RFVP: Rollback-Free Value Prediction with Safe-to-Approximate Loads

Amir Yazdanbakhsh Gennady Pekhimenko[§] Bradley Thwaites
Hadi Esmaeilzadeh Taesoo Kim Onur Mutlu[§] Todd C. Mowry[§]
Georgia Institute of Technology Carnegie Mellon University[§]

Abstract

This paper aims to tackle two fundamental memory bottlenecks: limited off-chip bandwidth (bandwidth wall) and long access latency (memory wall). To achieve this goal, our approach exploits the inherent error resilience of a wide range of applications. We introduce an approximation technique, called Rollback-Free Value Prediction (RFVP). When certain safe-to-approximate load operations miss in the cache, RFVP predicts the requested values. However, RFVP never checks for or recovers from load value mispredictions, hence avoiding the high cost of pipeline flushes and re-executions. RFVP mitigates the memory wall by enabling the execution to continue without stalling for long-latency memory accesses. To mitigate the bandwidth wall, RFVP drops some fraction of load requests which miss in the cache after predicting their values. Dropping requests reduces memory bandwidth contention by removing them from the system. The drop rate then becomes a knob to control the tradeoff between performance/energy efficiency and output quality.

For a diverse set of applications from Rodinia, Mars, and NVIDIA SDK, employing RFVP with a 14KB predictor per streaming multiprocessor (SM) in a modern GPU delivers, on average, 40% speedup and 31% energy reduction, with average 8.8% quality loss. With 10% loss in quality, the benefits reach a maximum of 2.4× speedup and 2.0× energy reduction. As an extension, we also evaluate RFVP's latency benefits for a single core CPU. For a subset of the SPEC CFP 2000/2006 benchmarks that are amenable to safe approximation, RFVP achieves, on average, 8% speedup and 6% energy reduction, with 0.9% average quality loss.

1. Introduction

The disparity between the speed of processors and off-chip memory is one of the main challenges in microprocessor design. Loads that miss in the last level cache can take hundreds of cycles to deliver data. This long latency causes frequent long stalls in the processor. This problem is known as the memory wall. Modern GPUs exploit data parallelism to hide main memory latency. However, this solution suffers from another fundamental bottleneck: the limited off-chip communication bandwidth. In fact, memory bandwidth is predicted to be one of the main performance-limiting factors in accelerator-rich architectures as technology scales [12]. This problem is known as the bandwidth wall [47]. Fortunately, there is an opportunity to leverage the inherent error resiliency of many emerging applications and services to tackle these problems. This paper exploits this opportunity.

Large classes of emerging applications such as web search, data analytics, machine learning, cyber-physical systems, augmented reality, and vision can tolerate error in large parts of their execution. Hence the growing interest in developing general-purpose approximation techniques. These techniques accept error in computation and trade *Quality of Result* for gains in performance, energy, and storage capacity. These techniques include (a) voltage over-scaling [17, 10], (b) loop perforation [56], (c) loop early termination [6], (d) computation substitution [49, 18, 3], (e) memoization [2, 48, 5], (f) limited fault recovery [15, 27], and (g) approximate data storage [33, 51]. However, there is a lack of approximation techniques that address the memory system performance bottlenecks of long access latency and limited off-chip bandwidth.

To mitigate these memory subsystem bottlenecks, this paper introduces a new approximation technique called *Rollback-Free Value Prediction (RFVP)*. The key idea is to predict the value of the *safe-to-approximate* loads when they miss in the cache, without checking for mispredictions or recovering from them, thus avoiding the high cost of pipeline flushes and re-executions. RFVP mitigates the memory wall by enabling the computation to continue without stalling for long-latency memory accesses of safe-to-approximate loads. To tackle the bandwidth wall, RFVP drops *a certain fraction* of the *cache misses* after predicting their values. Dropping these requests reduces the memory bandwidth demand as well as memory and cache contention. The drop rate becomes a knob to control the tradeoff between performance-energy and quality.

In this work, we aim to devise concepts and mechanisms that maximize RFVP's opportunities for speedup and energy gains, while keeping the quality degradations acceptably small. We provide architectural mechanisms to control quality degradation and always guarantee execution without catastrophic failures by leveraging programmer annotations. RFVP shares some similarities with traditional *exact* value prediction techniques [54, 32, 16, 20, 44] that can mitigate the memory wall. However, it fundamentally differs from the prior work in that it does not check for misspeculations and does not recover from them. Consequently, RFVP not only avoids the high cost of recovery, but is able to drop a fraction of the memory requests to mitigate the bandwidth wall.

This paper makes the following contributions:

(1) We introduce a new approximation technique, Rollback-Free Value Prediction (RFVP), that addresses two important system bottlenecks: long memory latency and limited off-chip bandwidth by utilizing value prediction mechanisms.

(2) We propose a new multi-value prediction architecture for SIMD load instructions in GPUs that request multiple values

in one access. To minimize the overhead of the multi-value predictor, we exploit the insight that there is significant value similarity across accesses in the adjacent threads (e.g., adjacent pixels in an image). Such value similarity has been shown in recent works [48, 5]. We use the two-delta predictor [16] as the base for our multi-value predictor. We perform a Pareto-optimality analysis to explore the design space of our predictor and apply the optimal design in a modern GPU.

(3) We provide a comprehensive evaluation of RFVP using a modern Fermi GPU architecture. For a diverse set of benchmarks from Rodinia, Mars, and NVIDIA SDK, employing RFVP delivers, on average, 40% speedup and 31% energy reduction, with average 8.8% quality loss. With less than 10% quality loss, the benefits reach a maximum of 2.4× speedup and 2.0× energy reduction. For a subset of SPEC CFP 2000/2006 benchmarks that are amenable to safe approximation, employing RFVP in a modern CPU achieves, on average, 8% speedup and 6% energy reduction, with 0.9% average quality loss.

2. Architecture Design for RFVP

2.1. Rollback-Free Value Prediction

Motivation. GPU architectures exploit data-level parallelism through many-thread SIMD execution to mitigate the penalties of long memory access latency. Concurrent SIMD threads issue many simultaneous memory accesses that require high off-chip bandwidth–one of the main bottlenecks for modern GPUs. Figure 1 illustrates the effects of memory bandwidth on application performance by varying the available off-chip bandwidth in the Fermi architecture. Many of the applications in our workload pool benefit significantly from increased bandwidth. For instance, a system with twice the baseline off-chip bandwidth enjoys 26% average speedup, with up to 80% speedup for the s.srad2 application. These results support that lowering bandwidth contention can result in significant performance benefits. RFVP exploits this insight and aims to lower the bandwidth pressure by dropping a fraction of the predicted safe-to-approximate loads, trading output quality for gains in performance and energy efficiency.

Overview. The key idea of rollback-free value prediction (RFVP) is to predict the values of the safe-to-approximate loads when they miss in the cache with no checks or recovery from misspeculations. RFVP not only avoids the high cost of checks and rollbacks but also drops a fraction of the cache misses. Dropping these misses enables RFVP to mitigate the bottleneck of limited off-chip bandwidth, and does not affect output quality when the value prediction is correct. All other requests are serviced normally, allowing the core to benefit from the spatial and temporal locality in future accesses.

Drop rate becomes a knob to control the tradeoff between performance/energy gains and quality loss. Higher drop rates cause the core to use more predicted approximate values and avoid accessing main memory. We expose the drop rate as a microarchitectural mechanism to the software. The compiler or the runtime system can use this knob to control the performance/energy and quality tradeoff. Furthermore, RFVP enables the core to continue without stalling for long-latency memory accesses that service the predicted load misses. Consequently, these cache-missing loads are removed from the critical path of the execution. We now elaborate on the safety guarantees with RFVP, its ISA extensions and their semantics, and the microarchitectural integration of RFVP.

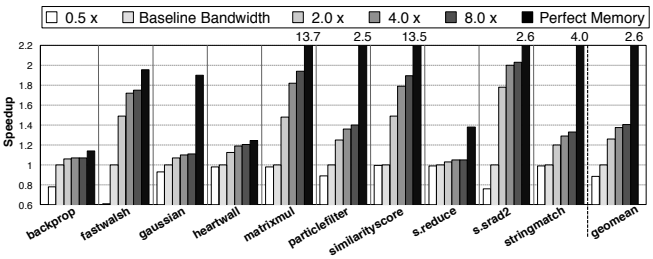


Figure 1: Performance improvement with different DRAM bandwidth and perfect memory (last bar). The baseline bandwidth is 173.25 GB/sec (based on the NVIDIA GTX 480 chipset with Fermi architecture). (The N legend indicates a configuration with N times the bandwidth of the baseline.)

