Temporal Memoization for Energy-Efficient Timing Error Recovery in GPGPU Architectures [*]

Abbas Rahimi

Department of Computer Science and Enginnering
University of California, San Diego
La Jolla, CA 92093, USA

Luca Benini

Department of Electrical, Electronic and Information Engineering
University of Bologna
40136 Bologna, Italy

Rajesh K. Gupta

Department of Computer Science and Enginnering
University of California, San Diego
La Jolla, CA 92093, USA

June 12, 2014

[*]UC San Diego CSE Department Technical Report CS2014-1006

Abstract

Manufacturing and environmental variability lead to timing errors in computing systems that are typically corrected by error detection and correction mechanisms at the circuit level. The cost and speed of recovery can be improved by *memoization*-based optimization methods that exploit spatial or temporal parallelisms in suitable computing fabrics such as general-purpose graphics processing units (GPGPUs). We propose here a *temporal memoization* technique for use in floating-point units (FPUs) in GPGPUs that uses value locality inside data-parallel programs. The technique recalls (*memorizes*) the context of error-free execution of an instruction on a FPU. Therefore, it avoids redundant execution and saves energy for FPU. To enable scalable and independent recovery, a single-cycle lookup table (LUT) is tightly coupled to every FPU to maintain contexts of recent error-free executions. The LUT reuses these memorized contexts to exactly, or approximately, correct errant FP instructions based on application needs. In real-world applications, the temporal memoization technique achieves an average energy saving of 13%–25% for a wide range of timing error rates (0%–4%) and outperforms recent advances in resilient architectures. This technique also enhances robustness in the voltage overscaling regime and achieves relative average energy saving of 44% with 11% voltage overscaling.

Contents

1 Introduction

The scaling of physical dimensions in semiconductor circuits opens the way to an astonishing over 7 billion transistors on a 28nm process which gives a grand total of 2,880 CUDA cores in recent GPGPU chips [1]. It is also leading to ever-increasing parametric variations across process, voltage and temperature (PVT) [2]. As designers build circuits operating near-threshold [3] in order to save power, and use voltage overscaling [4] to reach performance targets, the effect of PVT variations are exacerbated.

The most common effect of variation is violation of timing specifications that cause circuit-level timing errors. IC designers commonly use conservative guardbands for the operating frequency or voltage to ensure error-free operation for the worst-case variations. These guardbands have been steadily increasing, thus leaving untapped performance and other costs of overdesign [5]. An alternative to overdesign is to make a design resilient to errors and variations. In this paper, we specifically focus on 'Design for Resiliency' (DFR) against timing errors.

Low-voltage DFR applies to both logic and memory blocks. For memory, 8T SRAM arrays utilize a tunable replica bits therefore enables reduction of the minimum operating voltage [7]. Similarly, in non-volatile memory area, resistive RAM (ReRAM/memristor) is a promising candidate at low power supply voltage [8]. For logic, error-detection sequential (EDS) [6] circuit sensors have been employed to reduce guardbanding in the designed circuits. A common strategy is to detect variability-induced delays by sampling and comparing signals near the clock edge to detect timing errors. The timing errors are corrected by replaying the errant operation with a larger guardband through various adaptation techniques. For instance, a resilient 45nm integer-scalar core [9] places EDS circuits in the critical paths of the pipeline stages. Once a timing error is detected during instruction execution, the core prevents the errant instruction from corrupting the architectural state and an error control unit (ECU) initially flushes the pipeline to resolve any complex bypass register issues. To ensure scalable error recovery, the ECU supports two separate techniques: instruction replay at half frequency, and multiple-issue instruction replay at the same frequency. These techniques impose energy overhead and latency penalty of up to 28 extra recovery cycles per error for the resilient 7-stage integer pipeline [9].

As energy becomes the dominant design metric, aggressive voltage scaling [4] and near-threshold operations [3] increase the rate of timing errors and correspondingly the costs (in energy, performance) of these recovery mechanisms. This cost is exacerbated in FP single-instruction multiple-data (SIMD) pipelined architectures where the pipeline dimensions are expanded both vertically (with wider parallel lanes) and horizontally (with deeper stages). The horizontally expanded deeper pipelines induce higher pipeline latency and higher cost of recovery through flushing and replaying the errant instruction. The FP pipelines consume higher energy-per-instruction than their integer counterparts and typically have high latency for instance over 100 cycles [10] to execute on a GPGPU. Effectively, these energy-hungry high-latency pipelines are prone to inefficiencies under timing errors. Similarly, in vertically expanded pipelines, there is a significant performance drop in a 10-lane SIMD architecture as single-stage-error probabilities increase [11]. In the lock-step execution, any error within any of the lanes will cause a global stall and force recovery of the entire SIMD pipeline.

Thus, in FP SIMD pipelines the error rate is multiplied by the wider width while the number of recovery cycles per error increases at least linearly with the pipeline length. This makes the cost of recovery per single error quadratically more expensive relative to scalar functional units. At the same time, parallel execution in the GPGPU architectures – described in Section 3– provides an important ability to reuse computation and reduce the cost of recovery from timing errors. This paper, exploits this opportunity to make three main contributions:

1. We have earlier proposed a *temporal memoization* technique for energy-efficient execution in GPGPUs [21]. We observe that the entropy of data-level parallelism is low due to high locality of values. The temporal memoization of recent error-free executions exploits this inherent value locality. We show that the memorized information can be used to reduce the energy of a FPU by returning the *pre-stored* result that avoids redundant execution, and further reduces the conventional recovery costs in the presence of a timing error. the memorized information can be utilized exactly or approximately depending upon the application needs.

2. We closely integrate a lightweight single-cycle LUT to the FPU to support instruction-level memoization. The design enables a scalable and independent recovery of individual FPUs. Section 4 covers these details.

3. We demonstrate the effectiveness of our technique on the Evergreen GPGPU architecture for error-tolerant image processing applications as well as error-intolerant general-purpose applications selected from AMD APP SDK v2.5 [23]. Multi2Sim [25], a cycle-accurate CPU-GPU simulation framework, is modified to collect the statistics for computing the temporal value locality out of 27 single precision floating-point instructions. The modified simulator code is available for downloading at [28]. Our experimental results in Section 5 show an average energy saving of 13% in case of an error-free execution environment (0% timing error); it further reaches to an average energy saving of 25% in the presence of 4% timing error rate thanks for improving recovery cost on the errant instructions. This technique also enhances robustness in the voltage overscaling scenario, up to an average energy saving of 44%.

2 Related Work

Sodani and Sohi [12] introduced the concept of instruction reuse from the observation that many instructions can be skipped if another instance has already been executed using the same input values. The instruction reuse enables memorization of the outcome of an instruction in hardware tables, therefore a processor can reuse it temporally if the processor performs the same instruction with the same input values within a limited period of time before the entry is overwritten. This temporal memorization technique is fundamentally limited by the latency and the low hit rate of the tables. To improve the hit rates, recent reuse techniques [13, 14] seek to improve association of the entries of the table with similar inputs to the same output. These tolerant techniques rely upon the tolerance in the output precision of multimedia algorithms to achieve high reuse rates, and work at the granularity of the FP instruction [14], or a region of FP instructions [13]. A load value prediction technique also exploits the value locality to predict the register file values being loaded from memory based on previously-seen values [15]. These techniques have been devised for single-core architectures without exploring their potential in combating timing errors in data-level parallel architectures.

