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

AMIR YAZDANBAKHSH, Georgia Institute of Technology
GENNADY PEKHIMENKO, Carnegie Mellon University
BRADLEY THWAITES and HADI ESMAEILZADEH, Georgia Institute of Technology
ONUR MUTLU and TODD C. MOWRY, Carnegie Mellon University

This article 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 does not check for or recover 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 a fraction of load requests that miss in the cache after predicting their values. Dropping requests reduces memory bandwidth contention by removing them from the system. The drop rate is a knob to control the trade-off between performance/energy efficiency and output quality. Our extensive evaluations show that RFVP, when used in GPUs, yields significant performance improvement and energy reduction for a wide range of quality-loss levels. We also evaluate RFVP's latency benefits for a single core CPU. The results show performance improvement and energy reduction for a wide variety of applications with less than 1% loss in quality.

Categories and Subject Descriptors: C.1.2 [**Computer Systems Organization**]: SIMD

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Load value approximation, GPUs, value prediction, memory latency, memory bandwidth

ACM Reference Format:
Amir Yazdanbakhsh, Gennady Pekhimenko, Bradley Thwaites, Hadi Esmaeilzadeh, Onur Mutlu, and Todd C. Mowry. 2016. RFVP: Rollback-free value prediction with safe-to-approximate loads. ACM Trans. Archit. Code Optim. 12, 4, Article 62 (January 2016), 26 pages.
DOI: http://dx.doi.org/10.1145/2836168

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 large-scale data parallelism to hide main-memory latency. However, this solution suffers from a

Authors' addresses: A. Yazdanbakhsh, 500 Northside Cir. NW, Apt. Q12, Atlanta, GA, 30309; email: a.yazdanbakhsh@gatech.edu; G. Pekhimenko, 1264 Waterdown Road, Burlington, ON, Canada, L7P4Z8; email: gpekhimenko@gmail.com; B. Thwaites, 809 south Lamar apartment #415 Austin Texas 78704; email: bthwaites@gatech.edu; H. Esmaeilzadeh, 266 Ferst Drive, KACB 2336, Atlanta, GA 30332-0765; email: hadi@cc.gatech.edu; O. Mutlu, 7135 Roycrest PI., Pittsburgh, PA 15208; email: omutlu@gmail.com; T. C. Mowry, Gates/Hillman Center 9113, 5000 Forbes Avenue Pittsburgh, PA 15213; email: tcm@cs.cmu.edu.

ACM 1544-3566/2016/01-ART62 $15.00
DOI: http://dx.doi.org/10.1145/2836168

fundamental bottleneck: limited off-chip memory bandwidth to supply data to processing units. In fact, memory bandwidth is predicted to be one of the main performance-limiting factors in accelerator-rich architectures as technology scales [Chung et al. 2010]. This problem is known as the bandwidth wall [Rogers et al. 2009]. Fortunately, there is an opportunity to leverage the inherent error resiliency of many emerging applications to tackle the memory and bandwidth problems. This article exploits this opportunity and introduces an approximation technique to mitigate these memory subsystem bottlenecks.

Large classes of emerging applications—such as web search, data analytics, machine learning, cyber-physical systems, augmented reality, and computer vision—can tolerate error in large parts of their execution, hence, the growing interest in developing general-purpose approximation techniques. These techniques can tolerate error in computation and trade *Quality of Result* for gains in performance, energy, storage capacity, and hardware cost [Yazdanbakhsh et al. 2015; Mahajan et al. 2015; Amant et al. 2014; Luo et al. 2014; Sidiroglou-Douskos et al. 2011]. These techniques include (a) voltage overscaling [Esmaeilzadeh et al. 2012a; Chakrapani et al. 2006], (b) loop perforation [Sidiroglou-Douskos et al. 2011], (c) loop early termination [Baek and Chilimbi 2010], (d) computation substitution [Amant et al. 2014; Samadi et al. 2013; Esmaeilzadeh et al. 2012b], (e) memoization [Samadi et al. 2014; Arnau et al. 2014; Alvarez et al. 2005], (f) limited fault recovery [de Kruijf et al. 2010; Li and Yeung 2007], and (g) approximate data storage [Luo et al. 2014; Sampson et al. 2013; Liu et al. 2011]. However, there is a lack of approximation techniques that address the key memory system performance bottlenecks of long-access latency and limited off-chip memory bandwidth.

To mitigate these memory subsystem bottlenecks, this article 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 trade-off 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 [Perais and Seznec 2014; Goeman et al. 2001; Sazeides and Smith 1997; Lipasti et al. 1996; Eickemeyer and Vassiliadis 1993] that can mitigate the memory wall. However, it fundamentally differs from 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.

In our initial work [Thwaites et al. 2014], we introduced the RFVP technique for CPUs to lower the effective memory-access latency. We also discussed the idea of dropping a fraction of memory requests [Thwaites et al. 2014]. Our results show that dropping memory requests in CPUs is not effective. Later, in a concurrent effort, San Miguel et al. [2014] proposed a technique that uses value prediction without checks for misprediction to address the memory latency bottleneck in CPU processors. They also studied the effect of dropping memory requests in CPUs and how it affects the accuracy

of the predictor, and corroborated our reported results [Thwaites et al. 2014]. This article explores the following directions that differ from our initial work [Thwaites et al. 2014] and the concurrent work [San Miguel et al. 2014]: (1) we specialize our techniques for GPU processors, targeting the bandwidth bottleneck rather than latency, showing that RFVP is an effective approach for mitigating both latency and bandwidth problems; (2) we utilize the value similarity of accesses across adjacent threads in many GPU applications to develop a low-overhead, multivalue predictor capable of producing values for many simultaneously missing loads as they execute lock-step in GPU cores; and (3) we drop a portion of missing load requests to address the limited off-chip bandwidth bottleneck.

This article 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 approximate value prediction mechanisms. The core idea in RFVP is to drop a fraction of cache-missing loads after predicting their values. Therefore, RFVP mitigates the memory bandwidth bottleneck and reduces the effective memory latency.
(2) We propose a new multivalue prediction mechanism for SIMD load instructions in GPUs. These SIMD load instructions request multiple values in one access. To minimize the overhead of the multivalue predictor, we exploit the insight that there is significant value similarity across accesses in the adjacent threads (e.g., due to existing similarity in adjacent pixels in an image). Such value similarity has been demonstrated in recent works [Samadi et al. 2014; Arnau et al. 2014]. For our multivalue predictor in RFVP, we use the two-delta value predictor [Eickemeyer and Vassiliadis 1993]. To find the best design parameters for our proposed predictor, we perform a Pareto-optimality analysis and explore the design space of our predictor, applying 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, 36% speedup, 27% energy reduction, and 48% reduction in off-chip memory bandwidth consumption, with an average 8.8% quality loss. With less than 10% quality loss, the benefits reach a maximum of 2.4× speedup, 2.0× energy reduction, and 2.3× off-chip memory bandwidth consumption 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, 9.7% speedup and 6.2% energy reduction, with 0.9% average quality loss.

2. ARCHITECTURE DESIGN FOR RFVP

2.1. Rollback-Free Value Prediction

Motivation. GPU architectures exploit large-scale 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 [Vijaykumar et al. 2015; Pekhimenko et al. 2015; Keckler et al. 2011]. Figure 1 illustrates the effects of memory bandwidth on application performance by varying the available off-chip bandwidth in the Nvidia GTX 480 chipset with 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 our expectation that alleviating the bandwidth bottleneck can result in significant performance benefits. RFVP exploits this insight and aims to lower the memory bandwidth

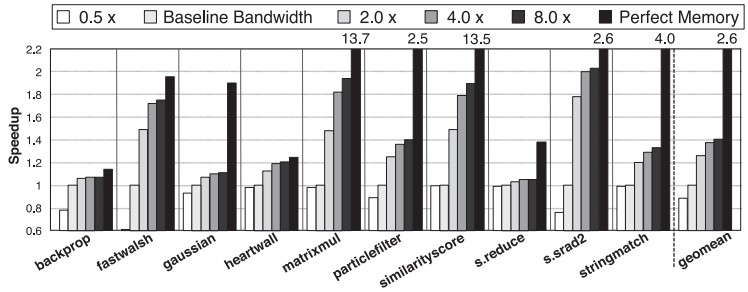


Fig. 1. Performance improvement with different amounts of DRAM bandwidth and perfect memory (last bar). The baseline bandwidth is 177.4GB/s (based on the Nvidia GTX 480 chipset with the Fermi architecture). The legend (N×) indicates a configuration with N times the bandwidth of the baseline. Perfect memory is an idealized system in which all memory accesses are L1 cache hits.

pressure by dropping a fraction of the value-predicted safe-to-approximate loads, slightly trading output quality for large gains in performance and energy efficiency.