2.2. Safe Approximation with RFVP

Not all load instructions can be safely approximated. For example, loads that affect critical data segments, array indices, pointer addresses, or control flow conditionals are usually not safe to approximate. RFVP is *not* used to predict the value of these loads. Furthermore, as prior work in approximation showed [50], safety is a semantic property of the program, and language construction with programmer annotations is necessary to identify safely-approximable instructions. As a result, the common and necessary practice is to rely on programming language support along with compiler optimizations to identify which instructions are safe to approximate [6, 8, 50, 17, 18]. Similarly, RFVP requires programmer annotations to determine the set of candidate load instructions for safe approximation. Therefore, any architecture that leverages RFVP needs to provide ISA extensions that enable the compiler to mark the safely-approximable loads. Section 2.3 describes these ISA extensions. Section 3 describes the details of our compilation and language support for RFVP.

2.3. Instruction Set Architecture to Support RFVP

We extend the ISA with two new features: (1) approximate load instructions, and (2) a new instruction for setting the drop rate. Similar to prior work [17], we extend the ISA with dual approximate versions of the load instructions. A bit in the opcode is set when a load is approximate, thus permitting the microarchitecture to use RFVP. Otherwise, the load is *precise* and must be executed normally. Executing an approximate load does not always invoke RFVP. RFVP is triggered *only* when the load misses in the cache. For ISAs without explicit load instructions, the compiler marks any safe-to-approximate instruction that can generate a load micro-op. RFVP will be triggered only when the load micro-op misses in the cache.

The *drop rate* is a knob that is exposed to the compiler to

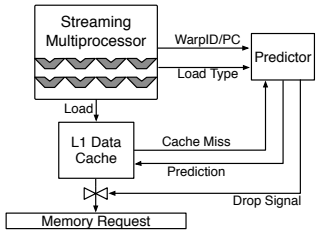


Figure 2: Microarchitecture integration of the predictor.

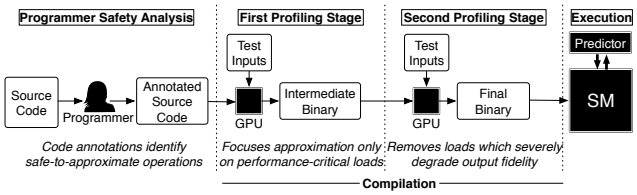


Figure 3: RFVP Workflow. Programmer annotations *guarantee* safety, while profiling *assists* output quality and performance.

control the quality tradeoffs. We provide an instruction that sets the value of a special register to the desired drop rate. This rate is usually set once during application execution (not for each load). More precisely, the drop rate is the percentage of approximate *cache misses* that will *not* initiate memory access requests, and instead trigger rollback-free value prediction [1]. When the request is not dropped, it will be considered a normal cache miss, and its value will be fetched from memory.

Semantically, an approximate load is a probabilistic load. That is, executing load.approx Reg<id>, MEMORY<address> assigns the exact value stored in MEMORY<address> to Reg<id> with some probability, referred to as the probability of exact assignment. The Reg<id> receives an arbitrary value in other cases. Intuitively, with RFVP, the probability of exact assignment is usually high for three reasons. First, our technique is triggered only by cache misses. Approximate loads which hit in the cache (usually a common case) return the correct value. Second, our automated profiling phase helps to eliminate any loads from the approximate list which are destructive to quality. Finally, even in the case of a cache miss, the value predictor may generate a correct value prediction. Our measurements with 50% drop rate show that, across all the GPU applications, the average probability of exact assignment to the approximate loads is 71%. This probability ranges from 43% to 88%. These results confirm the effectiveness of using cache misses as a trigger for RFVP. However, we do not expect the compiler to reason about these probabilities.

2.4. Integrating RFVP in the Microarchitecture

As Figure 2 illustrates, the value predictor supplies the data to the core when triggered. The core then uses the data as if it were supplied by the cache. The core commits the load instruction without any checks or pipeline stalls associated with the original miss. In the microarchitecture, we use a simple pseudo-random number generator, a Linear Feedback Shift Register (LFSR) [39], to determine when to drop the request based on the specified drop rate.

In modern GPUs, each Streaming Multiprocessor (SM) contains several Stream Processors (SP) and has its own dedicated L1. We augment each SM with an RFVP predictor that is triggered by its L1 data cache misses. Integrating the RFVP predictor with SMs requires special consideration because each GPU SIMD load instruction accesses multiple data elements for multiple concurrent threads. In the case of an approximate load miss, if the predictor drops the request, it predicts the entire cache line. The predictor supplies the requested words back to the SM, and also inserts the predicted line in the L1 cache. If RFVP did not update the cache line, the subsequent safe-to-approximate loads to the same cache line would produce another miss. Since RFVP does not predict nor drop *all* missing safe-to-approximate loads, the line would be requested from memory in the next access. Due to the temporal locality of the cache line accesses, RFVP would not be able to effectively reduce bandwidth consumption.

Since predicted lines may be written to memory, we require that any data accessed by a precise load must not share a cache line with data accessed by approximate loads. The compiler is responsible for allocating objects in memory such that precise and approximate data never share a cache line. We accomplish this by always requiring that the compiler allocate objects in memory at cache line granularity. Approximate data will always begin at a cache line boundary, and will be padded to end on a cache line boundary. Thus, we can ensure that any data predictions will not contaminate precise load operations. The same stipulation has been set forth in several recent works in approximate computing, such as EnerJ [50] and Truffle [17].

The coalescing logic in the SMs handles memory divergence and serializes the divergent threads. Since RFVP is only triggered by cache misses that happen after coalescing, RFVP is agnostic to memory divergence.

3. Language and Software Support for RFVP

Our design principle for RFVP is to maximize the opportunities for gains in performance and energy efficiency, while limiting the adverse effects of approximation on output quality. We develop a profile-directed compilation workflow, summarized in Figure 3. In the first step, the workflow uses the programmer-supplied annotations to determine the loads that are safe to approximate and *will not* cause catastrophic failures if approximated. The second step identifies the performance-critical safe-to-approximate loads. These safe-to-approximate loads are the ones that provide a higher potential for performance improvement. These performance-critical safe-to-approximate loads are the candidate for approximation with RFVP. However, approximating all of the candidate loads may significantly degrade output quality. Thus, we develop a third step that identifies which of the can-

[1] Another option is to enable dropping after a certain percentage of all cache accesses including hits. Such a policy may be desirable for controlling error in multi-kernel workloads.

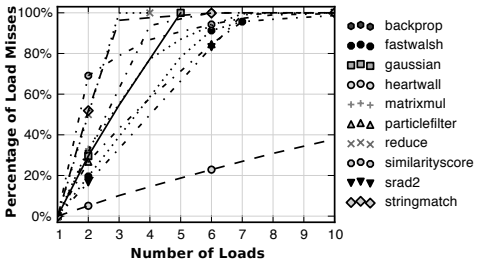


Figure 4: Cumulative distribution function (CDF) plot of the LLC load cache misses. A point (x, y) indicates that y percent of the cache misses are caused by x distinct load instructions.

didate safe-to-approximate loads need to be excluded from the RFVP-predictable set to keep the quality degradation to low and acceptable levels. This step also determines the drop rate.[2]

3.1. Providing Safety Guarantees

The first step is to ensure that loads which can cause safety violations are excluded from RFVP. Any viable approximation technique, including ours, needs to provide strict safety guarantees. That is to say applying approximation should only cause graceful quality degradations without catastrophic failures, e.g., segmentation faults or infinite loops.

Safety is a semantic property of a program [50, 8]. Therefore, only the programmer can reliably identify which instructions are safe to approximate. For example, EnerJ [50] provides language constructs and compiler support for annotating safe-to-approximate operations in Java. We rely on similar techniques. The programmer uses the following rule to ensure safety. The rule of thumb is that it is usually not safe to approximate array indices, pointers, and control flow conditionals. However, even after excluding these cases to ensure safety, as the results confirm, RFVP still provides significant performance and energy gains because there are still enough performance critical loads that are safe to approximate.

Figure 5 shows code snippets from our application to illustrate how approximating load instructions can lead to safety violations. In Figure 5a, it is not safe to approximate loads from ei, row, d_iS[row] variables that are used as array indices. Approximating such loads may lead to array out-of-bounds accesses and segmentation faults. In Figure 5b, it is unsafe to approximate variable d_Src, which is a pointer. Approximation of this variable may lead to memory safety violations and segmentation faults. In Figure 5c, it is not safe to approximate the ei_new and in2_elem variables because they affect control flow. Approximating such loads may lead to infinite loops or premature termination. In many cases, control flow in the form of if-then-else statement can be if-converted to data flow. Therefore, it might be safe to approximate the loads that affect the if-convertible control flow conditionals. Figure 5d illustrates such a case. Loads for both value and newValue are safe-to-approximate even though they affect the if condition.

