected exponent calculated by the "Compute Exponent" block and the number of sign bits in the significand result, we can determine if the significand will fit within

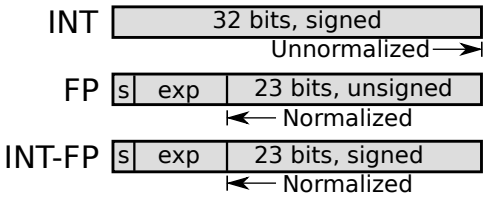


Figure 5. Comparison of standard 32-bit INT format, FP format and our INT-FP format. FP and INT-FP both have an 8-bit exponent and a sign bit.

the 23 bits normally allocated for the FP mantissa. Since the INT-FP significand is signed, the leading zero counter is augmented with the ability to count leading ones in the case of a negative value.

Our baseline assumes that four execution units are grouped in a SIMT cluster. A simple check is performed to determine if all lanes in the cluster are asserting *lane_intfp* and can output INT-FP values. If so, *cluster_intfp* is asserted and normalization is performed as in FP computations. If any lane cannot output an INT-FP value, all lanes in the cluster will output standard right-aligned INT values. This check is performed because we add one INT-FP state bit for every four registers in the register file, so each set of four registers must stored in the same format. This optimization is optional, however, since registers could be treated separately by added more INT-FP state bits.

A few other changes are required in the execution unit to ensure efficient processing of INT-FP values. Like addition, alignment must be performed before bitwise operations. Since the significand is signed, alignment requires a sign extension. Finally, shift operations can be optimized such that it is only necessary to increment or decrement the INT-FP exponent.

3.2. Register File Protection

Modern GPU architectures employ ECC to protect the register file against bit flips [22]. For our precision-aware protection approach, we instead replace the most important bits with robust SRAM cells. As previously shown in Figure 2, register banks are shared by a cluster of four execution units. Each bank is comprised of 128-bit entries that can store four 32-bit registers or two 64-bit values.

In our design, we harden the exponent and sign bits that would correspond to single-precision FP values stored in each 32-bit subentry, as shown in Figure 6. In addition, a number of mantissa MSBs are hardened, depending on the protection budget. When double-precision execution units are present, the resulting values are stored across two 32-bit subentries. To best exploit register file protection, we add a state bit per entry to indicate if the entry contains two 64-bit values. When this bit is set, it indicates that the stored 64-bit values are interleaved across two 32-bit subentries. This

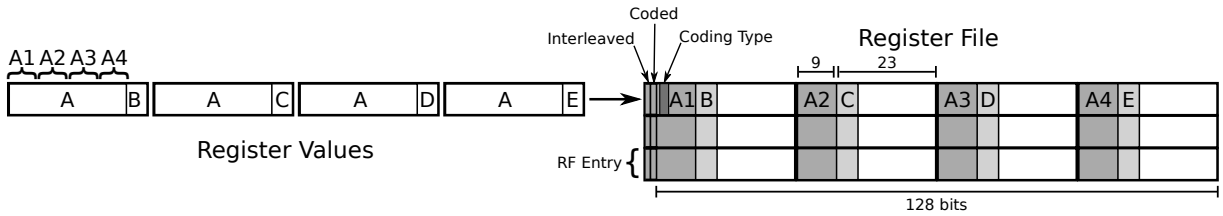


Figure 6. Register encoding illustration. For each set of four registers in standard INT or FP format, the MSBs of each register are compared before writing to the register file. If all MSBs match, the MSB value is distributed across the hardened cells in each 128-bit entry, as shown. The unique LSBs for each register can then be stored in hardened cells (indicated by the shading).

allows us to store the most important bits in the hardened cells for both double and single-precision values.

Applying selective cell hardening in this manner allows us to protect against large magnitude errors in FP values as well as INT values stored in our INT-FP format. To accommodate the latter, we include an additional state bit per 128-bit entry to indicate if the registers are stored in INT-FP format. We will refer to this bit as the Coded bit. This gives us a total of two extra metadata bits per 128-bit entry, as shown in Figure 6. As previously mentioned, using a single Coded bit per register file entry requires that all grouped values are stored in the same format.