Overview. As explained earlier, the key idea of RFVP is to predict the values of the safe-to-approximate loads when they miss in the cache with no checks or recovery from misprediction. 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 processing core to benefit from the spatial and temporal locality in future accesses.

Drop rate is a knob to control the trade-off between performance/energy gains and quality loss. A higher drop rate enables the core to use more predicted approximate values, thereby reducing main-memory accesses. We expose the drop rate as an architectural knob to the software. The compiler or the runtime system can use this knob to control the performance/energy and quality trade-off. Furthermore, RFVP enables the core to continue without stalling for long-latency memory accesses that service the value-predicted load misses. Consequently, these cache-missing loads are removed from the critical path of the program execution. We now elaborate on the safety guarantees with RFVP, its ISA extensions and their semantics, and its integration into the microarchitecture.

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 values of such loads. As prior work in approximation showed [Park et al. 2015; Sampson et al. 2011], 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 [Carbin et al. 2013; Esmaeilzadeh et al. 2012a, 2012b; Sampson et al. 2011; Baek and Chilimbi 2010]. 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 workflow and language support for RFVP.

2.3. Instruction Set Architecture to Support RFVP

We extend the ISA with two new instructions: (1) an approximate load instruction, and (2) a new instruction for setting the drop rate. Similar to prior work [Esmaeilzadeh et al. 2012a], we extend the ISA with dual approximate versions of the load instructions. A bit in the opcode is set when a load is approximate, permitting the microarchitecture to use RFVP. Otherwise, the load is *precise* and must be executed normally. 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. In this case, RFVP is triggered only when the load micro-op misses in the cache.

Drop rate. The *drop rate* is a knob that is exposed to the compiler to control the quality trade-offs. 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 fraction of approximate *cache misses* that do *not* initiate memory-access requests, and instead only trigger RFVP.[1] When the request is not dropped, it is considered a normal cache miss, and its value is fetched from memory.

Approximate load. 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 a predicted value in other cases. Intuitively, with RFVP, the probability of exact assignment is usually high for two reasons: (1) Our technique is triggered only by cache misses. Approximate loads that hit in the cache (the common case) return the correct value and (2) even in the case of a cache miss, the value predictor may generate a correct value prediction. Our measurements with a 50% drop rate show that, across all 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 take these probabilities into consideration.

2.4. Integrating RFVP into the Microarchitecture

As Figure 2 illustrates, the RFVP value predictor supplies the data to the processing core when triggered by a safe-to-approximate load. 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) [Murase 1992], 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 into the L1 cache.

If RFVP does not predict the *entire* cache line, a subsequent safe-to-approximate load to the same cache line would lead to another miss. But since RFVP does *not* predict and drop *all* missing safe-to-approximate loads, the same line would need to

[1]Another option is to enable dropping a certain fraction of all cache accesses, including hits. Such a policy may be desirable for controlling error in multikernel workloads.

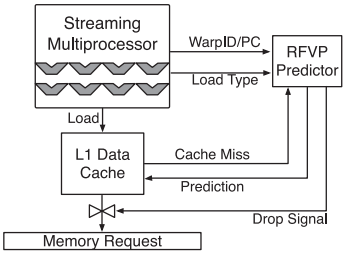


Fig. 2. Integration of the RFVP predictor into the GPU microarchitecture.

be requested again from memory. Thus, RFVP would not be able to effectively reduce bandwidth consumption if it did not insert the entire cache line. Hence, our decision is to value-predict and insert the *entire* cache line.

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 always begins at a cache-line boundary, and is padded to end at a cache line boundary. Thus, we can ensure that data-value prediction does not contaminate precise load operations.[2] The same stipulation has been set forth in several recent works in approximate computing, such as Truffle [Esmaeilzadeh et al. 2012a] and EnerJ [Sampson et al. 2011].

The coalescing logic in the SMs handles memory divergence and serializes the divergent threads. Since RFVP is triggered only 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 performance and energy-efficiency improvements, while limiting the adverse effects of approximation on output quality.

To achieve this goal, the compilation workflow for RFVP first identifies *performance-critical* loads. A performance-critical load is one that provides a higher potential for performance improvement when its latency is reduced. Second, among the performance-critical loads, the programmer determines and annotates the loads that are safe to approximate, which means they *will not* cause catastrophic failures if approximated. The list of performance-critical *and* safe-to-approximate loads are the candidates for approximation with RFVP. Afterwards, we determine the drop rate using either (1) programmer annotation, (2) compiler heuristics, or (3) a runtime system such as SAGE [Samadi et al. 2013]. In our evaluation, we empirically pick the best drop rate to maximize performance while maintaining high output quality.[3]

3.1. Targeting Performance-Critical Loads

The first step in the compilation workflow for RFVP is to identify the subset of the loads that cause the largest percentage of cache misses. As prior work has shown [Collins et al. 2001] and our experiments corroborate, only a very small fraction of the load instructions cause most of the total cache misses. Figure 3 illustrates this trend by

[2]Note that we do not use any padding for the baseline experiments. As a result, there is no artificial increase in bandwidth consumption in the baseline.

[3]A light profiling step can be utilized to automatically tune the drop rate for the target output quality requirement.

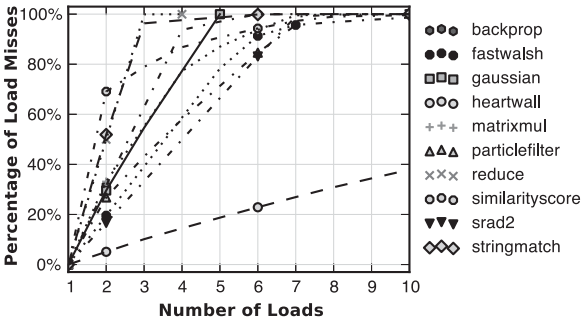


Fig. 3. Cumulative distribution function (CDF) plot of the LLC load cache misses. A point (x, y) indicates that y percent of the cache misses are caused by x distinct load instructions.

showing the cumulative distribution function (CDF) of the LLC cache misses caused by distinct load instructions in the GPU. As Figure 3 shows, in all of our GPU applications except one, almost six loads cause more than 80% of the misses. We refer to these loads as the *performance-critical* loads. This hot subset of loads is selected without any *a priori* knowledge of the applications. We use one training input to profile the benchmarks. The output of the profiling is a sorted list of the loads in decreasing order based on the fraction of cache misses that each load causes. Note that the list of performance-critical loads is obtained only once and is not changed based on the inputs. Clearly, focusing RFVP on these loads will provide the opportunity to eliminate a majority of the cache misses. Furthermore, the focus on a small, selected subset of loads reduces the predictor size and consequently its overheads. Therefore, this step provides the set of the most performance-critical and safe-to-approximate loads as candidates for approximation. We explain the nature of some of these loads in the Appendix. Note that programmer annotations identify which of these performance-critical loads are safe to approximate (see Sections 2.2 and 3.2).

3.2. Providing Safety Guarantees

The next step is to ensure that loads that can cause safety violations are excluded from RFVP. Any viable approximation technique, including ours, needs to provide strict safety guarantees. In other words, applying approximation should cause only graceful quality degradations without catastrophic failures, for example, segmentation faults or infinite loops.

Safety is a semantic property of a program [Park et al. 2015; Yazdanbakhsh et al. 2015; Mahajan et al. 2015; Carbin et al. 2013; Sampson et al. 2011]. Therefore, only the programmer can reliably identify which instructions are safe to approximate. For example, EnerJ [Sampson et al. 2011] provides language constructs and compiler support for annotating safe-to-approximate operations in Java. We do not expect the programmer to deeply annotate the approximable loads. Instead, programming models such as FlexJava [Park et al. 2015] and EnerJ [Sampson et al. 2011] are used that allow the programmer to annotate only the variable declarations, and the compiler automatically infers the approximable loads from these annotations. 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, RFVP still provides significant performance and energy gains (as our results in Section 6 confirm) because there are still enough performance-critical loads that are safe to approximate.