3.2. Targeting Performance-Critical Loads

The next step is a profiling pass that identifies the subset of the loads that cause the largest percentage of cache misses. As prior work has shown [13], and our experiments corroborate, only a few load instructions cause the large majority of the total cache misses. Figure 4 illustrates this trend by showing the cumulative distribution function of the LLC cache misses caused by distinct load instructions in the GPU. As Figure 4 shows, in all of our GPU applications except one, six loads cause more than 80% of the misses. We refer to these loads as the performance-critical loads. Clearly, focusing rollback-free value prediction on these loads will provide the opportunity to eliminate the majority of the cache misses. Furthermore, the focus will reduce the predictor size and consequently its overheads. Therefore, this step provides a subset of this list that contains the most performance-critical and safe-to-approximate loads as candidates for approximation. Note that the programmer annotation identify the safe-to-approximate loads and not the profiling.

3.3. Avoiding Significant Quality Degradations

The first two steps provide a small list of safe and performance-critical loads. However, approximating all these loads may lead to significant quality degradation. Therefore, in the last step, we perform a quality profiling pass that identifies the approximable loads that significantly degrade quality. This pass examines the output quality degradation by individually approximating the safe loads. A load is removed from the approximable list if approximating it individually leads to quality degradation higher than a programmer-defined threshold. Furthermore, any approximation technique may prolong convergence for iterative algorithms. We guard against this case by removing safe-to-approximate load instructions which increase run time when approximated.

Finally, the compiler uses a simple heuristic algorithm to statically determine the highest drop rate given a statistical quality requirement and a set of representative inputs. Of the set of representative inputs, half are used for profiling and the rest are used for validation. The algorithm works as follows: (1) select a "moderate" drop rate around 25% as the baseline; (2) run the application with test data to determine the output error at that drop rate; (3) if the quality degradation is too large, decrease the drop rate by some small delta, if the quality degradation is permitted to be higher, increase the drop rate by the delta; (4) repeat steps 2 and 3 until reaching the maximum drop rate that statistically satisfies the quality expectation.

Alternatively, we can determine the drop rate dynamically at run time using techniques such as those described in SAGE [49]. SAGE uses computation sampling and occasional redundant execution on the CPU to dynamically monitor and control approximation. While setting the drop rate dynamically may provide an advantage of more adaptive error control, it also has a disadvantage of some additional overheads. Ultimately, we considered this tradeoff and decided to

[2]We use GPGPU-Sim [7] for profiling the GPU applications.

```
void srad2{
  N = d_c[ei];
  S = d_c[d_iS[row] + d_Nr * col];
  W = d_c[ei];
  E = d_c[row + d_Nr * d_jE[col]];
}
```

(a) A snippet from srad

```
float *d_Src = d_Input  + base;
for(int pos = threadIdx.x;
    pos < N; pos += blockDim.x)
{
    s_data[pos] = d_Src[pos];
}
```

(b) A snippet from fastwalsh

```
while(ei_new < in2_elem){
  row = (ei_new+1)
        % d_common.in2_rows - 1;
  col = (ei_new+1)
        / d_common.in2_rows + 1;
}
```

(c) A snippet from heartwall

```
if (value - newValue < .5f)
{
  result =  newValue;
}
else
  result =  newValue + 1;
```

(d) A snippet from particlefilter

Figure 5: Code examples with different safety violations.

use a static drop rate based on profiling information in our evaluation, but using such a dynamic quality control scheme is a viable alternative.

Either technique results in a statistical guarantee that the output error will be within the bounds set by the programmer. Although these techniques do not strictly guarantee quality for any program input, they provide confidence that the program will satisfy the quality expectation if the inputs are drawn from the same distribution used for profiling. Such statistical guarantees are commensurate with other state-of-the-art techniques in approximate computing [35, 49, 56, 6]. Even dynamic quality control only provides statistical guarantees. Generally, providing formal quality guarantees for approximation techniques across all possible inputs is still an open research problem. Altogether, these steps provide a compilation workflow that focus RFVP on the safe-to-approximate loads with the highest potential–both in terms of performance and effect on the output quality.

4. Value Predictor Design for RFVP

One of the main design challenges for effective rollback-free value prediction is devising a low-overhead fast-learning value predictor. The predictor needs to quickly adapt to the rapidly-changing value patterns in every approximate load instruction. There are several modern *exact* value predictors [20, 44]. We use the two-delta stride predictor [16] due to its low complexity and reasonable accuracy as the base for multi-value prediction. We have also experimented with other value prediction mechanisms such as dfcm [20], last value [31] and stride [54]. Empirically, two-delta provides a good tradeoff between accuracy and complexity. We choose this scheme because it only requires one addition to perform the prediction and a few additions and subtractions for learning. It also requires lower storage overhead than more accurate context-sensitive alternatives [20, 44]. However, this predictor cannot be readily used for multi-value prediction which is required for GPUs. Due to the SIMD execution model in modern GPUs, the predictor needs to generate multiple concurrent predictions for multiple concurrent threads.

Below, we first describe the design of the base predictor, and then devise an architecture that performs full cache line multi-value GPU prediction.

4.1. Base Predictor for RFVP

Figure 6 illustrates the structure[3] of the two-delta predictor [16], which we use as a base design for rollback-free value prediction in GPUs. The predictor consists of a value history

[3]For clarity, Figure 6 does not depict the update logic of the predictor.

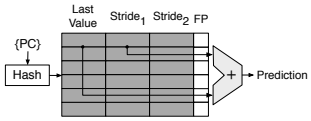


Figure 6: Structure of the base two-delta [16] predictor.

table that tracks the values of the load instructions. The table is indexed by a hash of the approximate load's PC. We use a hash function that is similar to the one used in [20]. Each row in the table stores three values: (1) the last *precise* value, (2) Stride_1, and (3) Stride_2. The last value plus Stride_1 makes up the prediction. When a safe-to-approximate load misses in the cache but is *not* dropped, the predictor updates the last value upon receiving the data from lower level memory. We refer to the value from memory as the current value. Then, it calculates the stride, the difference between the last value and the current value. If the stride is equal to the Stride_2, it stores the stride in Stride_1. Otherwise Stride_1 will not be updated. The predictor always stores the stride in Stride_2. The two-delta predictor only updates the Stride_1, which is the prediction stride, if it observes the same stride twice in a row. This technique lowers the rate of mispredictions. However, for floating point loads, it is unlikely to observe two matching strides. Floating point additions and subtractions are also costly. Furthermore, RFVP is performing *approximate* value predictions for error-resilient applications that can tolerate small deviations in floating point values. Considering these challenges and the approximate nature of the target applications, our two-delta predictor simply outputs the last value for floating point loads. We add a bit to each row of the predictor to indicate whether or not the corresponding load is a floating point instruction.

4.2. Rollback-Free Value Predictor for GPUs

Here we expound the RFVP predictor design for multi-value prediction in GPUs, where SIMD loads read multiple words.

GPU predictor structure. The fundamental challenge in designing the GPU predictor is that a single data request is a SIMD load that must produce values for many concurrent threads. A naive approach to performing value prediction in GPUs is to replicate the single value predictor for each concurrent thread. For example, in a typical modern GPU, there may be as many as 1536 threads in flight during execution. Therefore, the naive predictor would require 1536 two-delta predictors, which of course is impractical. Fortunately, while each SIMD load requires many predicted data elements, adjacent threads operate on data that has significant value similarity. In other words, we expect that the value in a memory

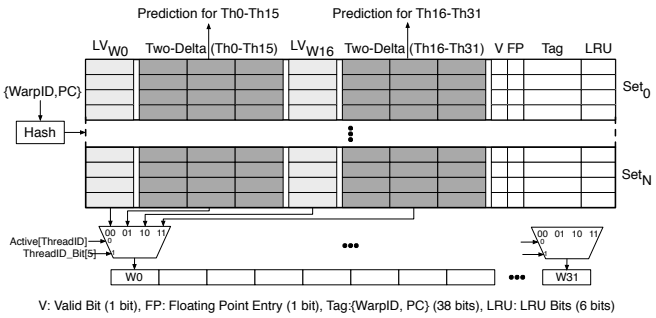


Figure 7: Structure of the multi-value predictor for RFVP in GPUs. The GPU predictor consists of two two-delta and two last value predictors. The GPU predictor is also set-associative to reduce the conflicts between loads from different active warps. It produces predictions for full cache lines.

location accessed by thread N will be similar to the values accessed by threads N-1 and N+1. This insight drives our value predictor design.