It is likely that values such as addresses or other large INTs cannot be stored in INT-FP format. To improve error protection in these cases, we note that across threads in the same warp, computed values and particularly addresses will often share the same MSBs. When the MSBs of all registers in a register file entry match, we can encode the entry such that all information is stored in protected cells. An illustration of this protected format is shown in Figure 6. As shown, when four registers have matching MSBs, the shared data is distributed across all four subentries, allowing all useful information to be stored in hardened cells. We can cleverly use the same previously-mentioned Coded bit to indicate when a register file entry is coded in this distributed MSB format. We distinguish between this format and the INT-FP format because the exponent of INT-FP values is naturally restricted to a certain range, since it is impossible for these values to be less than one. Thus we designate a certain out-of-bounds value for the exponent of the first register in the entry to indicate that the MSBs have been distributed. In fact, it is sufficient to check the upper nibble of this exponent. We store INT and FP values in the distributed MSB format when possible for additional protection.

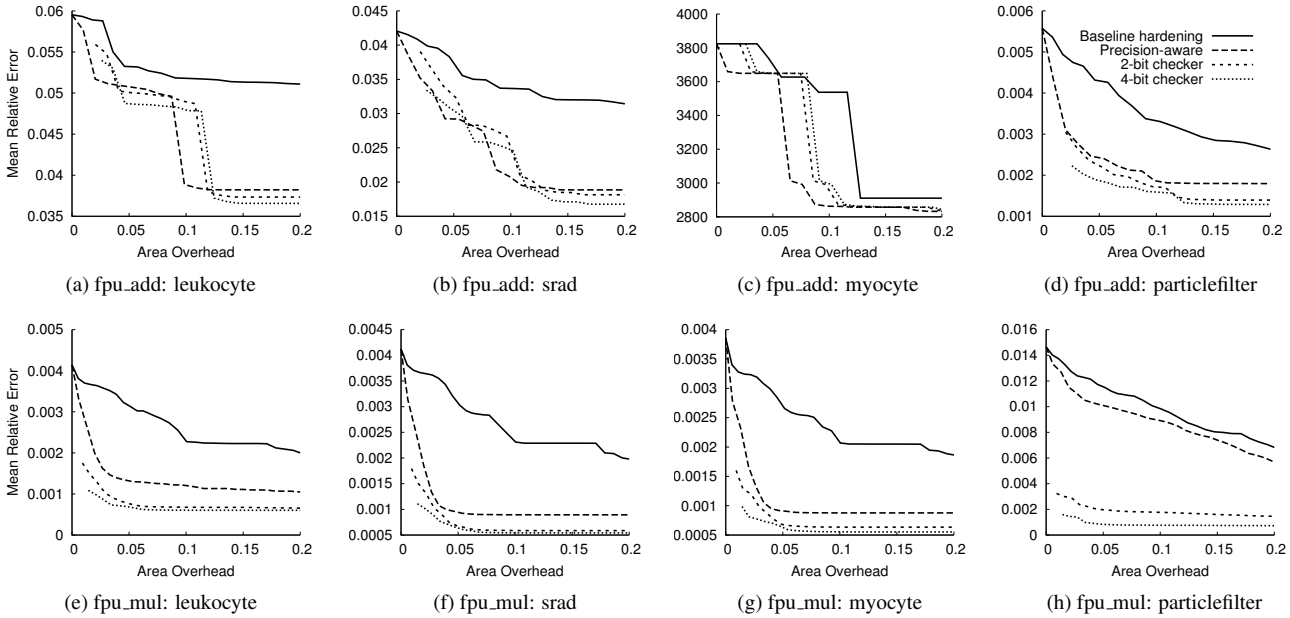
4. Experimental Results

4.1. Methodology

To evaluate our proposed techniques for execution logic, we synthesized OpenSPARC FP adder and multiplier units in 65nm technology [25]. Although the GPU execution unit uses a combined multiplier and adder, our evaluation uses the OpenSPARC units since they are freely-available and demonstrate the effectiveness of our approach. Thus, we present separate results for protecting the adder and multiplier. Because these circuits are relatively large, circuit-level simulation is not feasible. Therefore, we use a timing-enabled gate-level simulator to perform statistical fault injection. This approach injects soft errors into a circuit randomly in time and space. A time offset is chosen randomly within the clock cycle, and a target gate is randomly selected using an area model accounting for the vulnerable drain area of all gates. The transient pulse duration is chosen using a probability distribution generated from SPICE simulations with varying levels of deposited charge, as in [33]. For each fault simulated, we apply different input operands to the FP unit. These operands are extracted from FP instructions in GPGPU benchmarks using GPGPU-Sim [2]. For our evaluation, we use the Rodinia benchmark suite, which is comprised of a number of highly-parallel FP and INT applications [3].