Figure 4 shows code examples from our applications to illustrate how approximating load instructions can lead to safety violations. In Figure 4(a), it is not safe to approximate loads from *ei*, *row*, *d_iS[row]* variables that are used as array indices.

```
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) Code example 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) Code example 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) Code example from *heartwall*

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

(d) Code example from *particlefilter*

Fig. 4. Code examples with different safety violations.

Approximating such loads may lead to out-of-bounds array accesses and segmentation faults. In Figure 4(b), 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 4(c), 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 an if-then-else statement can be if-converted to dataflow [Allen et al. 1983]. Therefore, it might be safe to approximate the loads that affect the if-convertible control flow conditionals. Figure 4(d) illustrates such a case. Loads for both *value* and *newValue* are safe to approximate even though they affect the if condition.

3.3. Drop-Rate Selection

These first two steps provide a small list of safe-to-approximate and performance-critical loads for RFVP. The final step in the RFVP compilation workflow is to pick the best drop rate that maximizes the benefits of RFVP while satisfying the target output quality requirement. In general, providing formal quality guarantees for approximation techniques across all possible inputs is still an open research problem. However, the drop rate is a knob that enables RFVP to explore various quality trade-offs. RFVP can use different techniques to select the drop rate. We can determine the drop rate dynamically at runtime using techniques such as those described in SAGE [Samadi et al. 2013]. SAGE uses computation sampling and occasional redundant execution on the CPU to dynamically monitor and control approximation. While dynamically setting the drop rate may provide the advantage of more *adaptive* error control, it comes at the cost of some additional overhead. Alternatively, a lightweight, profile-driven technique can statically determine the drop rate that satisfies the target output quality. To this end, the compiler can perform a stochastic binary search to determine the drop rate. The compiler starts with a drop rate of 50%. If this drop rate satisfies the output quality target, the compiler increases the drop rate by a delta. Otherwise, it reduces the drop rate. This process continues until the highest drop rate that stochastically satisfies the target quality is found. Ultimately, the compiler empirically picks a static drop rate in our evaluation, but using a dynamic scheme to adjust drop rate is also a viable alternative.

Altogether, the three steps (described in Sections 3.1, 3.2, and 3.3) provide a compilation workflow that focuses RFVP on the safe-to-approximate loads with the highest potential in terms of performance and energy savings while satisfying the target output quality requirement.

4. VALUE PREDICTOR DESIGN FOR RFVP

One of the main design challenges for effective RFVP is devising a low-overhead and 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 (*e.g.*, Perais and Seznec [2014] and Goeman et al. [2001]). We use the *two-delta* stride predictor [Eickemeyer and Vassiliadis 1993], due to its low complexity and reasonable accuracy, as the base for our multivalue prediction mechanism for GPUs (which predicts all values in an entire cache line). We have also experimented with other value-prediction mechanisms such as DFCM [Goeman et al. 2001], last value [Lipasti and Shen 1996] and stride [Sazeides and Smith 1997]. Empirically, two-delta predictor provides a good trade-off between accuracy and complexity. We choose this scheme because it requires only one addition to perform the prediction and only a few additions and subtractions for training. It also requires lower storage overhead than more accurate context-sensitive alternatives [Perais and Seznec 2014; Goeman et al. 2001]. However, this predictor cannot be readily used for multivalue prediction (for predicting the entire cache line), which is required for GPUs, as explained earlier. Due to the SIMD execution model in modern GPUs, the predictor needs to generate multiple parallel predictions for multiple parallel threads.

In the next section, we first describe the design of the base predictor, then devise a new predictor that performs full cache line multivalue GPU prediction.

4.1. Base Predictor for RFVP

Figure 5 illustrates the structure of the two-delta predictor [Eickemeyer and Vassiliadis 1993], which we use as the base predictor design for RFVP in GPUs.[4] The predictor consists of a value-history 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 Goeman et al. [2001]. Each row in the table stores three values: (1) the last *precise* value (64-bit), (2) Stride_1 (16-bit), and (3) Stride_2 (16b). The last value plus Stride_1 makes up the predicted value. 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, the predictor calculates the stride, the difference between the last value and the current value. If the stride is equal to 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 updates Stride_1, which is the prediction stride, only if it observes the same stride twice in a row. This technique produces a low rate of mispredictions, especially for integer workloads [Eickemeyer and Vassiliadis 1993]. However, for floating-point loads, it is unlikely to observe two matching strides. Floating-point additions and subtractions are also costly. Furthermore, RFVP performs *approximate* value predictions for error-resilient applications that can tolerate small deviations in floating-point values. Considering these challenges with floating-point value prediction 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. Value-Predictor Design for GPUs

Here, we elaborate on the RFVP predictor design for multivalue prediction, (i.e., for predicting the entire cache line) in GPUs, in which SIMD loads read multiple words.

[4]For clarity, Figure 5 does not depict the update logic of the predictor.

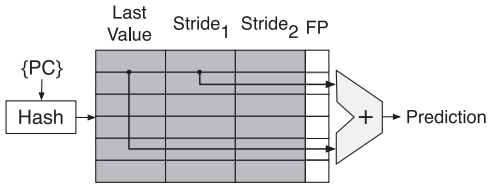


Fig. 5. Structure of the base two-delta predictor.

GPU predictor structure. The fundamental challenge in designing the GPU predictor is that a single data request is an SIMD load that must produce values for multiple parallel threads. A naive approach to performing value prediction in GPUs is to replicate the single-value predictor for each parallel thread. For example, in a typical modern GPU, there may be as many as 1,536 threads in flight during execution. Therefore, the naive predictor would require 1,536 separate two-delta predictors, which is impractical. Fortunately, we find that, 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 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, adjacent threads in a warp process data elements with some degree of value similarity, for example, pixels of an image. Previous work [Samadi et al. 2013] shows the value similarity between the neighboring locations in memory for GPGPU workloads. Furthermore, GPU bandwidth compression techniques (e.g., Vijaykumar et al. [2015]) exploit this value similarity in GPGPU workloads to compress data with simple compression algorithms [Pekhimenko et al. 2012]. Our evaluation also shows significant value similarity between adjacent threads in the applications that we study.

In our multivalue predictor design for GPUs, we leverage (1) the existing value similarity in the adjacent threads and (2) the fact that predictions are only *approximations* and the application can tolerate small prediction errors.