In many GPU applications, the adjacent threads in a warp process data elements with some degree of value similarity, e.g. pixels of an image. We also leverage the fact that predictions are only *approximations* and the application can tolerate small errors. We exploit these opportunities and design a predictor that consists of only two parallel specialized two-delta predictors. As Figure 7 shows, the Two-Delta (Th0–Th15) structure generates predictions for threads with ThreadID=0–15. Similarly, the Two-Delta (Th16-Th32) structure generates predictions for threads with ThreadID=16–31. The GPU predictor is indexed by the hash of the WarpID plus the load PC. This combination uniquely identifies the load. We always update the predictor with the value of the active thread with the lowest threadID. The GPU predictor also performs full cache line prediction. Each cache line in our design has 32 4-byte words.

We add a column to each two-delta predictor that tracks the last value of the $word_0$ and $word_{16}$ in the cache line being accessed by the approximate load. When predicting the cache line, all the words that are accessed by the active thread will be filled by the pair of two-delta predictors. However, there might be less than 32 active threads, leaving "gaps" in the predicted cache line. These gaps are filled in with the value of $word_0$ and $word_{16}$. The last value of $word_0$ may fill $words_{0-15}$ and the last value of $word_{16}$ may fill $words_{16-31}$. To reduce the conflicts between loads from different active warps, we make the GPU predictor set associative with LRU replacement policy. As Figure 7 shows, for each row in the predictor, we keep the corresponding load's {WarpID, PC} as the row tag. The load values will only be predicted if their {WarpID, PC} matches the row tag. For measurements, we use a predictor that has 192 entries, is 4-way set associative, and consists of two two-delta predictors and two last value predictors. Section 6 provides a detailed design space exploration for the GPU predictor. Note that the none of the predictors store the full cache line. Instead, the predicted cache line is inserted in the cache. Each predictor only tracks the value of the active thread with the lowest ID.

Table 1: GPU applications, input data, and quality metrics.

	Name	Suite	Domain	Quality Metric	Profiling Set (10 of)	Test Set (10 of)	Approx Loads
GPU Applications	backprop	Rodinia	Machine Learning	Avg Relative Error	A Neural Network with 65,536 Neurons	A Neural Network with 262,144 Neurons	(10, 2)
	fastwalsh	NVIDIA SDK	Signal Processing	Image Diff	128x128-Pixel Color Image	512x512-Pixel Color Image	(2, 1, 4)
	gaussian	NVIDIA SDK	Image Processing	Image Diff	128x128-Pixel Color Image	512x512-Pixel Color Image	5
	heartwall	Rodinia	Medical Imaging	Avg Displacement	A Frame of Ultrasound Image	Five Frames of Ultrasound Images	10
	matrixmul	Mars	Scientific	NRMSE	Two 128x128 Matrices	Two 512x512 Matrices	8
	particle filter	Rodinia	Medical Imaging	Avg Displacement	256x256x10 Cube with 100 Particles	512x512x10 Cube with 2,000 Particles	(2, 3)
	similarity score	Mars	Web Mining	NRMSE	One HTML File	Five HTML Files	8
	s.reduce	Rodinia	Image Processing	NRMSE	128x128-Pixel Color Image	512x512-Pixel Color Image	2
	s.srad2	Rodinia	Image Processing	NRMSE	128x128-Pixel Color Image	512x512-Pixel Color Image	4
	string match	Mars	Web Mining	Missmatch Rate	4 MB File	16 MB File	1

5. Experimental Methodology

We use a diverse set of applications, cycle-accurate simulation, and low-level energy modeling to evaluate RFVP in a modern GPU. This section details our experimental methodology and Section 6 presents the results.

5.1. Experimental Methodology for GPUs

Applications. As Table 1 shows, we use a diverse set of GPU applications from the Rodinia [11], NVIDIA SDK [1], and Mars [22] benchmark suites to evaluate RFVP with the GPU architectures. Columns 1-3 of Table 1 summarize these applications and their domains. The applications are amenable to approximation and represent a wide range of domains including pattern recognition, machine learning, image processing, scientific computing, medical imaging, and web mining. One of the applications, srad takes an inordinately long time to simulate to completion. Therefore, we evaluate the two kernels that dominate srad's runtime separately. These kernels are denoted as s.reduce and s.srad2 in Table 1. We use NVCC 4.2 from the CUDA SDK to compile the applications. Furthermore, we optimize the number of thread blocks and number of threads per block of each kernel for our simulated hardware.

Quality metrics. Column 4 of Table 1 lists each application's quality metric. Each application-specific error metric determines the application's output quality loss as it undergoes RFVP approximation. Using application-specific quality metrics is commensurate with other work on approximation [17, 50, 6, 18, 3]. To measure quality loss, we compare the output from the RFVP-enabled execution to the output with no approximation. For similarityscore, s.reduce, s.rad2 and matrixmul, which generate numerical outputs, we use the normalized root-mean-square error (NRMSE) as the quality metric. The backprop application solves a regression problem and generates a numeric output. The regression error is measured in relative error. Since gaussian and fastwalsh output images, we use the image difference RMSE as the quality metric. The heartwall application finds the inner and outer walls of a heart from 3D images and computes the location of each wall. We measure the quality loss using the average Euclidean distance between the corresponding points of the approximate and pre-

Table 2: GPU microarchitectural parameters.

Processor: 700 MHz, No. Compute Units: 30, SMs: 16, Warp Size: 32, SIMD Width: 8, No. of Threads per Core: 1024, **L1 Data Cache**: 16KB, 128B line, 4-way, LRU; **Shared Memory**: 48KB, 32 banks; **L2 Unified Cache**: 768KB, 128B line, 8-way, LRU; **Memory**: GDDR5, 924 MHz, FR-FCFS, 4 memory channels, **Bandwidth**: 173.25 GB/sec

cise output. We use the same metric for particlefilter, which computes locations of particles in a 3D space. Finally, we use the total mismatch rate for stringmatch.

Profiling and load identification. As Table 1 shows in columns 5 and 6, we use distinct data sets for profiling and final measurements. Doing so avoids biasing our results towards a particular data set. We use smaller data sets for profiling and larger datasets for final measurements. Using larger data sets ensures that the simulations capture all relevant application behavior. The final column of Table 1 lists the number of approximate loads, which are identified in the profiling phase. For some applications, such as backprop, fastwalsh, and particlefilter, we identify approximable loads for each kernel individually. In this case we list the number of loads for each kernel as a tuple in Table 1 (e.g., (2, 1, 4) for fastwalsh).

Cycle-accurate simulations. We use the GPGPU-Sim cycle-accurate simulator version 3.1 [7]. We modified the simulator to include our ISA extensions, value prediction, and all necessary cache and memory logic to support RFVP. We use one of GPGPU-Sim's default configurations that closely models an NVIDIA GTX 480 chipset with Fermi architecture. Table 2 summarizes the microarchitectural parameters of the chipset. To account for random events in the simulations, we run each application 10 times and report the average results. We also run the applications to completion.

Energy modeling and overheads. To measure the energy benefits of RFVP, we use GPUWattch [25], which is integrated with GPGPU-Sim. RFVP comes with overheads including the prediction tables, arithmetic operation, and allocation of the predicted lines in the cache. Our simulator changes enable GPUWattch to account for the caching overheads. We estimate the prediction table read and write energy using CACTI version 6.5 [38]. We extract the overhead of arithmetic operations from McPAT [26]. Our energy evaluations use a 40 nm process node and 700 MHz clock frequency. Furthermore, we have synthesized the LFSR and the hash function and incorporated the energy overheads. The default RFVP prediction table size is 14 KB per SM and the GPU consists of 16 SMs. The GPU off-chip memory bandwidth is 173.25 GB/sec.

6. Experimental Results

This section empirically evaluates the tradeoffs between performance, energy, and quality when RFVP is employed in a modern GPU. This section also includes a Pareto analysis of the RFVP predictor design.