To compute the area overhead of each protection technique, we synthesized each baseline FP circuit as well as the different width checker circuits in 65nm. Based on the analysis in [32], we conservatively upsize the gates selected for hardening by a factor of three. The impact of upsizing is translated into area overhead using standard cell library data. For latch hardening, we use the resilient latch design and overheads presented in [20]. For our baseline and all of the protected scenarios, we assume the presence of the exponent checking hardware described in [18].

To generate gate rankings for the baseline and precision-aware selective hardening experiments, we use the algorithms from Section 3 along with 10000 randomized inputs. Once the gate rankings have been generated for both cases, we then perform statistical fault injection while using inputs from the Rodinia benchmarks. Note that the gate rankings used for selective hardening remain the same regardless of which benchmark is run. To quantify the benefits of the precision-aware hardening, we compute the relative error induced in the result by each fault, and report the mean rel-



(a) fpu_add: leukocyte (b) fpu_add: srad (c) fpu_add: myocyte (d) fpu_add: particlefilter

(e) fpu_mul: leukocyte (f) fpu_mul: srad (g) fpu_mul: myocyte (h) fpu_mul: particlefilter

Figure 7. Results for corner-case benchmarks showing error reduction in the output of the FP adder and multiplier in the presence of soft errors. Error is shown for traditional hardening, precision-aware hardening, and precision-aware hardening combined with a significand checker. Precision-aware protection provides significant benefit over traditional selective hardening in most cases.

ative error. Faults that never create an error in the output because they strike a hardened gate, are logically masked, or are timing-window masked (not latched) result in no error. Similarly, faults that are detected by the exponent monitoring hardware or by our MSB checker circuit (when applicable) are considered to have zero error, since the instruction will be replayed to obtain the correct result.

4.2. Results

Figure 7 shows the effect of traditional and precision-aware hardening on the error in the FP results for selected corner cases. With increasing area budget, we harden additional gates in the order of our computed ranking. We also show the impact of our MSB checker circuit when combined with precision-aware selective hardening. Note that the area for the MSB checker cases is not fixed because we combine the (fixed-overhead) checker with a variable amount of gate hardening. For most benchmarks, we see results similar to those shown for the *leukocyte* and *srad* in the figure. In these cases, precision-aware selective hardening provides significant benefits in terms of error magnitude when compared to the traditional selective hardening approach. Some interesting cases exist, however, depending on the computations performed by a benchmark. In the case of the *myocite* benchmark, for instance, a large portion of the FP add instructions subtract numbers that are very close in magnitude, requiring large normalizations. Unmitigated faults in the mantissa can corrupt the normalization step and therefore the exponent. Because the external exponent monitor relies on data from the mantissa in this special case, such exponent errors can go undetected. These undetected and unmasked large errors create the erratic trend shown in Figure 7c. This is an artifact of the exponent checker, which could potentially be improved.

On the other hand, benchmarks such as *particlefilter* are unique in that for many operations the mantissa bits are mostly zeros. This has particular implications for our hardening scheme in the multiplier unit due to the increased number of carries typically present in multiplication. If many partial products are zero, a soft error in the multiplier array is less likely to affect the MSBs of the result through creation of erroneous carries. This changes the error-inducing potential of each gate such that it no longer matches well with the computed rankings. In all cases, the mean relative error for the baseline hardening scheme monotonically decreases since more hardening can only reduce the number of errors, and faults that are masked due to hardening are considered to create an error of magnitude zero.