Beyond instruction-level reuse, various techniques have been proposed to mitigate the variation-induced timing errors, including adaptive management of guardbanding through 'predict-and-prevent' mechanisms [16, 17, 22, 18, 19], and ‘detect-then-correct’ mechanisms [6, 7, 9, 11]. A brief review of the main concepts and their embodiments follows.

Predict-and-prevent techniques try to avoid timing errors while reducing guardbands, for instance for individual instructions [16]. The instruction program counter of an out-of-order pipeline is used for an early prediction of an upcoming timing violation by searching in a large predictor table [22]. At higher levels, a *procedure hopping* technique is proposed to avoid voltage droops [17]. In the context of the GPGPU architecture, hierarchically focused guardbanding [18] has been proposed earlier at two levels: fine-grained instruction-level and coarse-grained kernel-level. Rahimi et al. [19] propose a compiler technique that periodically regenerates healthy codes that reduces the aging-induced performance degradation of the GPGPUs. The predictive techniques cannot eliminate the entire guardbanding to work efficiently at the edge of failure specially so with frequent timing errors in the voltage overscaling and near-threshold regimes.

Detect-then-correct technique for SIMD architectures decouples the lanes through private queues that prevent error events in any single lane from stalling all other lanes [11]. This enables each lane to recover from errors independently. The decoupling queues cause slip between lanes which requires additional architectural mechanisms to ensure correct execution. Further, the decouple queue relies on the recovery based on the global clock-gating which involves stalling the entire lane. This causes one cycle recovery penalty over a two-stage execution unit [11]. However, propagating a global stall signal over a deep GPGPU pipeline [10] is expensive. A *spatial memoization* [20] technique also broadcasts output result of an error-free instruction across all *error-prone* lanes, tightens its scalability. Hence, the cost of scalable recovery (e.g., [9]) per single timing error on these architectures is high limiting their utility to low error rate circumstances. Our present work enhances the scope of ‘detect-then-correct’ approaches in a GPGPU context thanks to an ultra-low cost recovery through memoization, thus offering both scalability and low-cost self-resiliency in the face of high timing error rates.

3 GPGPU Architecture

We focus on the Evergreen family of AMD GPGPUs (a.k.a. Radeon HD 5000 series), that targets general-purpose data-intensive applications. The Radeon HD 5870 GPGPU consists of 20 compute units, a global front-end

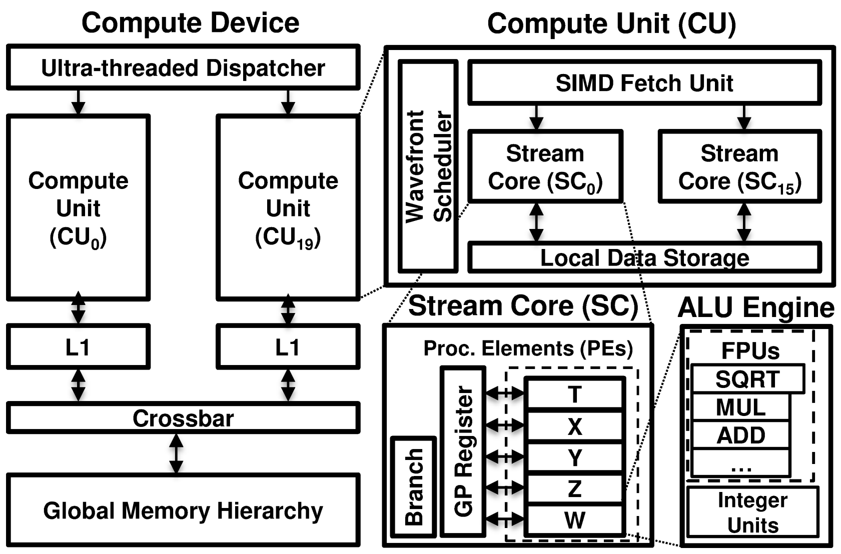


Figure 1: Block diagram of the Radeon HD 5870 GPGPU.

ultra-thread dispatcher, and a crossbar to connect the memory hierarchy. Each compute unit contains a set of 16 Stream Cores (SCs), i.e., 16 parallel lanes. Within a compute unit, a shared instruction fetch unit provides the same machine instruction for all SCs to execute in a SIMD fashion. Each SC contains five Processing Elements (PEs) – labeled X, Y, Z, W, and T – forming an ALU engine to execute Evergreen machine instructions in a vector-like fashion. Finally, the ALU engine has a pool of pipelined integer and FP units. The block diagram of the architecture is shown in Figure 1.

The device kernel is written in OpenCL which runs on a GPU device. An instance of the OpenCL kernel is called a work-item. Each SC is devoted to the execution of one work-item. In the Radeon HD 5870, a *wavefront* is defined as the total number of 64 work-items virtually executing at the same time on a compute unit. To manage 64 work-items in a wavefront on 16 SCs of the compute unit, a wavefront is split into *subwavefronts* at the execute stage, where each subwavefront contains as many work-items as available SCs. In other words, SCs execute the instructions from the wavefront mapped to the SIMD unit in a 4-slot time-multiplexed manner using the integer units and FPUs. The time-multiplexing at the cycle granularity relies on the functional units to be fully pipelined.

Evergreen assembly code uses a clause-based format classified in three categories: ALU clause, TEX clause, and control-flow instructions. The control-flow instructions triggering ALU clauses will be placed in the input queue at the ALU engine. There is only one wavefront associated with the ALU engine. After fetch and decode stages, the source operands for each instruction are read that can come from the register file or local memory. For higher throughput, buffers are attached to SCs to read the registers ahead of time. The core stage of a GPGPU is the execute stage, where arithmetic instructions are carried out in each SC. When the source operands for all work-items in the wavefront are ready, the execution stage starts to issue the operations into the SCs. Finally, the result of the computation is written back to the destination operands.

4 Temporal Memoization

We briefly describe how value locality can increase the resiliency of the GPGPUs followed by a description of the proposed temporal memorization technique. We divide all applications into two classes: error-tolerant image processing applications and error-intolerant general-purpose applications selected from AMD APP SDK v2.5 [23]. For error-tolerant applications, we have examined two image processing filters: Sobel and Gaussian. The error-tolerant applications exhibit enhanced error resilience at the application-level when multiple valid output values are permitted, in effect, creating a relation from input values to (multiple) output values. Instead of a single number, the output value is associated with a quality metric that may be within the constraints of application-specific fidelity metrics such as peak signal-to-noise ratio (PSNR) [24]. Therefore, if execution is not 100% numerically correct, the application can still appear to execute correctly from the users perspective [13, 14, 24].

In case of error-intolerant applications that do not have such inherent algorithmic tolerance, even a single bit error could result in unacceptable program execution. In this class, we have examined five applications: Black-Scholes and Binomial models for European-style options in financial engineering; one-dimension Haar wavelet transform; fast Walsh transform; and Eigenvalues of a symmetric matrix.