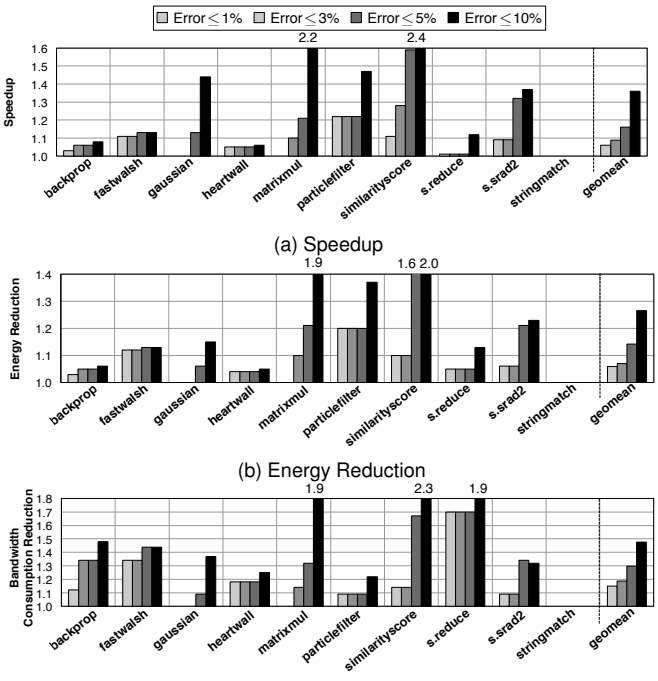


Figure 8: GPU (a) performance improvement, (b) energy reduction, and (c) bandwidth consumption reduction for 1%, 3%, 5%, and 10% quality degradation.

6.1. GPU Measurements

6.1.1. Performance, Energy, Bandwidth, and Quality Figure 8a shows the speedup with RFVP for 1%, 3%, 5%, and 10% quality degradation. We have explored this tradeoff by setting different drop rates, which is RFVP's knob for quality control. The baseline is the default architecture without RFVP. Figures 8b and 8c illustrate the energy reduction and the reduction in off-chip bandwidth consumption, respectively.

As Figures 8a and 8b show, RFVP yields, on average, 36% speedup and 27% energy reduction with 10% quality loss. The speedup is as high as 2.2× for matrixmul and 2.4× for similarityscore with 10% quality loss. The maximum energy reduction is 2.0× for similarityscore. RFVP yields these benefits despite approximating less than 10 static performance-critical load instruction per kernel. The results show the effectiveness of our profiling stage in focusing approximation where it is most beneficial.

With 5% quality loss, the average performance and energy gains are 16% and 14%, respectively. These results demonstrate RFVP's ability to navigate the tradeoff between quality and performance-energy based on the user requirements.

Even with a small quality degradation of 1%, RFVP yields significant speedup and energy reduction in several cases, including fastwalsh, particlefilter, similarityscore, s.srad2. In particular, the benefits are as high as 22% speedup and 20% energy reduction for particlefilter with 1% quality loss.

Comparing Figures 8a, 8b, and 8c shows that the benefits strongly correlate with the reduction in bandwidth consumption. This strong correlation suggests that RFVP is able to significantly improve both GPU performance and energy

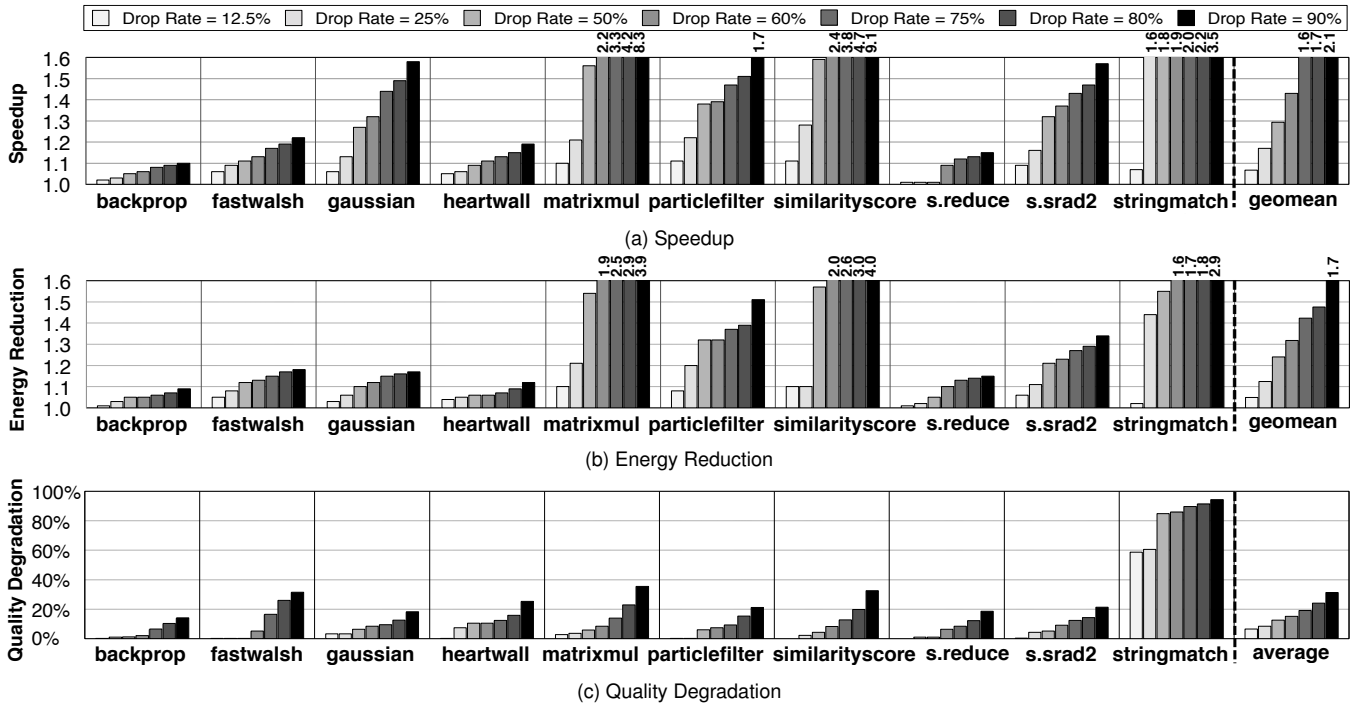


Figure 9: Exploring (a) speedup, (b) energy reduction, and (c) quality trade-offs with different drop rates.

consumption by predicting load values and dropping memory access requests. The applications for which the bandwidth consumption is reduced the most (matrixmul, similarityscore), are usually the ones that benefit the most from RFVP. One notable exception is s.reduce. Figure 8c shows that RFVP reduces this application's bandwidth consumption significantly (up to 90%), yet the performance and energy benefits are relatively modest (about 10%). However, Figure 1 illustrates that s.reduce yields less than 40% performance benefit even with unlimited memory bandwidth. Therefore, the benefits from RFVP are predictably limited even with significant bandwidth reduction. This case shows that the applications' sensitivity to off-chip communication bandwidth is an important factor in RFVP's ability to achieve performance and energy benefits. Also, Figure 8 shows no benefits for stringmatch with less than 10% quality degradation. This case is an interesting outlier which we discuss in greater detail in the next subsection. To better understand the sources of the benefits, we perform an experiment in which RFVP fills the L1 cache with predicted values, but does *not* drop the corresponding memory accesses. In this scenario, RFVP yields only 2% performance improvement and *increases* energy consumption by 2% on average for these applications. These results further suggest that the source of RFVP's benefits come primarily from reduced bandwidth consumption, which is a large bottleneck in GPUs that hide latency with many-thread execution. We study the effects of RFVP on single-core CPUs that are more latency sensitive in Section 7.

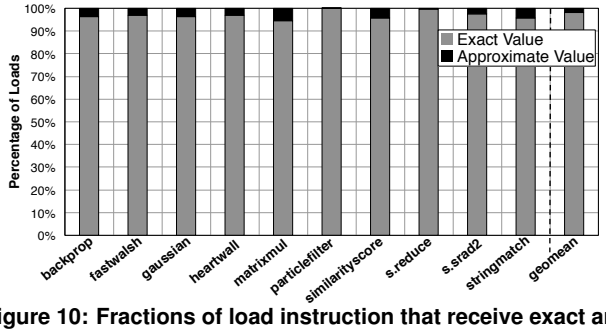


Figure 10: Fractions of load instruction that receive exact and approximate values during execution. The drop rate is 25%.

All applications but one benefit considerably from RFVP due to reduced off-chip communication. Particularly, the energy benefits are due to reduced runtime and fewer costly data fetches from off-chip memory. Overall, these results confirm the effectiveness of rollback-free value prediction in mitigating the bandwidth bottleneck for a diverse set of GPU applications.

6.1.2. Sources of Quality Degradation To determine the effectiveness of our value prediction, we measure the portion of load operations which ultimately return imprecise values. Figure 10 shows the result of these measurements. The results show that on average only 2% (max 5.4%) of all dynamic load instructions return imprecise values. Thus, the large majority of all dynamic loads return correct values. The prediction accuracy is relatively low, yet commensurate with prior works on value prediction [16, 20, 9]. However, our profiling phase focuses approximation on the safe-to-approximate loads that do not significantly degrade output quality. These observations help justify the low quality degradation shown in Figure 9c.