We design a predictor that consists of *only two* specialized two-delta predictors. In order to perform the *entire cache line*[5] prediction, we introduce special prediction and update mechanisms for RFVP, which we explain later in this section. Additionally, to reduce the conflicts between loads from different active warps, we make the GPU predictor set associative (using the Least Recently Used (LRU) replacement policy). As Figure 6 shows, for each row in the predictor, we keep the corresponding load's {WarpID, PC} as the row tag. A load value is predicted only if its {WarpID, PC} matches the row tag. For most 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. Using set associativity for the predictor is a unique aspect of our design that performs multivalue prediction. Section 6 provides a detailed design-space exploration for the GPU predictor, which includes the set associativity and number of parallel predictors, as well as the number of entries.

We explain the prediction and update mechanisms for our GPU configuration (Table II) in which there are 32 threads per warp. Our RFVP predictor can be easily adapted for other GPU configurations.

RFVP prediction mechanism. When there is a match between the {WarpID, PC} of a SIMD load and one of the row tags of the RFVP predictor, the predictor generates two predictions: one for ThreadID=0–15 and one for ThreadID=16–31. RFVP

[5]In our GPU configuration (Table II), each cache line has 32 4B words.

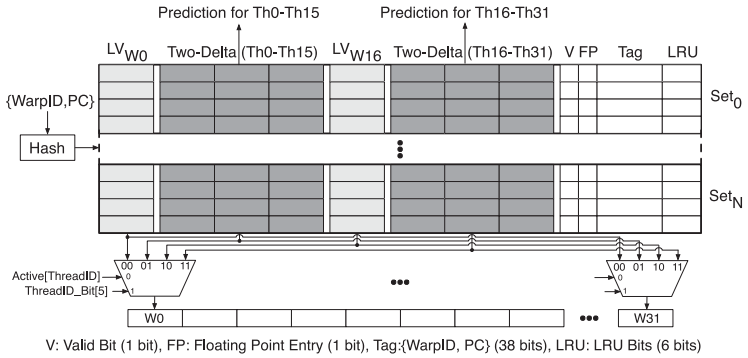


Fig. 6. Structure of the multivalue predictor for RFVP in GPUs. The GPU predictor consists of two two-delta and two last-value predictors. The GPU predictor is also set-associative to reduce the conflicts between loads from different active warps. The predictor produces predictions for full cache lines.

achieves the entire cache-line prediction by replicating the two predicted values for the corresponding threads. As Figure 6 shows, the Two-Delta (Th0–Th15) structure generates a prediction value that is replicated for threads 0 to 15. Similarly, the Two-Delta (Th16-Th31) generates a prediction for threads 16 to 31. Note that each of the two two-delta predictors works similarly as the baseline two-delta predictor [Eickemeyer and Vassiliadis 1993]. Using this approach, RFVP is able to predict the *entire cache line* for each SIMD load access. Due to the high cost of the floating-point operations, our RFVP predictor falls back to a simple last-value predictor for FP values. In other words, the predictor outputs the last value entry of each of the two two-delta predictors as the predicted data. We use the FP bit in the RFVP predictor to identify the floating-point loads.

In the GPU execution model, there might be situations in which an issued warp has less than 32 active threads. Having less than 32 active threads causes gaps in the predicted cache line. However, the data in these gaps might be used later by other warps. The simple approach is not to perform value prediction for these gaps and fill them with random data. Our evaluation shows that this approach leads to significant output-quality degradation. To avoid this quality degradation, RFVP fills the gaps with approximated data. We add a column to each two-delta predictor that tracks the last value of $\mathbf{word_0}$ and $\mathbf{word_{16}}$ in the cache line being accessed by the approximate load. When predicting the cache line, all the words that are accessed by the active threads are filled by the pair of two-delta predictors. The last value column of thread group **Th0–Th15** (LV_{W0}) is used to fill the gaps in W0 to W15. Similarly, the last value column of thread group **Th16–Th31** (LV_{W16}) is used to fill the gaps in W16 to W31. This proposed mechanism in RFVP guarantees that all the threads get value-predicted approximated data (instead of random data) and avoids significant output-quality degradation.

RFVP update mechanism. When a safe-to-approximate load misses in the cache but is *not* dropped, the predictor updates the two-delta predictor upon receiving the data from lower-level memory. The fetched data from lower-level memory is *precise*; we refer to its value as the current value. The Two-Delta (Th0-Th15) structure is updated with the current value of the active thread of the thread group ThreadID=0–15 with the lowest threadID. Similarly, the Two-Delta (Th16-Th31) is updated with the current value of the active thread of thread group ThreadID=16–31 with the lowest threadID.

5. EXPERIMENTAL METHODOLOGY

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

Table I. Evaluated GPU Applications, Input Data, Quality Metrics, and Characteristics

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

5.1. Experimental Methodology for GPUs

Applications. As Table I shows, we use a diverse set of already optimized GPU benchmarks from Rodinia [Che et al. 2009], Nvidia SDK, and Mars [He et al. 2008] benchmark suites to evaluate RFVP with the GPU architectures. Columns 1 through 3 of Table I 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 I. We use NVCC 4.2 from the CUDA SDK with -O3 flag to compile the applications for the Fermi microarchitecture. Furthermore, we do not perform any optimizations in the source code in favor of RFVP. We optimize only the number of thread blocks and number of threads per block of each kernel for our simulated hardware.

Quality metrics. Column 4 of Table I lists each application's quality metric. Each application-specific quality metric determines the application's output-quality loss as it undergoes RFVP approximation. Using application-specific quality metrics is commensurate with other works on approximation [Amant et al. 2014; Esmaeilzadeh et al. 2012b, 2012a; Sampson et al. 2011; Baek and Chilimbi 2010]. 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 as relative error. Since *gaussian* and *fastwalsh* output images, we use the image difference Root-Mean-Square Error (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 precise 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*.

Table II. Simulated GPU Microarchitectural Parameters

System Overview: 15 SMs, 32 threads/warp, 6 memory channels; **Shader Core Config**: 1.4GHz, 2 Schedulers/SM [Rogers et al. 2012], **Resources/SM**: 48 warps/SM, 32,768 registers, 32KB shared memory; **L1 Data Cache**: 16KB, 128B line, 4-way, LRU; **L2 Unified Cache**: 768KB, 128B line, 8-way, LRU; **Memory**: GDDR5, 924MHz, 16 banks/MC, FR-FCFS; **Interconnect**: 700MHz, 1 crossbar/direction (15 SMs, 6 MCs); **Off-Chip Bandwidth**: 177.4GB/s

Load identification. The final column of Table I lists the number of static approximate loads identified. These loads are a subset of all the loads that are marked as safe to approximate by the programmer. There are only a few loads that contribute significantly to the total number of cache misses. Therefore, RFVP focuses on the intersection of the performance-critical loads and the loads that are marked by the programmer as safe to approximate (as described in Sections 3.1 and 3.2). For some applications, such as *backprop*, *fastwalsh*, and *particlefilter*, we list the number of approximate loads for each individual kernel within the application as a tuple in Table I, for example, (2, 1, 4) for *fastwalsh*.

Cycle-level microarchitectural simulation. We use the cycle-level GPGPU-Sim simulator version 3.1 [Bakhoda et al. 2009]. We modified the simulator to include our ISA extensions, value prediction, and all necessary cache and memory logic to support RFVP. Table II summarizes the microarchitectural parameters of our baseline GPU. We run each application 10 times with different input datasets to completion and report the average results.