4.1 FP Instruction-Level Memoization

We focus on the individual FPUs to observe the dispersion of the input operands at the finest granularity. To expose the value locality for each FPU operations, we consider a private FIFO for every individual FPU. These FIFOs have a small depth and keep the distinct sets of the input operands in the order of instruction arrivals. The FIFO matches a set of incoming input operands and the current content of entries of FIFO using a matching constraint. Since the applications exhibit varying degrees of tolerance to errors, we consider a matching constraint to limit the absolute numerical difference between the incoming input operands and the matched stored operands in the FIFO, as shown in Equation 1.

$$\exists i \mid input\_operands - FIFO[i] \mid \leq threshold \tag{1}$$

By choosing different `threshold` values two matching constraints can be applied: (i) *Exact matching* constraint by setting `threshold=0` that enforces zero numerical difference, i.e., full bit-by-bit matching of the input operands of the FPU with the FIFO's entries. This constraint is applicable for the error-intolerant applications. (ii) *Approximate matching* constraint by setting `threshold > 0` that relaxes the criteria of the exact matching during comparison of the operands by accepting some degree of numerical difference. Setting `threshold` value in the range of [1, 0) allow mismatches in the less significant bits of the fraction parts. For our image processing filters it is required to set `threshold` value such that to guarantee the PSNR of greater than 30dB – this is generally considered acceptable from users perspective in image processing applications.

Images in Figure 2 are the outputs of Sobel filter with different `threshold` values of 0, 0.2, 0.4, 0.6, 1.0. As shown, the `threshold=0` results in the exact matching without any quality degradation (PSNR=∞). Increasing the `threshold` value leads to higher numerical error acceptance that decreases the PSNR: `threshold=0.4` causes PSNR of 40dB, while `threshold=1.0` lowers PSNR to 30dB. Images in Figure 3 are the outputs of Gaussian filter with the same `threshold` range. The `threshold=0` guarantees the absence of any numerical error therefore does not degrade the quality. By increasing the `threshold` value the PNSR decreases. The `threshold=0.8` causes PSNR of 30dB which is acceptable from users perspective. However, further increasing of `threshold` produces unacceptable quality.

The same experiments are repeated with a different input image (*book*)

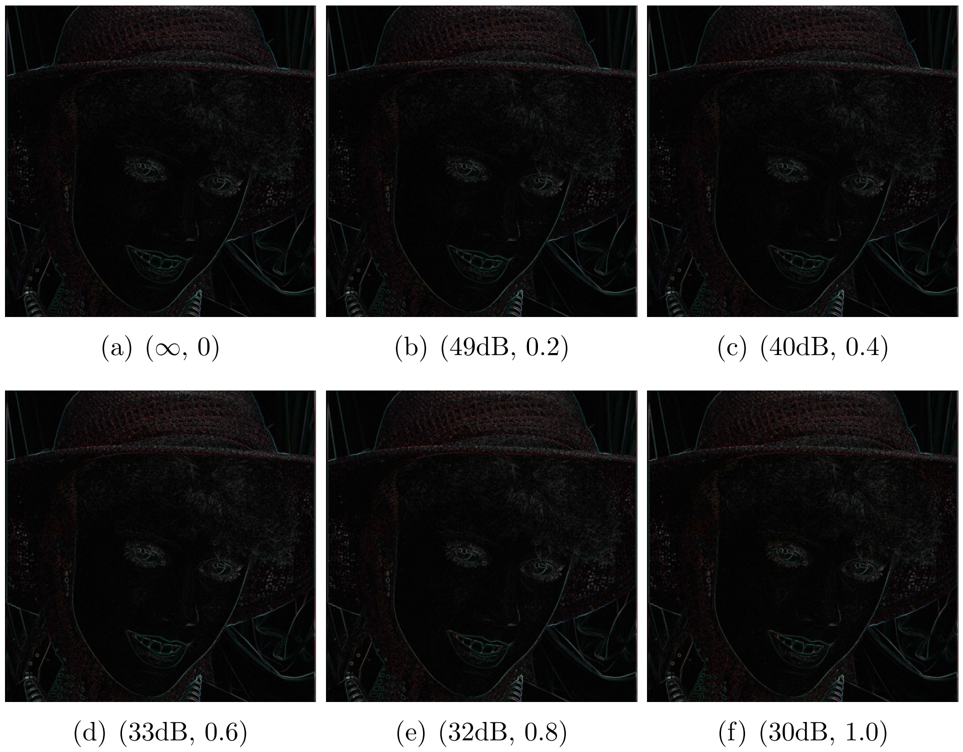
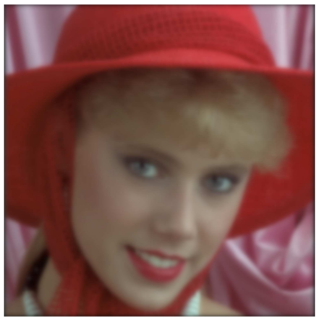

(a) (∞, 0) (b) (49dB, 0.2) (c) (40dB, 0.4) (d) (33dB, 0.6) (e) (32dB, 0.8) (f) (30dB, 1.0)

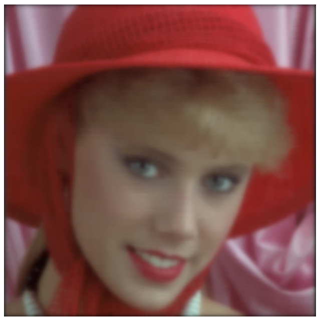
Figure 2: Each image shows a pair of the PSNR and the desired `threshold` of approximation for Sobel filter with *face* as the input image.

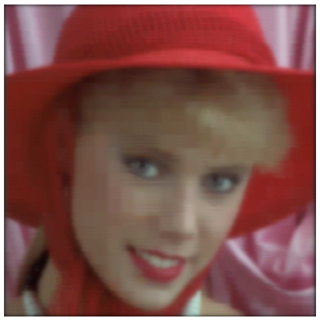
for both filters in Figure 4 and Figure 5. As shown, for this input image both Sobel and Gaussian filters experience a cutoff `threshold` value of 0.2, as the `threshold=0.4` produces PSNR of less than 30dB.

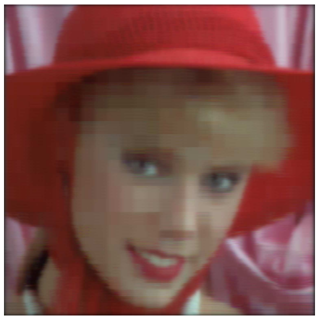
We assess the overall hit rate of applications when the FPUs utilizing FIFOs with various FIFO length of 2, 4, 8, 16, 32, and 64 entries. Increasing the FIFO size with 2 entries by a factor of 2×, 4×, 8×, 16×, and 32× led to 2%, 4%, 8%, 12%, and 17% higher hit rates. The hit rate increases less than 20% when the size of FIFOs is increased from 2 to 64. Therefore, we have used the FIFOs with 2 entries for our proposed temporal memoization technique, and we evaluate its energy saving in Section 5.