6.1.3. Quality Tradeoffs with Drop Rate Drop rate is RFVP's knob for navigating the quality tradeoffs. It dictates what percentage of the missed approximate loads to predict and drop. For example, with 12.5% rate, RFVP drops one out of eight approximate load misses. We examine the effect of this knob on performance, energy, and quality by sweeping the drop rate from 12.5% to 90%. Figure 9 illustrates the effect of drop rate on speedup (Figure 9a), energy reduction (Figure 9b), and quality degradation (Figure 9c).

As the drop rate increases, so do the performance and energy benefits. However, the benefits come with some cost in output quality. On average, speedup ranges from 1.07× with 12.5% drop rate, to as much as 2.1× with 90% drop rate. Correspondingly, the average energy reduction ranges from 1.05× to 1.7× and the quality degradation ranges from 6.5% to 31%.

Figure 9c shows that in all but one case, quality degradation increases slowly and steadily as the drop rate increases. The clear exception is stringmatch. It searches a file with a large number of strings to find the lines that contain a search word. This application's input data set only contains English words with very low value locality. Furthermore, the application output is the indices of the matching lines, which provides a very low margin for error. Either the index is correctly identified or the output is wrong. The quality metric is the percentage of the correctly matched lines. During search, even if a single character is incorrect, the likelihood of matching the words and identifying the correct lines is low.

Even though stringmatch shows 61% speedup and 44% energy reduction with 25% drop rate, its quality loss of 60% is not acceptable. In fact, stringmatch is an example of an application that cannot benefit from RFVP due to low error tolerance.

As Figure 9 shows, each application tolerates the effects of RFVP approximation differently. For some applications, such as gaussian and fastwalsh, as the rate of approximation (drop rate) increases, speedup, energy reduction and quality loss gradually increase. In other applications such as matrixmul and similarityscore, the performance and energy benefits increase sharply while the quality degradation increases gradually. For example in similarityscore, increasing the drop rate from 25% to 50% yields a jump in speedup (from 28% to 59%) and energy reduction (from 10% to 57%), while quality loss only rises 2%.

Those applications, which experience a jump in benefits, are usually the ones that show the most sensitivity to the available off-chip communication bandwidth (see Figure 1).

6.1.4. Design Space Exploration and Pareto Analysis The two main design parameters of the GPU predictor are the number of parallel predictors and the number of entries in each predictor. We vary these two parameters to explore the design space of the GPU predictor and perform a Pareto analysis to find the optimal configuration. Figure 11 shows the result of this design space exploration. The x-axis captures the complexity of the predictor in terms of size in KBytes. The y-axis is the Normalized Energy×Normalized Delay×Error

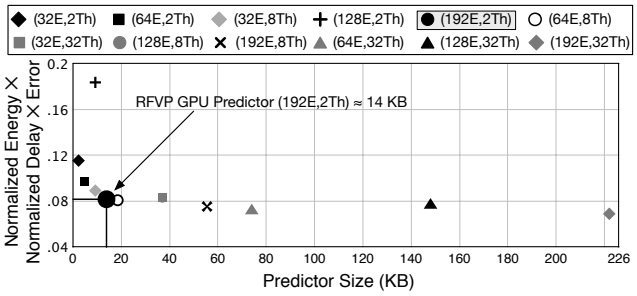


Figure 11: GPU predictor design space exploration and Pareto analysis. A predictor configuration of (192E,2Th), which is our default configuration, is the most Pareto optimal design point. In this graph, lower and left is better. The normalization baseline is the execution without RFVP. The (*x*E,*y*Th) represents the configuration with *y* parallel predictors each with *x* entries. All the predictors are 4-way set associative.

across all the GPU applications. The normalization baseline is the execution without RFVP. This product simultaneously captures the three metrics of interest, namely performance, energy, and quality. The optimal predictor minimizes size (left on the x-axis), energy dissipation, execution delay, and error (lower on the y-axis). In Figure 11, (*x*E,*y*Th) represents a configuration with *y* parallel predictors each with *x* entries. All the predictors are 4-way set associative.

In Figure 11, the knee of the curve is the most cost-effective point. This Pareto-optimal design is the (192E,2Th) configuration, which requires 14 KB of storage, and is our default configuration.

This design space exploration shows that the number of entries in the prediction table has a clear effect on the potential benefits. Increasing the number of entries from 32 to 192 provides 1.4× improvement in Normalized Energy×Normalized Delay×Error. A higher number of entries lowers the chance of destructive aliasing in the prediction table that leads to eviction of value history from the prediction tables. However, adding more parallel predictors beyond a certain point does not provide any significant benefit and wastes area. With fewer predictors, RFVP relies more on the value locality across the threads, which is a common case in GPU applications.

Exploiting value locality reduces RFVP's area overhead without significantly degrading output quality.

Finally, we compare the benefits of RFVP with the benefits that can be achieved by simply enlarging the caches by the RFVP predictor size. We found that, for the studied applications, the increased L1 size in each SM results in 4% performance improvement and 1% energy savings on average. The increased L2 size yields only 2% performance improvement and 1% energy savings on average. RFVP provides significantly higher benefits with the same overhead by trading output quality for performance and energy gains.

7. CPU Experiments

To understand the effectiveness of of RFVP in a system where latency is the primary concern, we investigate the application

Table 3: CPU applications, input data, and quality metrics.

	Name	Suite	Domain	Quality Metric	Profiling Set	Test Set	Approx Loads
CPU Applications	bwaves	CFP2006	Scientific	NRMSE	Test Set	Reference Set	26
	cactusADM	CFP2006	Scientific	NRMSE	Test Set	Reference Set	28
	fma3D	CFP2000	Scientific	NRMSE	Test Set	Reference Set	27
	gemsFDTD	CFP2006	Scientific	NRMSE	Test Set	Reference Set	23
	soplex	CFP2006	Optimization	NRMSE	Test Set	Reference Set	21
	swim	CFP2000	Scientific	NRMSE	Test Set	Reference Set	23

of RFVP in a single core CPU system.

7.1. Methodology

Applications. As Table 3 shows, we evaluate RFVP for CPUs using an approximable subset of SPEC CFP2000/2006. The applications come from the domains of scientific computing and optimization. As the work in [55] discusses, the CFP2000/2006 benchmarks have some natural tolerance to approximation. When these floating point applications discretize continuous-time inputs, the resulting data is naturally imprecise. We compile the benchmarks using gcc version 4.6.

Quality metrics. As discussed below, our subset of the SPEC applications produce numerical outputs. Therefore, we use NRMSE to measure the quality loss. For swim, the output consist of all diagonal elements of the velocity fields of a fluid model. In fma3d, the outputs are position and velocity values for 3D solids. In bwaves, the outputs define the behavior of blast waves in 3D viscous flow. The cactusADM benchmark outputs a set of coordinate values for space-time in response to matter content. The soplex benchmark solves a linear programming problem and outputs the solution. Finally, GemsFDTD outputs the radar cross section of a perfectly conducting object using the Maxwell equations.

Profiling and load identification. As in the GPU evaluations, we use smaller (SPEC test) data sets for profiling and larger (SPEC reference) data sets for the performance, energy, and quality evaluations. We use Valgrind with the Cachegrind tool [42] for both the profiling and final quality of result evaluation. We modify Cachegrind to support rollback-free value prediction. Valgrind is fast enough to both perform the profiling and run our applications until completion with reference data sets. Thus, we use Valgrind plus Cachegrind for profiling, approximate loads selection, and final quality assessments.

Cycle-accurate simulations. We implement RFVP in the MARSSx86 cycle-accurate simulator [43]. The baseline memory system includes a 32 KB L1 cache, a 2 MB LLC, and external memory with 200-cycle access latency. In modern processors, the LLC size is often 2 MB $\times$ number of cores. Thus, we use a 2 MB LLC for our single core experiments. Furthermore, the simulations accurately model port and interconnect contention at all levels of the memory hierarchy. The core model follows the Intel Nehalem microarchitecture [37]. Because simulation until completion is impractical for SPEC applications with reference data sets, we use Simpoint [21] to identify the representative application phases. We perform all the measurements for the same amount of work in the application using markers in the code. Table 4 summarizes the microarchitectural parameters for the CPU simulations. As in the GPU evaluations, we run each application 10 times and report the average to account for random events in simulation.

Table 4: CPU microarchitectural parameters.