Energy modeling and overheads. To measure the energy benefits of RFVP, we use GPUWattch [Leng et al. 2013], which is integrated with GPGPU-Sim. GPUWattch models the power consumption of the cores, on-chip interconnect, caches, memory controller, and DRAM. RFVP comes with overheads, including the prediction tables, arithmetic operations, 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 [Muralimanohar et al. 2007]. We extract the overhead of arithmetic operations from McPAT [Li et al. 2009]. Our energy evaluations use a 40nm process node and 1.4GHz clock frequency (similar to the shader core clock frequency). Furthermore, we have synthesized the LFSR and the hash function and incorporated the energy overheads. The default RFVP prediction table size is 14KB per SM and the GPU consists of 15 SMs. The off-chip memory bandwidth of the simulated GPU is 177.4GB/s.

6. EXPERIMENTAL RESULTS

6.1. GPU Measurements

Speedup, energy, memory bandwidth, and quality. Figure 7(a) shows the speedup with RFVP for maximum 1%, 3%, 5%, and 10% quality degradation. We have explored this trade-off by setting different drop rates, which is RFVP's knob for quality control. The baseline is the default GPU system that we model without RFVP. Figures 7(b) and 7(c) illustrate the energy reduction and the reduction in off-chip bandwidth consumption, respectively. The error bars show the standard deviation of 10 simulation runs with different inputs.

As Figures 7(a) and 7(b) 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 fewer than 10 static performance-critical load instructions per kernel. The results show the effectiveness of our proposed mechanism in focusing approximation where it is most beneficial.

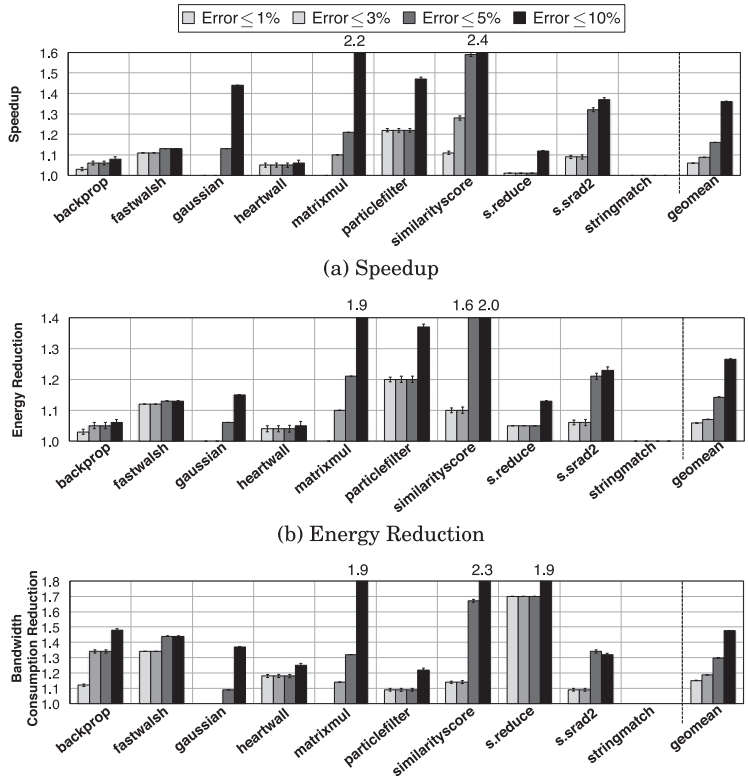


Fig. 7. GPU (a) performance improvement, (b) energy reduction, and (c) memory bandwidth consumption reduction for at most 1%, 3%, 5%, and 10% quality degradation. Error bars show the standard deviation of 10 simulation runs with different inputs.

With 5% quality loss, the average performance and energy gains are 16% and 14%, respectively. Thus, RFVP is able to navigate the trade-off between quality loss and performance-energy improvement based on the user requirements.

Even with a small quality degradation (*e.g.*, 1%), RFVP yields significant speedup and energy reduction in several applications, including *fastwalsh*, *particlefilter*, *similarityscore*, *s.srad2*. The benefits are as high as 22% speedup and 20% energy reduction for *particlefilter* with strictly less than 1% quality loss.

Comparing Figures 7(a), 7(b), and 7(c) 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 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 7(c) 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 perfect memory. Therefore, the benefits from RFVP are expected to be limited for this application even with significant bandwidth reduction. This case shows

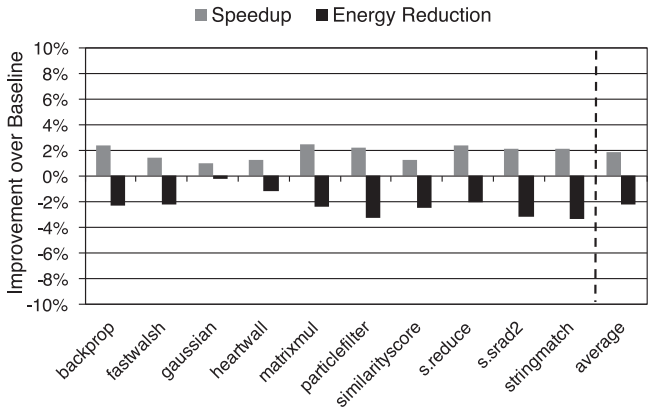


Fig. 8. Speedup and energy reduction when RFVP does *not* drop memory requests and sends them to the lower memory subsystem (the value-predicted line is still inserted in the L1 cache).

that an application's performance sensitivity to off-chip communication bandwidth is an important factor in RFVP's ability to improve performance and energy efficiency. Also, RFVP provides no benefit for *stringmatch* with 10% quality degradation. This case is an interesting outlier that we discuss in greater detail in the next section.

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 requests. In this scenario, when the memory request completes, the predictor is updated with the fetched value from the main memory. The results are presented in Figure 8. Without dropping requests, RFVP yields only 2% performance improvement and *increases* energy consumption by 2% on average for these applications. These results suggest that the source of RFVP's benefits come primarily from reduced bandwidth consumption, which is a major bottleneck in GPUs that hide latency with many-thread execution.[6]

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

Sources of quality degradation. To determine the effectiveness of our value prediction, we measure the portion of load operations that return approximate values. Figure 9 shows the result of these measurements for three different drop rates: 12.5%, 25%, and 50%. The results show that, on average, only 2% (max 5.4%) of all dynamic load instructions return approximate values for a 25% drop rate. This percentage increases to 3% (max 7.5%) for a 50% drop rate. Thus, a large majority of all dynamic loads return exact values, even at reasonably high drop rates. The prediction accuracy is relatively low (on average, 63%), yet commensurate with prior works on value prediction [Ceze et al. 2006; Goeman et al. 2001; Eickemeyer and Vassiliadis 1993]. However, RFVP focuses approximation only on the small subset of loads that are *both* performance-critical and safe to approximate. Thus, due to the small fraction of

[6]We study the effects of RFVP on single-core CPUs that are more latency sensitive in Section 7.

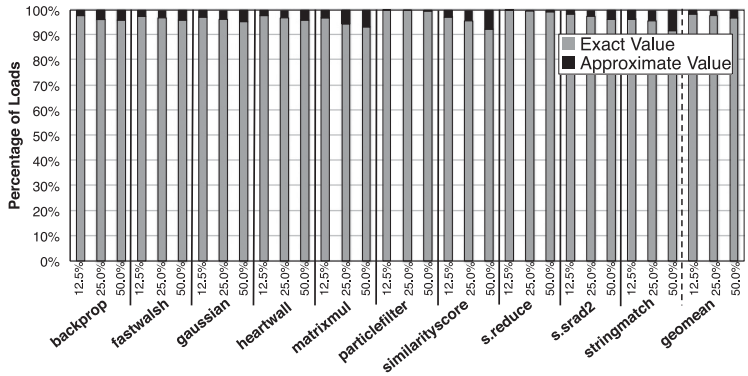


Fig. 9. Fraction of load instructions that receive exact and approximate values during execution for three different drop rates: 12.5%, 25%, and 50%.

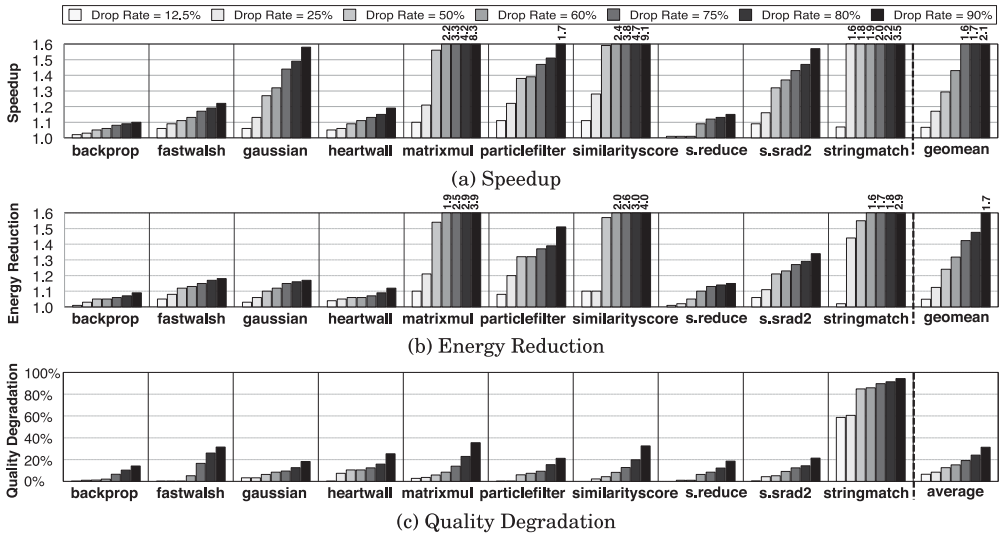


Fig. 10. Effect of *drop rate* on RFVP's (a) speedup, (b) energy reduction, and (c) quality degradation.

loads predicted with approximate values, RFVP leads to low quality degradations (as we explain in Figure 10(c)).

Quality trade-offs with drop rate. Drop rate is RFVP's knob for navigating quality trade-offs. It dictates the fraction of the value-predicted approximate load cache misses that are also dropped. For example, with a 12.5% drop rate, RFVP drops one out of eight approximate load cache-misses. We examine the effect of this knob on performance, energy, and quality by sweeping the drop rate from 12.5% to 90%. Figure 10 illustrates the effect of drop rate on speedup (Figure 10(a)), energy reduction (Figure 10(b)), and quality degradation (Figure 10(c)).

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

Figure 10(c) shows that, in all but one case, quality degradation increases slowly and steadily as the drop rate increases. The clear exception is *stringmatch*. This application searches a file with a large number of strings to find the lines that contain a search word. This application's input dataset contains only English words with very low value locality. The application outputs the indices of the matching lines, which have 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 identified lines. During search, even if a single character is incorrect, the likelihood of matching the words and identifying the correct lines is low.

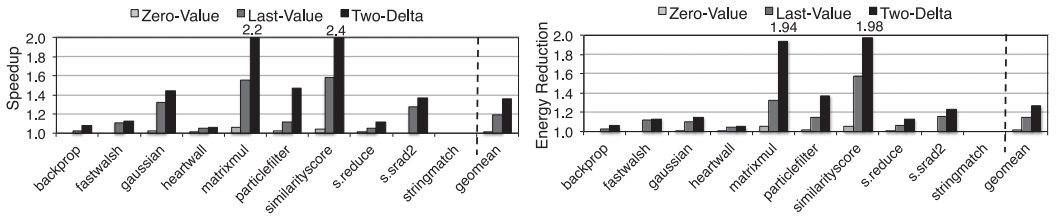
Even though *stringmatch* shows 61% speedup and 44% energy reduction with a 25% drop rate, the corresponding quality loss of 60% is not acceptable. In fact, *stringmatch* is an example of an application that *cannot* benefit from RFVP due to its low error tolerance.

As Figure 10 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 rises by only 2%.

We conclude that RFVP provides high performance and energy-efficiency benefits at acceptable quality-loss levels (as shown in Figure 10), for applications whose performance is most sensitive to memory bandwidth (as shown in Figure 1).