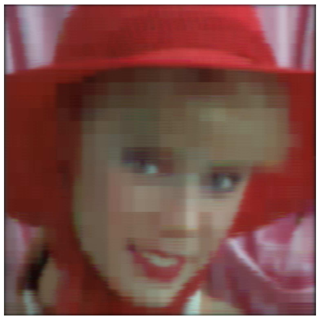
We also measure the overall hit rate of the FIFOs for different types of the FPUs. Figure 6 shows the hit rates for five types of the FPUs when executing Sobel filter with two different input images. As shown, the hit rate depends on the FPU operations. The SQRT displays the highest hit rate for


(a) (∞, 0)


(b) (46dB, 0.2)


(c) (38dB, 0.4)


(d) (34dB, 0.6)


(e) (30dB, 0.8)

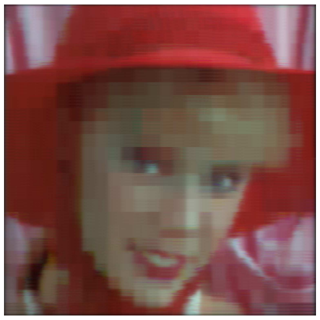
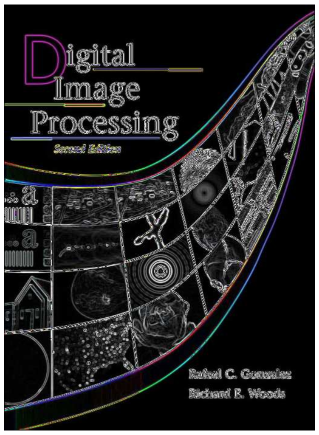
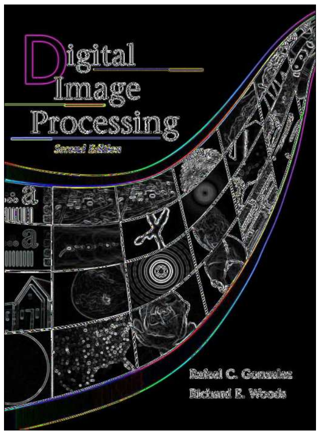

(f) (28dB, 1.0)

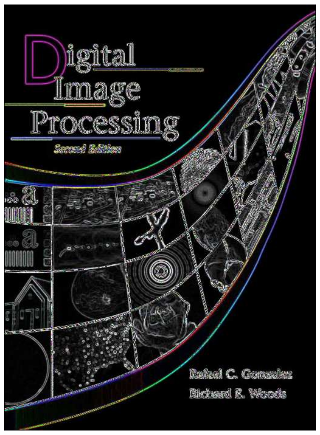
Figure 3: Each image shows a pair of the PSNR and the desired `threshold` of approximation for Gaussian filter with *face* as the input image.



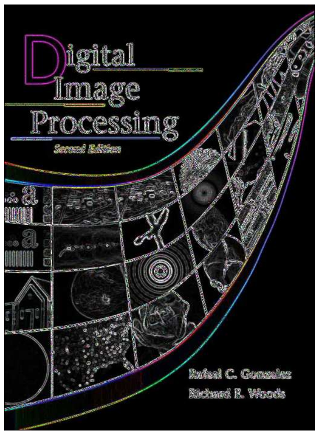
(a) (∞, 0)



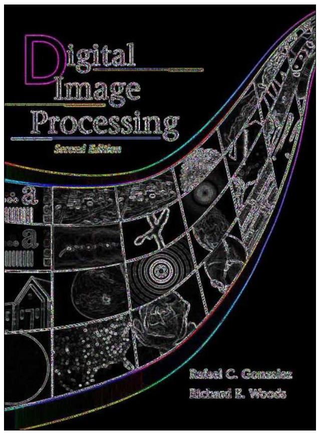
(b) (31dB, 0.2)



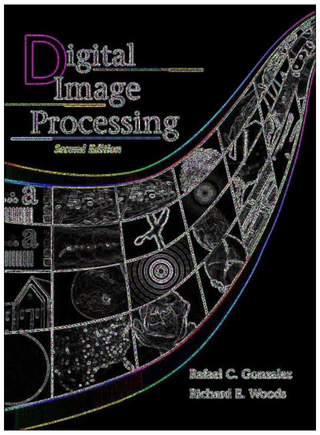
(c) (25dB, 0.4)



(d) (22dB, 0.6)

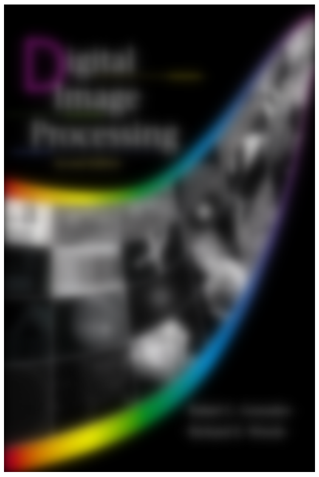


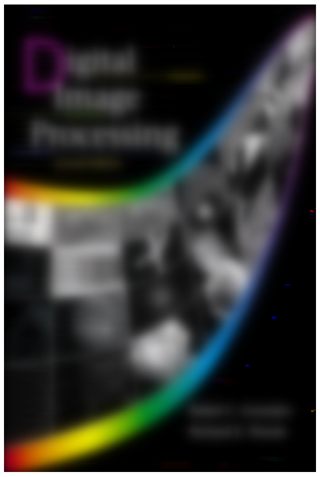
(e) (20dB, 0.8)

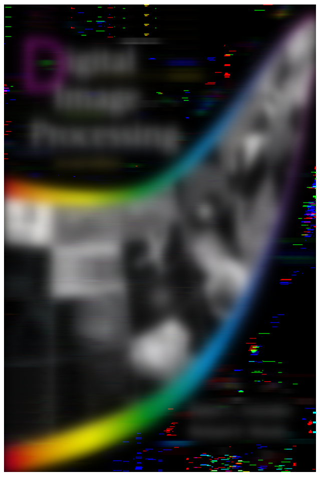


(f) (19dB, 1.0)

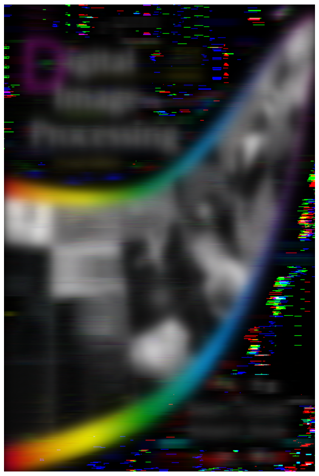
Figure 4: Each image shows a pair of the PSNR and the desired `threshold` of approximation for Sobel filter with *book* as the input image.