Processor: Fetch/Issue Width: 4/5, INT ALUs/FPUs: 6/6, Load/Store Queue: 48-entry/32-entry, ROB Entries: 128, Issue Queue Entries: 36, INT/FP Physical Registers: 256/256, Branch Predictor: Tournament 48 KB, BTB Sets/Ways: 1024/4, RAS Entries: 64, Dependence Predictor: 4096-entry Bloom Filter, ITLB/DTLB Entries: 128/256; **L1:** 32 KB I$, 32 KB D$, 64B line, 8-Way, Latency: 2 cycles; **L2:** 2 MB, 64B line, 8-Way, Latency: 20 cycles; **Memory Latency:** 200 cycles

Table 5: CPU L2 MPKI comparison with and without RFVP.

	bwaves	cactusADM	fma3d	gemsFDTD	soplex	swim
Baseline	11.6	5	1.5	23.1	26.5	3.9
RFVP	2.2	3.9	0.6	10.3	21.4	2.4

Energy modeling and overheads. We use McPAT [26] and CACTI [38] to measure energy benefits while considering all the overheads associated with RFVP. The caching overheads are incorporated into the statistics that Marssx86 produces for McPAT. As in the GPU case, we estimate the prediction table overhead using CACTI version 6.5, and extract the arithmetic operations overhead from McPAT. The energy evaluations use a 45 nm process, 0.9 Vdd and 3.0 GHz core clock frequency.

7.2. Results

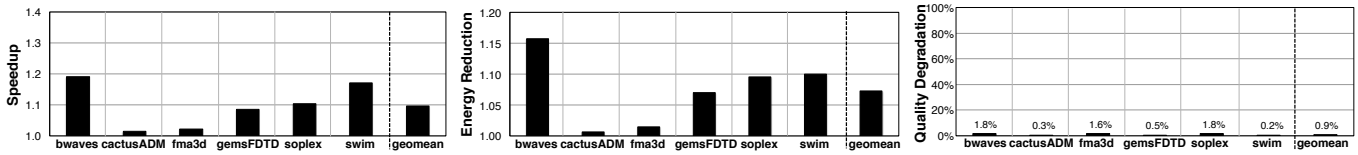
Figure 12 shows the speedup, energy reduction, and quality degradation with RFVP. The baseline is the execution with no approximation. In this case, RFVP aims to mitigate the long memory access latencies. Thus, RFVP predicts all missing approximate load requests but does not drop any of them. We experimented with dropping requests in the CPU experiments. However, there was no significant benefit since these single-threaded CPU workloads are not sensitive to the off-chip communication bandwidth.

As Figure 12 shows, RFVP provides 8% average speedup and 6% energy reduction with a single-core CPU. The average quality loss is 0.9%.

While the CPU benefits are lower than the GPU benefits, the CPU quality degradations are also comparatively low. The GPU applications in our workload pool are more amenable to approximation than the CPU applications. That is, a larger fraction of the performance-critical loads are safe to approximate in GPU workloads. Nevertheless, Figure 12 shows that bwaves gains 19% speedup and 16% energy reduction with only 1.8% quality degradation.

To better understand the CPU performance and energy benefits, we examine MPKI reduction in the L2 cache, and present the results in Table 5. RFVP reduces MPKI by enabling the core to continue without stalling for memory to supply data. Usually, a larger reduction in MPKI leads to larger benefits. For example, for bwaves the L2 MPKI drops from 11.6 to 2.2, leading to 19% speedup and 16% energy reduction.

To understand the low quality degradations of the CPU applications with RFVP, we also study the distribution the fraction of the load values that receive approximate and precise values during execution. The trends are similar to the ones that we observed for the GPU experiment (see Figure 10). In



(a) Speedup (b) Energy Reduction (c) Quality Degradation

Figure 12: Employing RFVP in a single-core CPU. (a) speedup, (b) energy reduction, and (c) quality degradation.

the CPU case, on average only 1.5% of all the dynamic loads receive imprecise values.

> Due to the overall low rate at which load instructions return imprecise data to the CPU, the applications experience low quality degradation in the final output. In fact, RFVP in the CPU case achieves performance and energy gains that are one order of magnitude greater than the quality loss.

The value prediction accuracy in the CPU case is on par with prior work [16, 20, 9] and the GPU case. Once again, the profiling phase focuses approximation on the safe-to-approximate loads that do not significantly degrade the output quality. These results show that RFVP effectively mitigates the long memory access latency with a low degradation in quality.

8. Related Work

General-purpose approximate computing. Recent work explored a variety of approximation techniques. However, approximation techniques that tackle memory subsystem performance bottlenecks are lacking. This paper defines a new technique that mitigates the memory subsystem bottlenecks of long access latency and limited off-chip bandwidth.

The existing techniques include (a) approximate storage designs [33, 51] that trades quality of data for reduced energy [33] and larger capacity [51], (b) voltage overscaling [17, 10, 40, 23, 24], (c) loop perforation [56, 36, 46], (d) loop early termination [6], (e) computation substitution [49, 6, 4, 53], (f) memoization [2, 48, 5], (g) limited fault recovery [15, 27, 28, 14, 36, 19, 61], (h) precision scaling [50, 59], and (i) approximate circuit synthesis [45, 60, 34, 41, 29, 30]. Most of these techniques (1) operate at the coarse granularity of a loop body or a functional call; (2) are agnostic to and unaware of micro-architectural events; (3) and are explicitly invoked by the code. In contrast, rollback-free value prediction (1) operates at the fine-granularity of a single load instruction and (2) is triggered by microarchitectural events, (3) without direct and explicit runtime software invocation. In this context, we discuss the most related work.

EnerJ [50] is a language for approximate computing. Its corresponding architecture, Truffle [17], leverages only voltage overscaling, floating point bitwidth reduction, and reduced DRAM refresh. We borrow the programming constructs and ISA augmentation approach from EnerJ and Truffle, respectively. However, we define our own novel microarchitectural approximation technique. EnerJ and Truffle reduce energy consumption in CPUs, while we improve both performance and energy efficiency in GPUs as well as CPUs. The work in [33] and [51] design approximate DRAM and Flash storage blocks. Flikker [33] reduced the DRAM refresh rate when approximate data is stored in main memory. The work in [5] uses hardware memoization to reduce redundant computation in GPUs. However, while this work eliminates execution within the SMs, it still requires data inputs to be read from memory. Some bandwidth savings may arise by eliminating these executions, but our work fundamentally differs in that it attacks the bandwidth wall directly by completely eliminating memory traffic. The work in [51] uses faulty flash blocks for storing approximate data to prolong its lifetime. This work also aims to improve the density and access latency of flash memory using multi-level cells with small error margins. The technique in [53] exploits approximation to mitigate branch and memory divergence in GPUs. In case of branch divergence, authors force all the threads to execute the most popular path. In case of memory divergence, they force all the threads to access the most commonly demanded memory block. Their work is agnostic to cache misses and does not leverage value prediction nor it drops memory requests. In contrast, our novel approximation technique predicts the value of the approximate loads that miss in the cache without ever recovering from the misprediction. Further, we reduce the bandwidth demand and memory contention by dropping a fraction of the approximate load requests after predicting their value. Our approach can be potentially combined with many of the prior works on approximation since it exclusively focuses on mitigating off-chip communication limitations.

Value prediction. RFVP takes inspiration from prior work that explores exact value prediction [54, 32, 62, 16, 57, 20, 44]. However, our work fundamentally differs from these techniques because it does not check for mispredictions and does not rollback from them. Furthermore, we drop a fraction of the load requests to reduce off-chip memory traffic. Among these techniques, Zhou et al. [62] use value prediction to speculatively prefetch cache misses that are normally serviced sequentially. They used value prediction to break dependence chains where one missing load's address depends on the previous missing load's value. However, they do not allow the speculative state to contaminate the microarchitectural state of the processor or the memory. Since their technique only initiates prefetches, they do not need to recover from value mispredictions. Our technique, however, is not used for prefetch requests. Instead, the predictor directly feeds the predicted value to the processor as an approximation of the load value.

Recently, in a concurrent submission, San Miguel et al. [52], proposed a technique which utilizes approximate

load handling via value prediction without checks for misprediction to address the memory latency bottleneck. Concurrently to [52], Thwaites et al. [58] proposed in a short paper a similar idea: predict the values of safe-to-approximate loads to reduce average memory access time in latency-critical applications. However, these works only use approximate load handing to solve latency bottlenecks in CPU systems. Our work differs from both works in the following ways: (1) we evaluate our techniques in a GPU environment, thus showing that RFVP is an effective tool for mitigating both latency *and* bandwidth constraints, (2) we drop a portion of missing load requests, thus addressing the fundamentally different bottleneck of limited off-chip bandwidth, and (3) we utilize the inherent value similarity of accesses across adjacent threads to develop a multi-value predictor capable of producing values for many simultaneously-missing loads with low overhead.