RFVP with a different base predictor. RFVP can employ a variety of base predictors. There is a trade-off between the simplicity of the predictor and the performance and energy reduction that it can provide. We study the performance and the energy reduction with three different base predictors for a given quality loss: (1) Zero-Value, (2) Last-Value, and (3) Two-Delta predictors. For each predictor, we pick the drop rate that leads to less than 10% quality loss. Figure 11 shows the speedup and energy reduction of using different predictors with RFVP for 10% quality loss. Using a Zero-Value predictor shows almost no speedup and energy reduction benefit, because simply predicting zero leads to significant quality degradation beyond the target of 10%. Only in *fastwalsh* and *heartwall* does RFVP with the Last-Value predictor provide almost similar improvements as our Two-Delta predictor. In these two applications, the majority of the loads are floating point and the Two-Delta predictor falls back to the simple Last-Value predictor. In the remaining applications, our predictor that uses the two-delta algorithm provides significantly higher benefits. These benefits are achieved with reasonably low area overhead, as discussed earlier.

Design space exploration and Pareto analysis. The two main design parameters of our GPU value predictor are: (1) the number of predictors per warp and (2) 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 best configuration. Figure 12shows the result of this design-space exploration. The x-axis captures the complexity of the predictor in terms of size (in kilobytes). The y-axis is the Normalized Energy$\times$Normalized Delay$\times$Error across all the GPU applications. The



(a) Speedup (b) Energy Reduction

Fig. 11. Effect of different value predictors on RFVP's (a) speedup and (b) energy reduction, with 10% quality loss.

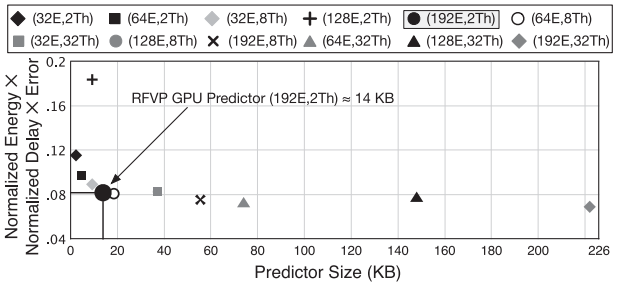


Fig. 12. GPU value predictor design-space exploration and Pareto analysis for RFVP. The point (xE,yTh) represents a configuration with y predictors per warp and each with x entries. All the predictors are 4-way set associative. The predictor configuration of (192E,2Th), which is our default configuration, is the most Pareto-optimal design point. In this graph, lower and left is better. The normalization baseline is our GPU system without RFVP.

normalization baseline is our GPU system without RFVP. This product simultaneously captures the three metrics of interest: performance, energy, and quality. The optimal predictor minimizes both size (left on the x-axis), energy dissipation, execution delay, and error (lower on the y-axis). In Figure 12, (xE,yTh) represents a configuration with y predictors per warp and each with x entries. All the predictors are 4-way set associative.

In Figure 12, the knee of the curve is the most cost-effective point. This Pareto-optimal design is the (192E,2Th) configuration, which requires 14KB 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 of RFVP. Increasing the number of entries from 32 to 192 provides 1.4× improvement in Normalized Energy× Normalized Delay × Error. More entries lower the chance of destructive aliasing in the prediction table that leads to the frequent eviction of value history from the prediction tables. However, adding more predictors per warp beyond a certain point does not provide any significant benefit in terms of improving the output quality, and instead wastes area and reduces the energy saving. With fewer predictors, RFVP relies more on the value locality across the threads, which is the common case in GPU applications [Samadi et al. 2014; Arnau et al. 2014].

Cache sensitivity study. We compare the benefits of RFVP with the benefits that can be achieved with simply enlarging the caches by same amount as the RFVP predictor size. Similar to other works [Rogers et al. 2012; Bakhoda et al. 2009], we do

Table III. Evaluated CPU Applications, Input Data, and Quality Metrics

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

Exploiting value similarity in GPU applications enables RFVP to use a smaller predictor without significant degradation in output quality. Thus, a 14KB predictor per each SM, which is the Pareto-optimal design, exploits value similarity and produces significant gains in performance and energy saving while maintaining high output quality.

not recompile the source code for different cache sizes. 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 improvements.

Comparison with loop perforation. With 10% quality loss, loop perforation [Sidiroglou-Douskos et al. 2011] provides 18% average speedup (max 25%) and 19% average energy reduction (max 28%). In contrast, RFVP provides 36% average speedup and 27% average energy reduction. Our results are on par with previous studies on loop perforation in GPUs [Samadi et al. 2013]. Loop perforation leads to lower performance because simply skipping loop iterations leads to significant output quality loss that limits the use of loop perforation.

7. EFFECT OF RFVP ON CPU-BASED SYSTEMS

To understand the effectiveness of RFVP in a system in which latency is the primary concern, we investigate the integration of RFVP in a single-core CPU system.

7.1. Methodology

Applications. As Table III shows, we evaluate RFVP for CPUs using an approximable subset of SPEC CFP 2000/2006. The applications come from the domains of scientific computing and optimization. As the work in Sethumadhavan et al. [2012] 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 with *-O3* to enable compiler optimizations.

Quality metrics. As discussed later, our subset of the SPEC applications produce numerical outputs. Therefore, we use NRMSE (see Section 5.1) to measure quality loss. For *swim*, the output consists 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*,

Table IV. Simulated 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 48KB, BTB Sets/Ways: 1024/4, RAS Entries: 64, Dependence Predictor: 4096-entry Bloom Filter, ITLB/DTLB Entries: 128/256; **L1:** 32KB I$, 32KB D$, 64B line, 8-Way, Latency: 2 cycles; **L2:** 2MB, 64B line, 8-Way, Latency: 20 cycles; **Memory Latency:** 200 cycles

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.

Load identification. We use an approach similar to the one we used in our GPU evaluation to identify the loads that are *both* performance-critical and safe to approximate. We use Valgrind with the Cachegrind tool [Nethercote and Seward 2007] for final quality of result evaluation. We modify Cachegrind to support RFVP. Valgrind is fast enough to run our applications until completion with SPEC reference datasets.

Cycle-level simulations. We implement RFVP in the MARSSx86 cycle-level simulator [Patel et al. 2011]. The baseline memory system includes a 32KB L1 cache, a 2MB LLC, and external memory with 200-cycle access latency. In modern processors, the LLC size is often 2MB $\times$ number of cores. Thus, we use a 2MB 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 [Molka et al. 2009]. Because simulation until completion is impractical for SPEC applications with reference datasets, we use Simpoint [Hamerly et al. 2004] 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 IV summarizes the microarchitectural parameters for our simulated CPU. As in GPU evaluations, we run each application 10 times with different input datasets and report the average.

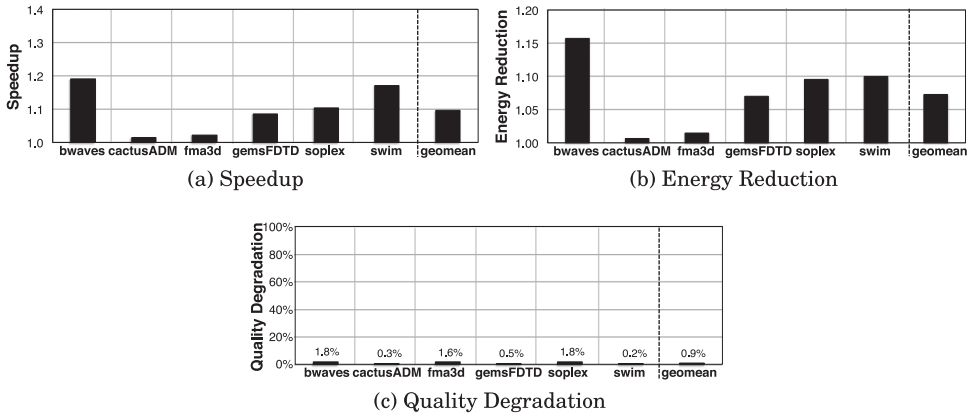
Energy modeling and overheads. We use McPAT [Li et al. 2009] and CACTI [Muralimanohar et al. 2007] to measure energy benefits while considering all the overheads associated with RFVP. The caching overheads are incorporated into the statistics that Marssx86 produces for McPAT. As in our GPU evaluations, we estimate the prediction table overhead using CACTI version 6.5, and extract the arithmetic operations overhead from McPAT. The energy evaluations use a 45nm process, 0.9Vdd and 3.0GHz core clock frequency.