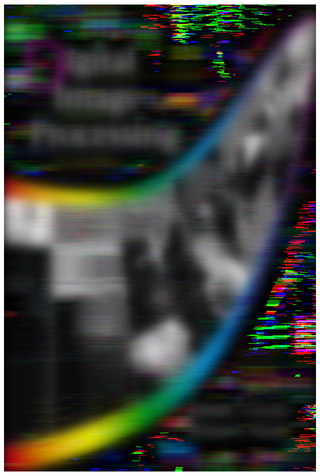


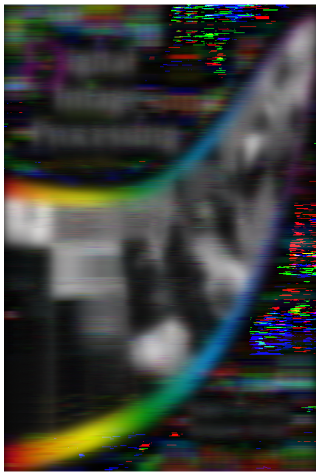




(a) (∞, 0) (b) (42dB, 0.2) (c) (25dB, 0.4)

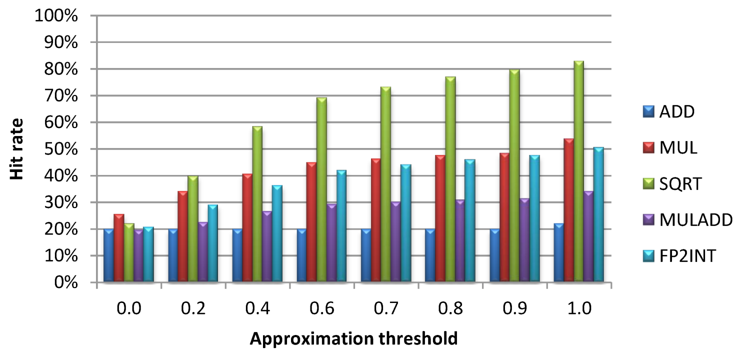




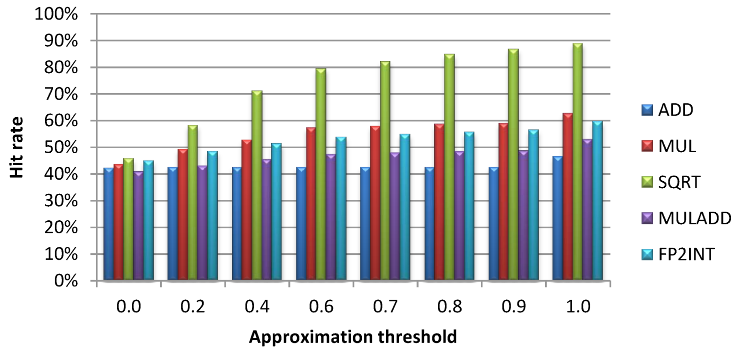


(d) (21dB, 0.6) (e) (18dB, 0.8) (f) (18dB, 1.0)

Figure 5: Each image shows a pair of the PSNR and the desired `threshold` of approximation for Gaussian filter with *book* as the input image.

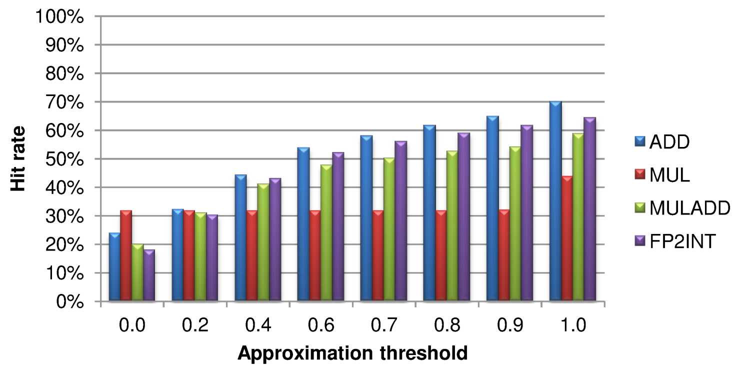


(a) input image *face*

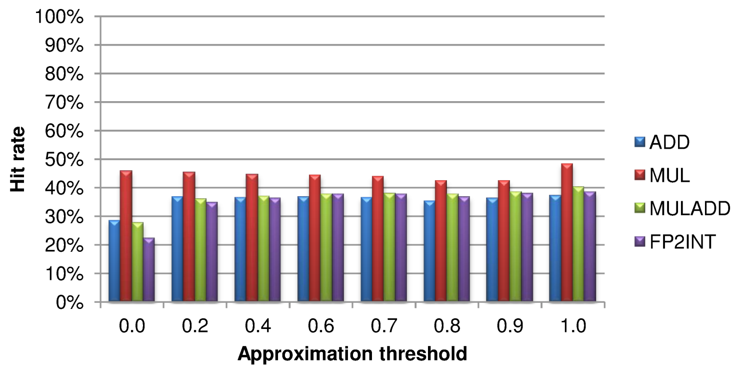


(b) input image *book*

Figure 6: Hit rate of the FIFOs for various FPUs as a function of approximation `threshold` when executing Sobel filter.



(a) input image *face*



(b) input image *book*

Figure 7: Hit rate of the FIFOs for various FPUs as a function of approximation `threshold` when executing Gaussian filter.

`threshold > 0`. With *face* as the input image, the SQRT exhibits 22%–83% hit rate for the `threshold` range of 0–1. It shows 46%–89% hit rate with *book* as the input image. All the FIFOs display a hit rate of greater than 20%; for instance, the MULADD experience hit rates of 20%–34% (or 41%–53%) with *face* (or *book*) as the input image. Clearly, the temporal value locality is a function of both operation type and input data.

We note that temporal value locality has been observed earlier in single-core architectures [15]. Our results show that the data-level parallel execution across both application classes also displays a significant value locality. This is primarily due to redundant contextual information on the SCs. As a result, the FPUs of GPGPUs experience a congested temporal value locality caused by the sub-wavefront time-multiplexing on the SCs that can be exposed by

Table 1: Kernels with selected input parameters and `threshold`.

Kernel	Input parameter	`threshold`
Sobel	*face* (1536×1536)	1.0
Gaussian	*face* (1536×1536)	0.8
Haar	1024	0.046
BinomialOption	20	0.000025
BlackScholes	20	0.000025
FWT	1000000	0.0
EigenValue	1000×1000	0.0

small FIFOs.

Table 1 shows the full set of kernels with the input parameters and the selected `threshold` of approximation. For Gaussian and Sobel filters, as the error-tolerant kernels, we use relatively large `threshold` values. These `threshold` values allow up to 1.0 (for Sobel) or 0.8 (for Gaussian) absolute numerical error during the operator matching, while providing the acceptable PSNR of greater than 30dB for the input image. Due to the absence of proper *fidelity metric* and domain expert knowledge for the rest of applications, we consider them as the error-intolerant kernels that need full numerical correctness. However, during our experiments we informed that three more kernels (Haar, BinomialOption, and BlackScholes) are able to tolerate small numerical errors. Setting the `threshold=0.046` for Harr and `threshold=0.000025` for BinomialOption/BlackScholes produces the output values that are accepted by the test program executed in the host code[1]. For FWT and EigenValue, we set the exact matching (`threshold=0.0`) that enforces full bit-by-bit matching of the operands.

Figure 8 shows the hit rate of the FIFOs for various FPUs during execution of the kernels with the input parameters and the `threshold` values listed in Table 1. It also shows the weighted average hit rate of the activated FPUs. Among all the FPUs, the SQRT and the FP-to-INT exhibit the highest hit rate of 97% across the kernels. The ADD, MUL, RECIP, MULADD show the maximum hit rate of 94%. Among the error-intolerant kernels, EigenValue shows an average hit rate of 94% across seven activated FPUs.

[1]The output values of the kernels are compared against the output values computed by the host code and the test program results in *passed*.