Figures 8 and 9 summarize the mean relative error reduction in FP add and multiply results for each benchmark with a fixed area overhead limit of 5%. We choose this overhead so we can include the cases with the checker in the comparison. The anomalous cases in which precision-aware hardening does not significantly improve on traditional hardening

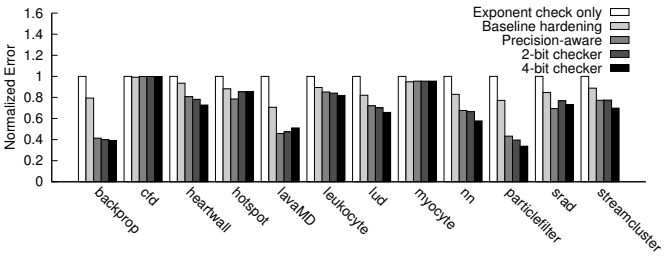


Figure 8. Adder error with 5% area overhead for FP operations with selective hardening schemes and checker circuits.

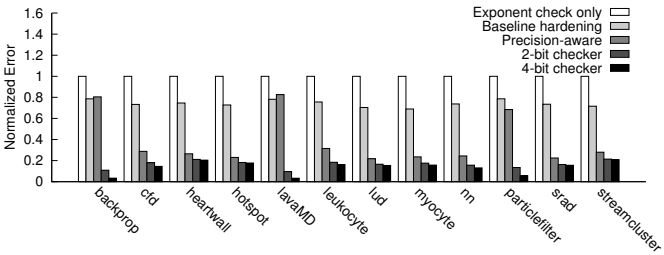


Figure 9. Multiplier error with 5% area overhead for FP operations with selective hardening schemes and checker circuits.

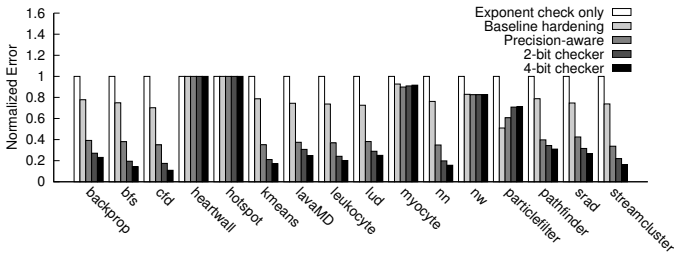


Figure 10. Adder error with 5% area overhead for INT operations with selective hardening schemes and checker circuits.

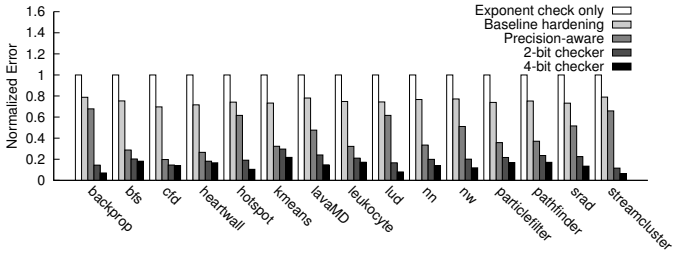


Figure 11. Multiplier error with 5% area overhead for INT operations with selective hardening schemes and checker circuits.

correspond to the scenarios previously discussed. When the 4-bit mantissa checker is combined with precision-aware hardening, we see an average error reduction of 87% for the multiplier and 31% for the adder with 5% area overhead. Figures 10 and 11 show similar results for INT computations. In this case, we perform the computation with INT-FP input values if possible. Based on our results, we observe that protection of the multiplier is most effective since it does not require significand alignment. The multiplier is also the larger arithmetic unit and therefore more vulnerable.

We also estimate power overheads for protecting the execution units. For the adder, multiplier, and checker circuits, we use the power reported by the Synopsys® tools assuming a 50% switching probability. To quantify the power overhead of hardening, we simulate the adder and multiplier circuits using the approach from [13] and triple the capacitance of hardened nodes. The calculated power overheads are shown in Table 1. Power overheads are shown for the case with 5% area overhead, as in Figures 8 and 9. As before, when a checker circuit is present, this area overhead is split between the checker circuit and precision-aware hardening. As expected, power overhead is higher than area overhead, since cell size does not scale linearly. We also note that our hardening technique is more power efficient than a traditional approach with an equivalent area overhead. The strong precision bias added to the precision-aware gate ranking allows the hardening of smaller gates and gates that switch less frequently as long as they have a significant potential contribution to error magnitude.