7.2. Results

Figure 13 shows the speedup, energy reduction, and quality degradation with RFVP. The baseline is the CPU system with no RFVP. Our proposed RFVP technique aims to mitigate the long memory-access latencies in a CPU. 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 13 shows, RFVP provides 9.7% average speedup and 6.2% energy reduction with a single-core CPU. The average quality loss is 0.9%.

While RFVP's benefits on the CPU system are lower than its benefits on the GPU system, the output quality degradations on the CPU system are also comparatively



(a) Speedup

(b) Energy Reduction

(c) Quality Degradation

Fig. 13. Effect of RFVP in a single-core CPU: (a) speedup, (b) energy reduction, and (c) quality degradation.

Table V. L2 MPKI Comparison with and without RFVP on the CPU System

	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

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 13 shows that one CPU application (*bwaves*) still gains 19% speedup and 16% energy reduction with only 1.8% quality degradation.

To better understand RFVP's performance and energy benefits on the CPU system, we examine MPKI reduction in the L2 cache, and present the results in Table V. 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 reduces 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 of 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 9). In 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 on a CPU system achieves performance and energy gains that are one order of magnitude greater than the quality loss.

The value prediction accuracy in the CPU system is on par with prior work [Ceze et al. 2006; Goeman et al. 2001; Eickemeyer and Vassiliadis 1993] and the GPU system. Once again, RFVP 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 output quality.

8. RELATED WORK

To our knowledge, this article is the first work that: (1) provides a mechanism for approximate value prediction for load instructions in GPUs, (2) enables memory bandwidth savings by enabling the dropping value-predicted memory requests at acceptable output quality loss levels, and (3) develops a new multiple-value prediction mechanism for GPUs that enables the prediction of entire cache lines.

Next, we discuss related works in (1) approximate computing, (2) value prediction, and (3) load value approximation.

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

EnerJ [Sampson et al. 2011] is a language for approximate computing. Its corresponding architecture, Truffle [Esmaeilzadeh et al. 2012a], leverages 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 Sampson et al. [2013] and Liu et al. [2011] design approximate DRAM and Flash storage blocks. Flikker [Liu et al. 2011] reduces the DRAM refresh rate when approximate data is stored in main memory. The work in Arnau et al. [2014] 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 bottleneck directly by eliminating memory traffic. The work in Sampson et al. [2013] 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 multilevel cells with small error margins. Luo et al. [2014] describe a heterogeneous-reliability memory system in which a part of memory is unreliable and thus can output approximate data. They show that such a memory system can lead to significant cost savings, but do not optimize performance. The technique in Sartori and Kumar [2013] exploits approximation to mitigate branch and memory divergence in GPUs. In the case of branch divergence, the authors force all the threads to execute the most popular path. In the 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 or drop memory requests. In contrast, our new 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 described prior works on approximation [Luo et al. 2014; Arnau et al. 2014; Sampson et al. 2013; Liu et al. 2011; Sidiroglou-Douskos et al. 2011], since it entirely focuses on mitigating memory latency and bandwidth bottlenecks.

Value prediction. RFVP takes inspiration from prior work that explores exact value prediction [Perais and Seznec 2014; Collange et al. 2010; Zhou and Conte 2005; Mutlu et al. 2005, 2003; Goeman et al. 2001; Thomas and Franklin 2001; Sazeides and Smith 1997; Lipasti et al. 1996; Eickemeyer and Vassiliadis 1993]. However, our work fundamentally differs from traditional value prediction techniques because it does *not* check

for mispredictions and does *not* recover from them. Furthermore, we drop a fraction of the load requests to reduce off-chip memory traffic. Among these techniques, Zhou and Conte [2005] and Mutlu et al. [2005] 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 initiates only 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, taking advantage of the error tolerance of applications.

Load value approximation. In our previous work [Thwaites et al. 2014], we introduced the RFVP technique for a conventional CPU processor to lower the effective memory-access latency. Later, in a concurrent effort, San Miguel et al. [2014] proposed a technique that uses value prediction without checks for misprediction to address the memory-latency bottleneck in CPU-based systems. San Miguel et al. [2014] corroborated our reported results in Thwaites et al. [2014]. Our work here differs from our previous work [Thwaites et al. 2014] and the concurrent work [San Miguel et al. 2014] as follows: (1) we develop techniques for GPU processors, targeting the memory bandwidth bottleneck rather than latency, showing that RFVP is an effective approach for mitigating both latency and bandwidth bottlenecks; (2) we utilize the value similarity of accesses across adjacent threads in GPUs to develop a low-overhead, multivalue predictor capable of producing values for many simultaneously missing loads as they execute in lock step in GPU cores; and (3) we drop a portion of cache-miss load requests to fundamentally reduce the memory bandwidth demand in GPUs.

9. CONCLUSIONS

This article introduces Rollback-Free Value Prediction (RFVP) and demonstrates its effectiveness in tackling two major memory system bottlenecks: (1) limited off-chip bandwidth and (2) long memory-access latency. RFVP predicts the values of safe-to-approximate loads only when they miss in the cache and drops a fraction of them without checking for mispredictions or recovering from them. Additionally, we utilize programmer annotations to *guarantee* safety, while our compilation workflow applies approximation only to the most performance-critical cache-missing loads and maintaining high output quality.

We develop a lightweight and fast-learning prediction mechanism for GPUs, which is capable of (1) predicting values in an entire cache line and (2) adapting to rapidly changing value patterns between individual loads with low hardware overhead.

RFVP uses predicted values both to hide the memory latency and ease bandwidth limitations. The drop rate is a knob that controls the trade-off between quality of results and performance/energy gains. Our extensive evaluation shows that RFVP, when used in GPUs, yields significant performance improvements and energy reductions for a wide range of quality-loss levels. As the acceptable quality loss increases, the benefits of RFVP increase. Even at a modest 1% acceptable quality loss, RFVP improves performance and reduces energy consumption by more than 20% in some applications. We also evaluate RFVP's latency benefits for a single-core CPU. The results show performance improvements and energy reductions for a wide variety of applications with quality loss less than 1%. These results confirm that RFVP is a promising technique to tackle the memory bandwidth and latency bottlenecks in applications that exhibit some level of error tolerance.