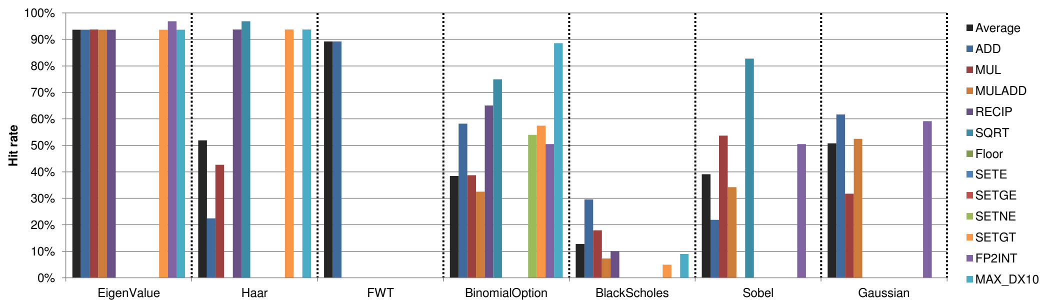


Figure 8: Hit rate of the FIFOs for *activated* FPUs during kernel execution with parameters listed in Table 1.

4.2 Resilient Architecture Utilizing Temporal Memoization

We now describe the design of a resilient GPGPU architecture that utilizes the temporal memoization technique. In Evergreen GPGPUs, every FPU has four execution stages and a throughput of one instruction per cycle. We instrument every FPU pipeline with the error detection and correction mechanisms proposed in [6, 9]. Essentially, every stage uses EDS circuit sensors to detect the timing errors by propagating an error signal toward the end of pipeline that finally reaches to the ECU, the error control unit. The ECU triggers the recovery mechanism through flushing, and issuing the errant instruction multiple times [9]. This forms the baseline 'detect-then-correct' mechanism shown as the white components in Figure 9.

To exploit the value locality, we tightly couple the FPU pipeline with our proposed temporal memoization module (shown as the gray components in Figure 9). This module has essentially a single-cycle LUT, and a set of flip-flops and buffers to propagate signals through the pipeline. The LUT is shown in the bottom part of Figure 9 and is composed of two parts: (i) a FIFO with two entries; (ii) a set of combinational comparators. In every entry, the FIFO maintains a set of input operands (e.g., in Figure 9, two inputs) and the computed result provided by the output of the FPU in the last stage (Q_S). The parallel combinational comparators implement the two matching constraints, and are programmable through a 32-bit memory-mapped register as a masking vector. They concurrently make either a full or partial comparison of the input operands with the stored operands in each entry based on the masking vector. The LUT works in parallel with

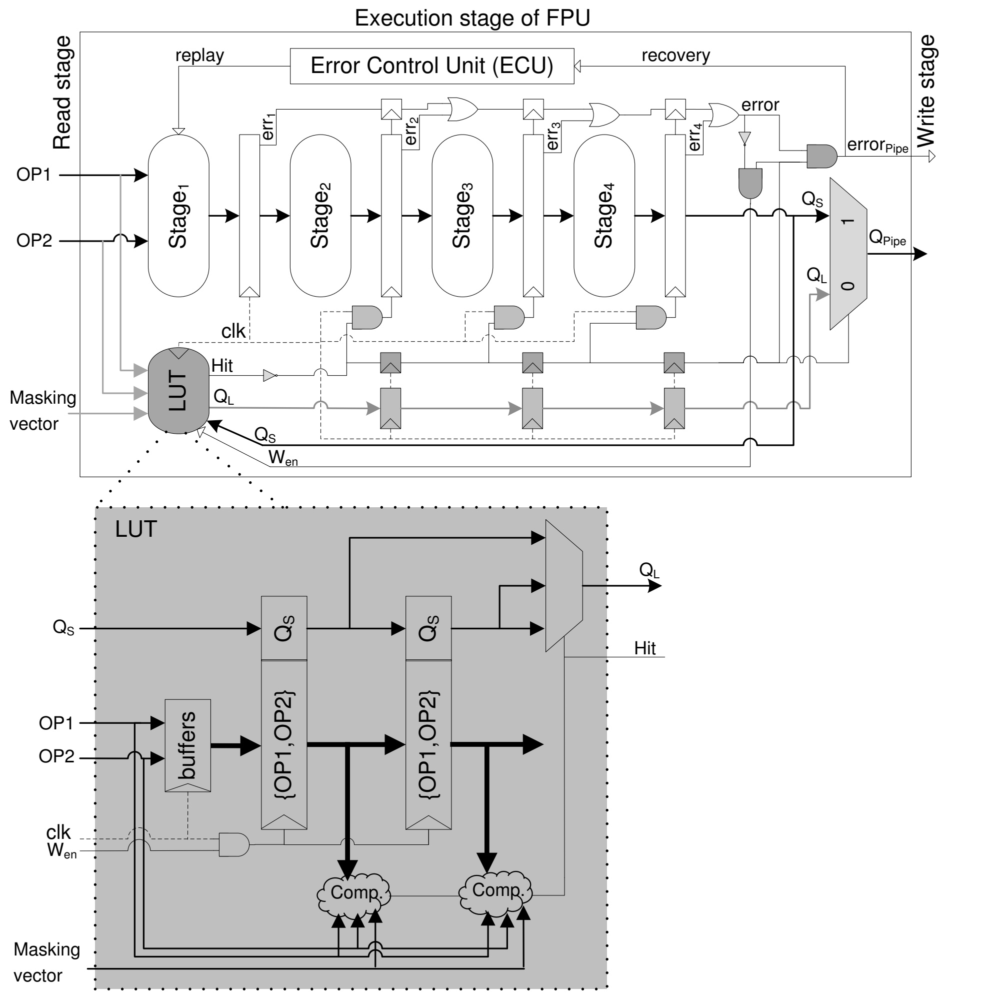


Figure 9: Execution stage of the resilient FPU with temporal memoization module (in gray); the LUT is shown below.

the first stage of the FPU. Therefore, for every set of input operands, the LUT searches the FIFO to find a match between the input operands and the operand values stored in the entries (i.e., whether the matching constraint in Equation 1 is satisfied or not). A match directly results in reuse of results computed earlier; therefore avoiding redundant execution and saving power, i.e., when {Hit, Error}={1,1/0}. Furthermore, this affords the temporal memoization module an opportunity to correct an errant instruction with zero cycle penalty.

The matching constraints are programmable and also allow commutativity of the operands where applicable. The FIFO maintains a limited number of recent distinct sets. Therefore, if a set of incoming input operands does not satisfy either matching constraints, the FIFO will be updated by cleaning its last entry and inserting the new incoming operands accordingly.

To enable reuse, the LUT propagates a hit signal alongside with the previously-computed result (Q_L) toward the end of pipeline. The LUT raises the hit signal that squashes the remaining stages of the FPU to avoid the redundant computation by clock-gating; the clock-gating signal is forwarded to the rest of stages, cycle by cycle. The stored result is also propagated toward the end of pipeline for the reuse purpose. The hit signal selects the propagated output of the LUT (Q_L) as the output of the FPU; it also disables the propagation of timing error signal (if any) to the ECU, thus avoids the costly recovery. Therefore, each hit event reduces energy by locally retrieving the result from the LUT, rather than doing full re-execution by the FPU. In case of a LUT miss, the FIFO is updated to maintain the last recently computed values. It is implemented through a write enable signal (W_{en}) that ensures there is no timing error during execution of all the stages of the FPU for computing Q_S. Finally, if a simultaneous timing error and miss occurred, the error signal will be propagated to the ECU that triggers the baseline recovery. Table 2 summarizes these four states. Each application has full control over the temporal memoization module as a programmable module through the memory-mapped registers. To determine the matching constraints, the application can set the 32-bit memory-mapped register of the masking vector accordingly. The error-tolerant applications can set the masking vector according to the `threshold` value to ignore the differences of the operands in the less significant bits of the fraction part. With the approximate matching constraint, the pair of instructions with two different input operands will have the same output. On the other hand, the exact matching constraint enables the reuse of the previously-computed

Table 2: Timing error handling with temporal memoization module.