When protecting the register file with precision-aware hardening, it is crucial to harden the exponent and sign bits, so we use this case as the default in our error magnitude experiments. We then augment this baseline by hardening different numbers of mantissa MSBs. As in our execution logic evaluation, we use register values captured from benchmarks in the Rodinia benchmark suite. Instead of statistical fault injection, which is helpful when dealing with a large number of error scenarios, we compute the mean error incurred from flipping all vulnerable bits in the register file. For FP values, we evaluate hardening different numbers of mantissa MSBs with and without the MSB distribution coding. For INT register values, we separately evaluate mantissa MSB hardening only, INT-FP format, MSB distribution encoding and the combination of both. We use GPGPU-Sim to capture registers from the same SIMT cluster so we can determine if it is possible to use INT-FP format or MSB distribution in a 128-bit register file entry. In our experiments, we also only allow INT-FP format for INT registers that store the result of a computation (not a load) since we rely on the execution unit to convert to INT-FP format.

Figure 12 shows the mean relative error in FP registers when hardening 4 or 8 mantissa MSBs with and without MSB distribution encoding. In the FP case, we note only a slight improvement from the MSB distribution encoding.

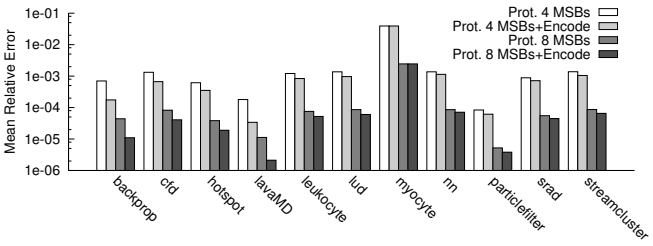


Figure 12. Mean relative error in FP registers with different types of protection.

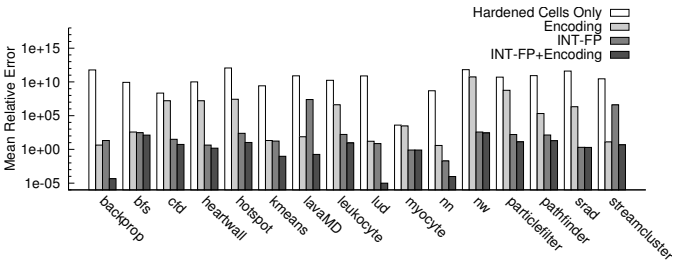


Figure 13. Mean relative error in INT registers. All cases harden the most significant 13 bits for 32-bit registers.

For the INT register results shown in Figure 13 the upper 13 bits of each 32-bit register are hardened. This corresponds to hardening four bits of the mantissa in the FP case, since the 9 sign and exponent bits are always hardened. For INT registers, we observe a significant benefit from combining INT-FP and the distributed MSB encoding. In addition to providing protection for large INT values such as addresses, the distributed MSB format is able to protect registers that are loaded from memory, and thus cannot be saved in INT-FP format.

When evaluating the overheads of register file protection, we compare against an ECC-protected register file, as present in modern GPUs. Since each register file entry is 128 bits, we generated a (137,128) minimum odd-weight-column SECDED code, which requires 9 extra parity bits to protect each 128-bit entry [10]. We use Synopsys® to synthesize the required encode and decode logic, and model the SRAM using HP CACTI [34]. For this experiment we model the overhead relative to a baseline 4kB bank with one read and one write port. To model the overhead of our protection scheme, we modified CACTI to account for the overhead of robust SRAM cells. We use the robust cell design presented in [11] for which power and area overheads are calculated in [35]. We also synthesize the peripheral logic needed to perform the MSB compare, encoding, and decoding. Table 2 shows the area and power overheads relative to an unprotected bank. For our approach, we experiment with hardening between two and eight mantissa MSBs in addition to the exponent and sign bits. As the results indicate, the area overhead of our result is comparable to ECC, but our approach incurs less power overhead. This is because our protected bank requires fewer bitlines and avoids the high glitching power of the ECC parity logic.