```
float newVal;
for (int i = 0; i < col4; i++)
{
   float4 v1 = matrix1 [i];
   float4 v2 = matrix2 [i];
   newVal += v1.x * v2.x;
   newVal += v1.y * v2.y;
   newVal += v1.z * v2.z;
   newVal += v1.w * v2.w;
}
```

(a) Code example from *matrixmul*

```
float likelihoodSum;
for (x = 0; x < numOnes; x++)
{
  likelihoodSum +=
    pow((I [ind[i * num + x]]
      - 100), 2) -
    pow((I [ind[i * num + x]]
      - 228), 2))
    / 50.0;
}
```

(b) Code example from *particlefilter*

```
float d_psum, d_psum2;
for(i=2; i<=df; i=2*i)
{
  d_psum[tx] = d_psum [tx] + d_psum [tx-i/2];
  d_psum2[tx] = d_psum2 [tx] + d_psum2 [tx-i/2];
}
```

(c) Code example from *s.reduce*

```
int d_cN, d_cS, d_cW, d_cE;
d_cN = d_c [ei];
d_cS = d_c [d_iS[row] + d_Nr*col];
d_cW = d_c [ei];
d_cE = d_c [row + d_Nr * d_jE[col]];
```

(d) Code example from *s.srad2*

Fig. 14. Code examples from some of the evaluated benchmarks. The gray shaded variables are the variables that are annotated by the programmer. The underlined variables are the safe-to-approximate loads that can be value-predicted by RFVP.

APPENDIX

A.1. Code Examples from Approximate Benchmarks

Figure 14 shows code examples from some of our studied applications to provide insights into where source of RFVP's benefits are coming from. The gray shaded variables are marked by the programmer. Similar to other works [Park et al. 2015; Sampson et al. 2011], we rely on the programmer to annotate the safe-to-approximate variables. The compilation workflow automatically infers the safe-to-approximate loads (as described in Section 3). In Figure 14, we show the safe-to-approximate loads with underlines. These code examples show that safe-to-approximate loads are mostly array elements and can access both integer and floating-point datatypes. However, in some scenarios, it might *not* be safe to approximate array elements. In Figure 14(d), we observe that the elements of array *d_iS* are used as index of array *d_c*. Thus, it is not safe to approximate the loads from this array.

ACKNOWLEDGMENTS

We thank anonymous reviewers of ACM TACO as well as anonymous reviewers of PACT 2014, ASPLOS 2015, and ISCA 2015, who reviewed previous versions of this work. This work was supported by a Qualcomm Innovation Fellowship; Microsoft Research PhD Fellowship; Nvidia; NSF awards #1409723, #1423172, #1212962, and CCF #1553192; Semiconductor Research Corporation contract #2014-EP-2577; and a gift from Google.

REFERENCES

J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *POPL*.

Carlos Alvarez, Jesus Corbal, and Mateo Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computing* 54, 7.

Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. 2014. General-purpose code acceleration with limited-precision analog computation. In *ISCA*.

Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2014. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. In *ISCA*.

Woongki Baek and Trishul M. Chilimbi. 2010. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*.

A Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong, and T. M. Aamodt. 2009. Analyzing CUDA workloads using a detailed GPU simulator. In *ISPASS*.

Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*.

Luis Ceze, Karin Strauss, James Tuck, Josep Torrellas, and Jose Renau. 2006. CAVA: Using checkpoint-assisted value prediction to hide L2 misses. *ACM Transaction on Architecture and Code Optimization* 3, 2.

Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. 2006. Ultra-efficient (Embedded) SoC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE*.

Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC*.

Eric S. Chung, Peter A. Milder, James C. Hoe, and Ken Mai. 2010. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPUs? In *MICRO*.

Sylvain Collange, David Defour, and Yao Zhang. 2010. Dynamic detection of uniform and affine vectors in gpgpu computations. In *Euro-Par (Parallel Processing Workshops)*.

Jamison D. Collins, Hong Wang, Dean M. Tullsen, Christopher Hughes, Yong-Fong Lee, Dan Lavery, and John P. Shen. 2001. Speculative precomputation: Long-range prefetching of delinquent loads. In *ISCA*.

Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. 2010. Relax: An architectural framework for software recovery of hardware faults. In *ISCA*.

Richard J. Eickemeyer and Stamatis Vassiliadis. 1993. A Load-instruction unit for pipelined processors. 37, 4.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012a. Architecture support for disciplined approximate programming. In *ASPLOS*.

Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012b. Neural acceleration for general-purpose approximate programs. In *MICRO*.

Bart Goeman, Hans Vandierendonck, and Koenraad De Bosschere. 2001. Differential FCM: Increasing value prediction accuracy by improving table usage efficiency. In *HPCA*.

Greg Hamerly, Erez Perelman, and Brad Calder. 2004. How to use SimPoint to pick simulation points. *ACM SIGMETRICS Performance Evaluation Review* 31, 4.

Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. 2008. Mars: A mapreduce framework on graphics processors. In *PACT*.

Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. 2011. GPUs and the future of parallel computing. *IEEE Micro* 5.

Jingwen Leng, Tayler Hetherington, Ahmed ElTantawy, Syed Gilani, Nam Sung Kim, Tor M. Aamodt, and Vijay Janapa Reddi. 2013. GPUWattch: Enabling energy optimizations in GPGPUs. In *ISCA*.

Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. 2009. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *MICRO*.

Xuanhua Li and Donald Yeung. 2007. Application-level correctness and its impact on fault tolerance. In *HPCA*.

Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the dataflow limit via value prediction. In *MICRO*.

Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. In *ASPLOS*.

Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*.

Yixin Luo, Sriram Govindan, Bhanu P. Sharma, Mark Santaniello, Justin Meza, Apoorv Kansal, Jie Liu, Badriddine Khessib, Kushagra Vaid, and Onur Mutlu. 2014. Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory. In *DSN*.

Divya Mahajan, Kartik Ramkrishnan, Rudra Jariwala, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Anandhavel Nagendrakumar, Abbas Rahimi, Hadi Esmaeilzadeh, and Kia Bazargan. 2015. Axilog: Abstractions for approximate hardware design and reuse. In *IEEE Micro*.

D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. 2009. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *PACT*.

Naveen Muralimanohar, Rajeev Balasubramonian, and Norm Jouppi. 2007. Optimizing NUCA organizations and wiring alternatives for large caches with CACTI 6.0. In *MICRO*.

Makoto Murase. 1992. Linear feedback shift register. US Patent.

Onur Mutlu, Hyesoon Kim, and Yale N. Patt. 2005. Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society.

Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt. 2003. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro* 23.

Nicholas Nethercote and Julian Seward. 2007. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*.

Jongse Park, Hadi Esmaeilzadeh, Xin Zhang, Mayur Naik, and William Harris. 2015. FlexJava: Language support for safe and modular approximate programming. In *FSE*.

Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSSx86: A full system simulator for x86 CPUs. In *DAC*.

Gennady Pekhimenko, Evgeny Bolotin, Mike O'Connor, Onur Mutlu, Todd Mowry, and Stephen Keckler. 2015. Toggle-aware compression for GPUs. *Computer Architecture Letters.*

Gennady Pekhimenko, Vivek Seshadri, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2012. Base-delta-immediate compression: Practical data compression for on-chip caches. In *PACT*.

A. Perais and A. Seznec. 2014. Practical data value speculation for future high-end processors. In *HPCA*.

Brian M. Rogers, Anil Krishna, Gordon B. Bell, Ken Vu, Xiaowei Jiang, and Yan Solihin. 2009. Scaling the bandwidth wall: Challenges in and avenues for CMP scaling. In *ISCA*.

Timothy Rogers, Mike O'Connor, and Tor Aamodt. 2012. Cache-conscious wavefront scheduling. In *MICRO*.

Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*.

Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. 2013. SAGE: Self-tuning approximation for graphics engines. In *MICRO*.

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

Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. 2013. Approximate storage in solid-state memories. In *MICRO*.

Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. 2014. Load value approximation. In *MICRO*.

J. Sartori and R. Kumar. 2013. Branch and data herding: Reducing control and memory divergence for error-tolerant GPU applications. *IEEE Transactions on Multimedia,* 15, 2.

Yiannakis Sazeides and James E. Smith. 1997. The predictability of data values. In *MICRO*.

S. Sethumadhavan, R. Roberts, and Y. Tsividis. 2012. A case for hybrid discrete-continuous architectures. *Computer Architecture Letters* 11, 1.

Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*.

R. Thomas and M. Franklin. 2001. Using dataflow based context for accurate value prediction. In *PACT*.

Bradley Thwaites, Gennady Pekhimenko, Hadi Esmaeilzadeh, Amir Yazdanbakhsh, Onur Mutlu, Jongse Park, Girish Mururu, and Todd Mowry. August, 2014. Rollback-free value prediction with approximate loads. In *PACT*.

Nandita Vijaykumar, Gennady Pekhimenko, Adwait Jog, Abhishek Bhowmick, Rachata Ausavarungnirun, Chita Das, Mahmut Kandemir, Todd C. Mowry, and Onur Mutlu. 2015. A case for core-assisted bottleneck acceleration in GPUs: Enabling flexible data compression with assist warps. In *ISCA*.

Amir Yazdanbakhsh, Divya Mahajan, Bradley Thwaites, Jongse Park, Anandhavel Nagendrakumar, Sindhuja Sethuraman, Kartik Ramkrishnan, Nishanthi Ravindran, Rudra Jariwala, Abbas Rahimi, Hadi Esmaeilzadeh, and Kia Bazargan. 2015. Axilog: Language support for approximate hardware design. In *DATE*.

Huiyang Zhou and Thomas M. Conte. 2005. Enhancing memory-level parallelism via recovery-free value prediction. *IEEE Transactions on Computers* 54, 7.

Received June 2015; revised August 2015; accepted October 2015