Hit	Error	Action	Q_{Pipe}
0	0	Normal execution + LUT update	Q_S
0	1	Triggering baseline recovery (ECU)	Q_S
1	0	LUT output reuse + FPU clock-gating	Q_L
1	1	LUT output reuse + FPU clock-gating + masking error	Q_L

results stored in the LUT while maintaining the full precision for the error-intolerant applications. Moreover, if an application lacks value locality, it can disable the entire memoization module by power-gating thus avoid any power penalty. Further, compiler-directed analysis techniques or domain experts with some application knowledge can also store pre-computed values in the LUT to use the most probable or critical results.

5 Experimental Results

Our methodology uses the AMD Evergreen GPGPUs, but can be applied to other GPGPU architectures as well. Multi2Sim [25], a cycle-accurate CPU-GPU simulation framework, is modified to collect the statistics for computing the temporal value locality out of 27 single precision floating-point instructions. The modified simulator code is available for downloading at [28]. The naive binaries of AMD APP SDK v2.5 [23] kernels are run on the simulator; the input values for the kernels are generated by the default OpenCL host program based on the parameters listed in Table 1. We analyzed the effectiveness of proposed technique in the presence of timing errors and voltage overscaling on TSMC 45-nm ASIC flow.

5.1 Implementation of Temporal Memoization Module

To keep the focus on the energy-hungry high-latency FP pipelines, we assume that the memory blocks are resilient, for instance by utilizing the tunable replica bits [7]. Since the fetch and decode stages display a low criticality [16], we focus on the execution stage consisting of six frequently exercised functional units: ADD, MUL, SQRT, RECIP, MULADD, FP2INT. On Evergreen, every ALU functional unit has a latency of four cycles and a throughput of one instruction per cycle [27]. Therefore, VHDL codes of the

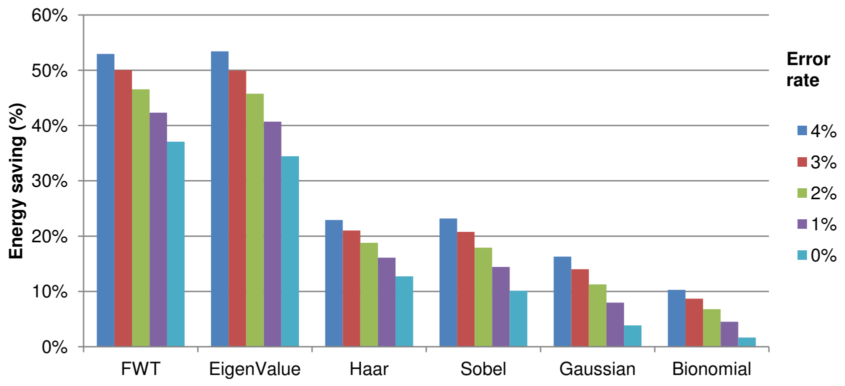


Figure 10: Energy saving of the proposed temporal memoization architecture while experiencing a range of timing error rates– considering energy consumption of ADD, MUL, SQRT, RECIP, MULADD, FP2INT.

FPUs are generated and optimized using FloPoCo [26] – an arithmetic synthesizable FP core generator. To achieve a balanced clock frequency across the FP pipelines, the RECIP has a latency of 16 cycles, while the rest of the FPU have four cycles latency.

The temporal memoization module for each FPU operations is described in Verilog synthesizable RTL. To integrate the resilient architecture, the memoization modules are integrated into the FPUs pipelines with the baseline recovery mechanism. This baseline recovery mechanism costs 12 cycles per error. Finally, the entire design is synthesized and mapped using the TSMC 45nm technology library. The front-end flow with multi V_{TH} cells has been performed using *Synopsys Design Compiler* with the topographical features, while *Synopsys IC Compiler* has been used for the back-end. The design has been optimized for timing a signoff frequency of 1GHz at (SS/0.81V/125°C), and then optimized for power using high V_{TH} cells. The memoization module does not limit the clock frequency as it has a positive slack of 14% of the clock period.

5.2 Energy Saving

We compare the energy saving for a range of timing error rates [0%, 4%] on the execution state of the six frequently exercised FPUs. Our implementation excludes the fact that the temporal memoization module may produce an erroneous result, because the module has a positive slack of 14% of the clock

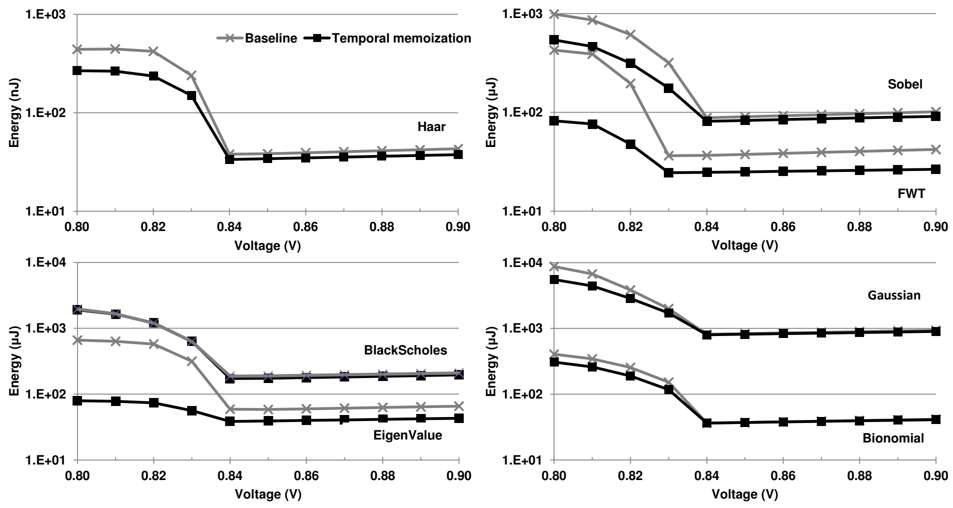


Figure 11: Total energy consumption of the proposed temporal memoization architecture in comparison with the baseline architecture under voltage overscaling– considering energy consumption of ADD, MUL, SQRT, RECIP, MULADD, FP2INT.

period. Therefore it is unlikely to face any timing errors. When there are no timing errors (0%), the proposed architecture has an average energy saving of 13% compared to the baseline architecture for all the applications. This scenario is similar to the value prediction techniques as proposed in [15], that is extended to GPGPU architectures. Moreover, as shown in Fig 10, the temporal memoization technique has a great potential of energy saving in the high error rate circumstances. On average 13%, 17%, 20%, 23%, 25% lower total energy is achieved compared to the baseline at the error rates of 0%, 1%, 2%, 3%, 4%. This is accomplished through the efficient memoization-based timing error recovery that does not impose any latency penalty as opposed to the baseline recovery.