Table 1. Execution unit power overheads

Circuit	Trad.	Prec.-Aware	2-bit ch.	4-bit ch.
FP Add	12.5%	9.6%	7.3%	7.9%
FP Mul	23.7%	6.8%	6.9%	7.3%

Table 2. Register file overheads

Protection Type	Area Overhead	Power Overhead
ECC	15.8%	32.9%
Prot. 2 MSB	14.4%	9.26%
Prot. 4 MSB	15.8%	9.88%
Prot. 6 MSB	17.3%	10.7%
Prot. 8 MSB	18.7%	11.3%

5. Conclusions

This paper proposes a novel approach for protecting the GPU execution logic and register file from particle strike-induced soft errors. A selective hardening scheme is introduced that provides cost-proportional protection by favoring gates that contribute most to error magnitude, rather than prior approaches that are precision-agnostic. Also, we show that a low-cost signficand MSB checker circuit can provide additional coverage, and that coverage can be extended to integer values by treating them like floating point values. Finally, we demonstrate a similar approach to protect the register file using robust SRAM cells and intelligent data encoding. Results indicate that the multiplier checker is particularly cost-effective, providing an error reduction of 87% when combined with precision-aware selective hardening.

Acknowledgement

This work was supported in part by generous grants from AMD, NSF (CCF-1116450, CCF-1318298, CCF-0953603, CNS-1217102, and CCF-1016262), MSIP GFP (CISS-2011-00311816), DARPA (HR0011-12-2-0019), donations from Qualcomm and Oracle research, and the IBM Ph.D. Fellowship program. Nam Sung Kim has a financial interest in AMD.

References

[1] Advanced Micro Devices. Heterogeneous computing – OpenCL and the ATI radeon ("Evergreen") architecture, 2008.

[2] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Proc. Int. Symp. Performance Analysis of Systems and Software*, pages 163–174, April 2009.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Proc. Int. Symp. Workload Characterization*, pages 44–54, Oct. 2009.

[4] M. R. Choudhury and K. Mohanram. Approximate logic circuits for low overhead, non-intrusive concurrent error detection. In *Proc. Design, Automation and Test in Europe*, pages 903–908, 2008.

[5] P. Eibl, A. Cook, and D. Sorin. Reduced precision checking for a floating point adder. In *Defect and Fault Tolerance in VLSI Systems*, pages 145–152, Oct. 2009.

[6] D. Franco, M. Vasconcelos, L. Naviner, and J. Naviner. Signal probability for reliability evaluation of logic circuits. *Microelectronics Reliability*, 48(8):1586–1591, 2008.

[7] M. Gebhart, S. W. Keckler, and W. J. Dally. A compile-time managed multi-level register file hierarchy. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 465–476, New York, NY, USA, 2011. ACM.

[8] S. Gilani, N. S. Kim, and M. Schulte. Power-efficient computing for compute-intensive GPGPU applications. In *Proc. 19th International Symposium on High Performance Computer Architecture (HPCA 2013)*, pages 330–341, 2013.

[9] K. J. Hass and J. W. Ambles. Single event transients in deep submicron CMOS. In *Proc. 42nd Midwest Symp. on Circuits and Systems*, pages 122–125, 1999.

[10] M. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development*, 14(4):395–401, 1970.

[11] S. Jahinuzzaman, D. Rennie, and M. Sachdev. A soft error tolerant 10t sram bit-cell with differential read capability. *IEEE Trans. Nuclear Science*, 56(6):3768–3773, 2009.

[12] H. Jeon and M. Annavaram. Warped-dmr: Light-weight error detection for gpgpu. In *Microarchitecture (MICRO), 2012 45th Annual IEEE/ACM International Symposium on*, pages 37–47, 2012.

[13] N. S. Kim, T. Kgil, V. Bertacco, T. Austin, and T. Mudge. Microarchitectural power modeling techniques for deep submicron microprocessors. In *Proc. Int. Symp. Low Power Electronics and Design*, pages 212–217, 2004.

[14] K. Lee, A. Shrivastava, I. Issenin, N. Dutt, and N. Venkatasubramanian. Mitigating soft error failures for multimedia applications by selective data protection. In *Proc. CASES '06*, CASES '06, pages 411–420, New York, NY, USA, 2006.