9. Conclusions

This paper introduces Rollback-Free Value Prediction (RFVP) and demonstrates its effectiveness in tackling two major memory system bottlenecks–limited off-chip bandwidth and long memory access latency. RFVP predicts the values of safe-to-approximate loads only when they miss in the cache with no checks or recovery from misspeculations. We utilize programmer annotations to *guarantee* safety, while a profile-directed compilation workflow applies approximation only to the loads which provide the most performance improvement with the least quality degradation. We extend and use a light-weight and fast-learning prediction mechanism, which is capable of adapting to rapidly-changing value patterns between individual loads with low hardware overhead.

RFVP uses these predicted values both to hide the memory latency and ease bandwidth limitations. The drop rate becomes a knob that controls the tradeoff between quality of results and performance/energy gains. Our extensive evaluation shows that RFVP for GPUs yields average 40% performance increase and 31% energy reduction with average 8.8% quality loss. Thus, RFVP achieves significant energy savings and performance improvement with limited loss of quality. The results support the effectiveness of RFVP in mitigating the two memory subsystem bottlenecks.

References

[1] NVIDIA corporation. NVIDIA CUDA SDK code samples. [Online]. Available: https://developer.nvidia.com/gpu-computing-sdk.

[2] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Trans. Comput.*, vol. 54, no. 7, 2005.

[3] R. S. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger, "General-purpose code acceleration with limited-precision analog computation," in *ISCA*, 2014.

[4] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: a language and compiler for algorithmic choice," in *PLDI*, 2009.

[5] J.-M. Arnau, J.-M. Parcerisa, and P. Xekalakis, "Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 529–540. Available: http://dl.acm.org/citation.cfm?id=2665671.2665748

[6] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *PLDI*, 2010.

[7] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *ISPASS*, 2009.

[8] M. Carbin, S. Misailovic, and M. C. Rinard, "Verifying quantitative reliability for programs that execute on unreliable hardware," in *OOPSLA*, 2013.

[9] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau, "Cava: Using checkpoint-assisted value prediction to hide l2 misses," *ACM Transactions on Architecture and Code Optimization*, vol. 3, no. 2, 2006.

[10] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee, "Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology," in *DATE*, 2006.

[11] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[12] E. S. Chung, P. A. Milder, J. C. Hoe, and K. Mai, "Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs?" in *MICRO*, 2010.

[13] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: Long-range prefetching of delinquent loads," in *ISCA*, 2001.

[14] M. de Kruijf and K. Sankaralingam, "Exploring the synergy of emerging workloads and silicon reliability trends," in *SELSE*, 2009.

[15] M. de Kruijf, S. Nomura, and K. Sankaralingam, "Relax: An architectural framework for software recovery of hardware faults," in *ISCA*, 2010.

[16] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, vol. 37, no. 4, 1993.

[17] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *ASPLOS*, 2012.

[18] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *MICRO*, 2012.

[19] Y. Fang, H. Li, and X. Li, "A fault criticality evaluation framework of digital systems for error tolerant video applications," in *ATS*, 2011.

[20] B. Goeman, H. Vandierendonck, and K. De Bosschere, "Differential fcm: Increasing value prediction accuracy by improving table usage efficiency," in *HPCA*, 2001.

[21] G. Hamerly, E. Perelman, and B. Calder, "How to use simpoint to pick simulation points," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 4, 2004.

[22] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang, "Mars: a mapreduce framework on graphics processors," in *PACT*, 2008.

[23] R. Hegde and N. R. Shanbhag, "Energy-efficient signal processing via algorithmic noise-tolerance," in *ISLPED*, 1999.

[24] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra, "ERSA: Error resilient system architecture for probabilistic applications," in *DATE*, 2010.

[25] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi, "Gpuwattch: Enabling energy optimizations in gpgpus," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ACM, 2013, pp. 487–498.

[26] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO*, 2009.

[27] X. Li and D. Yeung, "Application-level correctness and its impact on fault tolerance," in *HPCA*, 2007.

[28] X. Li and D. Yeung, "Exploiting application-level correctness for low-cost fault tolerance," *J. Instruction-Level Parallelism*, 2008.

[29] A. Lingamneni, C. Enz, K. Palem, and C. Piguet, "Synthesizing parsimonious inexact circuits through probabilistic design techniques," *ACM Trans. Embed. Comput. Syst.*, vol. 12, no. 2s, 2013.

[30] A. Lingamneni, K. K. Muntimadugu, C. Enz, R. M. Karp, K. V. Palem, and C. Piguet, "Algorithmic methodologies for ultra-efficient inexact architectures for sustaining technology scaling," in *CF*, 2012.

[31] M. H. Lipasti and J. P. Shen, "Exceeding the dataflow limit via value prediction," in *MICRO*, 1996.

[32] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *ASPLOS*, 1996.

[33] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: Saving refresh-power in mobile devices through critical data partitioning," in *ASPLOS*, 2011.

[34] J. Miao, A. Gerstlauer, and M. Orshansky, "Approximate logic synthesis under general error magnitude and frequency constraints," in *ICCAD*, 2013.

[35] S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard, "Chisel: reliability-and accuracy-aware optimization of approximate computational kernels," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. ACM, 2014, pp. 309–328.

[36] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard, "Quality of service profiling," in *ICSE*, 2010.

[37] D. Molka, D. Hackenberg, R. Schone, and M. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *PACT*, 2009.

[38] N. Muralimanohar, R. Balasubramonian, and N. Jouppi, "Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0," in *MICRO*, 2007.

[39] M. Murase, "Linear feedback shift register," 1992, US Patent.

[40] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones, "Scalable stochastic processors," in *DATE*, 2010.

[41] K. Nepal, Y. Li, R. I. Bahar, and S. Reda, "ABACUS: A technique for automated behavioral synthesis of approximate computing circuits," in *DATE*, 2014.

[42] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *PLDI*, 2007.

[43] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A full system simulator for x86 CPUs," in *DAC*, 2011.

[44] A. Perais and A. Seznec, "Practical Data Value Speculation for Future High-end Processors," in *HPCA*, 2014.

[45] A. Ranjan, A. Raha, S. Venkataramani, K. Roy, and A. Raghunathan, "Aslan: Synthesis of approximate sequential circuits," in *DATE*, 2014.

[46] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou, "Patterns and statistical analysis for understanding reduced resource computing," in *Onward!*, 2010.

[47] B. M. Rogers, A. Krishna, G. B. Bell, K. Vu, X. Jiang, and Y. Solihin, "Scaling the Bandwidth Wall: Challenges in and Avenues for CMP scaling," in *ISCA*, 2009.

[48] M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke, "Paraprox: pattern-based approximation for data parallel applications," in *ASPLOS*, 2014.

[49] M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke, "Sage: self-tuning approximation for graphics engines," in *MICRO*, 2013.

[50] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: Approximate data types for safe and general low-power computation," in *PLDI*, 2011.

[51] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *MICRO*, 2013.

[52] J. San Miguel, M. Badr, and N. E. Jerger, "Load value approximation," in *MICRO*, 2014.

[53] J. Sartori and R. Kumar, "Branch and data herding: Reducing control and memory divergence for error-tolerant gpu applications," *Multimedia, IEEE Transactions on*, vol. 15, no. 2, 2013.

[54] Y. Sazeides and J. E. Smith, "The predictability of data values," in *MICRO*, 1997.

[55] S. Sethumadhavan, R. Roberts, and Y. Tsividis, "A case for hybrid discrete-continuous architectures," *Computer Architecture Letters*, vol. 11, no. 1, pp. 1–4, Jan 2012.

[56] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *FSE*, 2011.

[57] R. Thomas and M. Franklin, "Using dataflow based context for accurate value prediction," in *PACT*, 2001.

[58] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry, "Rollback-free value prediction with approximate loads," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*, 2014, pp. 493–494.

[59] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," in *MICRO*, 2013.

[60] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "Salsa: Systematic logic synthesis of approximate circuits," in *DAC*, 2012.

[61] V. Wong and M. Horowitz, "Soft error resilience of probabilistic inference applications," in *SELSE*, 2006.

[62] H. Zhou and T. M. Conte, "Enhancing memory-level parallelism via recovery-free value prediction," *IEEE Trans. Comput.*, vol. 54, 2005.