5.3 Memoization-Based Voltage Overscaling

We also compare energy saving of the proposed architecture with a baseline architecture that also utilizes recent resilient techniques [11, 9]. For the baseline architecture, we consider the decoupling queues technique [11] with the error detection of EDS, and the baseline recovery mechanism of the multiple-issue instruction replay [9] adapted for the FPUs to support scalability. Our proposed temporal memoization architecture superposes the temporal memoization modules on the baseline architecture. In our experiments the EDS, the ECU and the temporal memoization modules are always ON; applications have only control on selecting either *approximate* or *exact* matching constraint.

We assess the efficacy of the proposed architecture in the voltage overscaling regime, while maintaining the constant clock frequency. We scale down the voltage of the FPUs in the range of 0.9V–0.8V. To ensure always correct functionality of the temporal memoization module, we maintain its operating voltage at the fixed nominal 0.9V. Voltage scaling feature of *Synopsys PrimeTime* is employed to analyze the delay and power variations under the voltage overscaling. Then, the voltage overscaling-induced delay is back annotated to the post-layout simulation which is coupled with Multi2Sim simulator to quantify the timing error rate. The baseline architecture triggers the recovery mechanism when any voltage overscaling-induced timing error occurs, while our proposed architecture does it in case of simultaneous events of the miss and the error.

Figure 11 illustrates the energy consumption of the two architectures at different voltage overscaling points for six applications. The proposed memoization architecture exhibits a great potential of survival in the voltage overscaling: (i) For all the applications, the proposed architecture achieves 13% average energy saving at the nominal voltage of 0.9V. At this operating voltage there is no timing error which is similar to the value prediction. (ii) Scaling down the voltage to 0.84V, reduces the gain of our energy saving to 11% since the FPUs of the baseline are nicely reduced their total power as consequence of negligible error rate, while we cannot proportionally scale down the power of the temporal memoization modules. (iii) Beyond 0.84V, our technique surpasses the baseline architecture due to the abrupt increasing of the error rate and therefore frequent recoveries. At voltage of 0.8V, the proposed technique reaches an average energy saving of 44%.

6 Conclusion

We exploit value locality in improving timing error correction in GPGPUs. A fast lightweight temporal memoization module independently stores recent error-free executions of a FPU which is sufficient enough to temporarily protect individual FPUs against timing errors. To efficiently reuse computations, the technique supports both exact and approximate error corrections for error-intolerant general-purpose and error-tolerant image processing applications, respectively. These real-world applications exhibit a low entropy due to the high contextual information. This avoids costly recovery, therefore improves the energy efficiency and reduces the total energy by average savings of 13% (for 0% timing error rate) to 25% (for 4% timing error rate). This technique also surpasses the baseline architecture by enhancing robustness and energy saving in the voltage overscaling regime.

7 Acknowledgments

This work was supported by the NSFs Variability Expedition under award n. 1029783, ERC-AdG MultiTherman (291125), and FP7 Virtical (288574).

References

[1] *Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110*, 2012.

[2] ITRS [Online]. Available: `http://public.itrs.net`

[3] M.R. Kakoee, et al., *Variation-Tolerant Architecture for Ultra Low Power Shared-L1 Processor Clusters,* IEEE Trans. on Circuits and Systems II, vol.59, no.12, pp.927-931, Dec. 2012.

[4] D. Jeon, et al., *Design Methodology for Voltage-Overscaled Ultra-Low-Power Systems,* IEEE Trans. on Circuits and Systems II, vol.59, no.12, pp.952-956, Dec. 2012.

[5] P. Gupta, et al., *Underdesigned and Opportunistic Computing in Presence of Hardware Variability* IEEE TCAD, 32(1) (2013) pp. 489-499.

[6] K.A. Bowman, et al., *Energy-Efficient and Metastability-Immune Resilient Circuits for Dynamic Variation Tolerance,* IEEE JSSC, 2009.

[7] A. Raychowdhury, et al., *Tunable Replica Bits for Dynamic Variation Tolerance in 8T SRAM Arrays,* IEEE JSSC, 2011.

[8] I. Kazi, et al., *A ReRAM-based non-volatile flip-flop with sub-VT read and CMOS voltage-compatible write,* Proc. IEEE NEWCAS, 2013.

[9] K.A. Bowman, et al., *A 45 nm Resilient Microprocessor Core for Dynamic Variation Tolerance,* IEEE JSSC, 2011.

[10] M.-M. Papadopoulou, et al., *Micro-benchmarking the GT200 GPU* Technical report, Computer Group, ECE, University of Toronto, 2009.

[11] R. Pawlowski, et al., *A 530mV 10-lane SIMD Processor with Variation Resiliency in 45nm SOI,* Proc. IEEE ISSCC, 2012, pp. 492-494.

[12] A. Sodani and G.S. Sohi, *Dynamic Instruction Reuse,* Proc. IEEE/ACM ISCA, 1997, pp. 492-494.

[13] C. A. Martinez, et al., *Dynamic Toler-ance Region Computing for Multimedia,* IEEE Transactions on Computers, vol.61, no.5, pp.650-665, May 2012.

[14] C. Alvarez, et al., *Fuzzy Memoization for Floating-point Multimedia Applications,* IEEE Transactions on Computers , vol.54, no.7, pp.922-927, July 2005.

[15] M. -H. Lipasti, et al., *Value Locality and Load Value Prediction,* Proc. ACM ASPLOS, 1996, pp. 138-147.

[16] A. Rahimi, et al. *Analysis of Instruction-level Vulnerability to Dynamic Voltage and Temperature Variations,* Proc. IEEE/ACM DATE, 2012.

[17] A. Rahimi, et al., *Procedure hopping: A low overhead solution to mitigate variability in shared-L1 processor clusters,* Proc. ACM/IEEE ISLPED, 2012, pp. 415-420.

[18] A. Rahimi, et al., *Hierarchically Focused Guardbanding: An Adaptive Approach to Mitigate PVT Variations and Aging,* Proc. ACM/IEEE DATE, 2013.

[19] A. Rahimi, et al., *Aging-Aware Compiler-Directed VLIW Assignment for GPU Architectures,* Proc. ACM/IEEE DAC, 2013.

[20] A. Rahimi, et al., *Spatial Memoization: Concurrent Instruction Reuse to Correct Timing Errors in SIMD Architectures,* IEEE TCAS II, 2013.

[21] A. Rahimi, et al., *Temporal memoization for energy-efficient timing error recovery in GPGPUs,* Proc. ACM/IEEE DATE, 2014.

[22] K. Chakraborty, et al., *Efficiently Tolerating Timing Violations in Pipelined Microprocessors,* Proc. ACM/IEEE DAC, 2013.

[23] AMD APP SDK 2.5 [Online]. Available: `www.amd.com/stream`

[24] M. A. Breuer, *Multi-media Applications and Imprecise Computation,* Proc. IEEE DSD, 2005.

[25] Multi2Sim [Online]. Available: `http://www.multi2sim.org/`

[26] FloPoCo [Online]. Available: `http://flopoco.gforge.inria.fr/`

[27] AMD APP OpenCL Programming Guide, Chapter 6.6.1, pp. 157, 2012.

[28] Temporal Memoization for GPGPUs [Online]. Available: `https://github.com/abbasucsd/temporal_memoization/archive/master.zip`