[15] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. Nvidia tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[16] C. Lisboa, M. Erigson, and L. Carro. System level approaches for mitigation of long duration transient faults in future technologies. In *Test Symposium, 2007. ETS '07. 12th IEEE European*, pages 165–172, May 2007.

[17] J.-C. Lo. Reliable floating-point arithmetic algorithms for error-coded operands. *IEEE Trans. on Computers*, 43(4):400–412, 1994.

[18] M. Maniatakos, Y. Makris, P. Kudva, and B. Fleischer. Exponent monitoring for low-cost concurrent error detection in fpu control logic. In *VLSI Test Symposium (VTS), 2011 IEEE 29th*, pages 235–240, May 2011.

[19] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. 36th Annual IEEE/ACM Int. Symp. on Microarchitecture*, pages 29–40, December 2003.

[20] H. Nan and K. Choi. High performance, low cost, and robust soft error tolerant latch designs for nanoscale cmos technology. *IEEE Trans. on Circuits and Systems*, 59(7):1445–1457, 2012.

[21] R. Nathan and D. J. Sorin. Argus-g: A low-cost error detection scheme for gpgpus. In *Workshop on Resilient Architectures (WRA)*, 2010.

[22] NVIDIA. NVIDIA's next generation CUDA compute architecture: Fermi, 2009.

[23] S. Pagliarini, G. dos Santos, L. de B. Naviner, and J.-F. Naviner. Exploring the feasibility of selective hardening for combinational logic. *Microelectronics Reliability*, 52(9-10):1843–1847, 2012.

[24] D. J. Palframan, N. S. Kim, and M. H. Lipasti. Time redundant parity for low-cost transient error detection. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 1–6, March 2011.

[25] I. Parulkar, A. Wood, J. C. Hoe, B. Falsafi, S. V. Adve, J. Torrellas, and S. Mitra. OpenSPARC: An open platform for hardware reliability experimentation. In *Workshop on Silicon Errors in Logic-System Effects*, 2008.

[26] V. Podlozhnyuk. Black-scholes option pricing. *Part of CUDA SDK documentation*, 2007.

[27] I. Polian, B. Becker, M. Nakasato, S. Ohtake, and H. Fujiwara. Low-cost hardening of image processing applications against soft errors. In *Proc. Int. Symp. on Defect and Fault-Tolerance in VLSI Systems*, pages 274–279, 2006.

[28] I. Polian and J. P. Hayes. Selective hardening: Toward cost-effective error tolerance. *IEEE Des. Test*, 28(3):54–63, May 2011.

[29] R. R. Rao, D. Blaauw, and D. Sylvester. Soft error reduction in combinational logic using gate resizing and flipflop selection. In *Proc. Int. Conf. on Computer-aided design*, pages 502–509, 2006.

[30] B. D. Sierawski, B. L. Bhuva, and L. W. Massengill. Reducing soft error rate in logic circuits through approximate logic functions. *Nuclear Science, IEEE Transactions on*, 53(6):3417 –3421, Dec. 2006.

[31] W. Sootkaneung and K. Saluja. Soft error reduction through gate input dependent weighted sizing in combinational circuits. In *Int. Symp. Quality Electronic Design (ISQED)*, pages 1–8, March 2011.

[32] V. Srinivasan, A. L. Sternberg, A. R. Duncan, W. H. Robinson, B. L. Bhuva, and L. W. Massengill. Single-event mitigation in combinational logic using targeted data path hardening. *IEEE Trans. Nuclear Science*, 52(6):2516–2523, Dec. 2005.

[33] F. Wang and V. Agrawal. Soft error rate determination for nanometer cmos vlsi logic. In *Southeastern Symp. System Theory*, pages 324–328, March 2008.

[34] S. Wilton and N. Jouppi. CACTI: an enhanced cache access and cycle time model. 31(5):677–688, May 1996.

[35] G. Zhang, J. Shao, F. Liang, and D. Bao. A novel single event upset hardened SRAM cell. *IEICE Electronics Express*, 9(3):140–145, 2012.

[36] Q. Zhou and K. Mohanram. Gate sizing to radiation harden combinational logic. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 25(1):155–166, Nov. 2006.