# Mitigating the Memory Bottleneck with Approximate Load Value Prediction

Amir Yazdanbakhsh, Gennady Pekhimenko*, Bradley Thwaites
Hadi Esmaeilzadeh, Onur Mutlu*, and Todd C. Mowry*

Georgia Institute of Technology *Carnegie Mellon University

***Abstract*— This paper aims to tackle two fundamental memory bottlenecks: limited off-chip bandwidth and long access latency. Our approach exploits the inherent error resilience of a wide range of applications through 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 long memory access latencies by enabling the execution to continue without stalling for these accesses. To mitigate the limited off-chip bandwidth, RFVP drops a fraction of load requests which miss in the cache after predicting their values. The drop rate then becomes a knob to control the tradeoff between performance/energy efficiency and output quality.**

**Our extensive evaluations show that RFVP, when used in GPUs, yields significant performance improvements and energy reductions for a wide range of quality loss levels.**

***Index Terms*—Memory Bandwidth, Load Value Approximation, Approximate Computing, GPUs, Value Prediction, Memory Latency.**

## I. 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 (*memory wall*). Modern GPUs exploit large-scale data parallelism to hide main memory latency. However, this solution suffers from a fundamental bottleneck: limited off-chip communication bandwidth to supply data to processing units (*bandwidth wall*). Fortunately, there is an opportunity to leverage the inherent error resiliency of many emerging applications to tackle the memory latency and bandwidth problems. Our paper exploits this opportunity.

Large classes of emerging applications such as 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, storage capacity, and hardware cost [1], [2], [3], [4].[1] However, there is a lack of approximation techniques that address the key memory system performance bottlenecks of long access latency and limited off-chip bandwidth.

[1]In the interest of space, we provide one representative citation in this paper.

To mitigate these memory subsystem bottlenecks, this paper introduces a new approximation technique called *Rollback-Free Value Prediction (RFVP)*. The key idea behind RFVP 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 devise new concepts and mechanisms that maximize RFVP's opportunities for performance and energy gains with acceptable quality loss.

## II. Architecture Design for RFVP

### A. 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 [5]. Figure 1 illustrates the effects of memory bandwidth on application performance by varying the available off-chip bandwidth in the Fermi architecture. These results support our expectation that alleviating the bandwidth bottleneck can result in significant performance benefits. RFVP aims to lower the memory bandwidth pressure by dropping a fraction of the predicted safe-to-approximate loads, trading output quality slightly for large gains in performance and energy efficiency.

**Overview.** As explained earlier, 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 processing core to benefit from the spatial and temporal locality in future accesses.

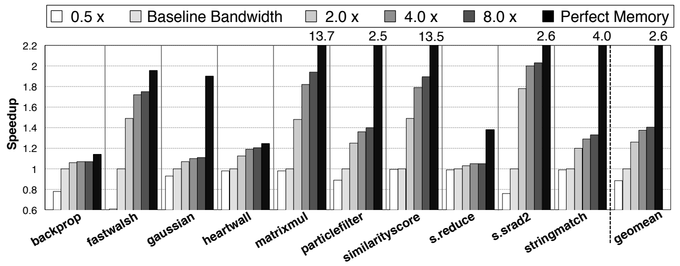


Fig. 1: Performance improvement with different amounts of DRAM bandwidth and perfect memory (last bar). The baseline bandwidth is 177.4 GB/sec (based on the Nvidia GTX 480 chipset with Fermi architecture). The legend (N×) indicates a configuration with N times the memory bandwidth of the baseline. Perfect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

*Drop rate* is a knob to control the tradeoff between performance/energy gains and quality loss. A higher drop rate causes the core to use more predicted approximate values and avoid accessing main memory. 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 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 program execution.

## B. 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 such loads. As prior work showed [6], safety is a semantic property of the program, and language construction with programmer annotations is necessary to identify safely-approximable instructions. 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 safe-to-approximate loads.

## C. Instruction Set Architecture to Support RFVP

We extend the ISA with two new features: (1) an approximate load instruction, and (2) a new instruction for setting the drop rate.

The extended ISA has two 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. Executing an approximate load does not always invoke the predictor. 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 is triggered only when the load micro-op misses in the cache.

The *drop rate* is a knob that is exposed to the compiler to 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 do *not* initiate memory access requests, and instead trigger RFVP. When the request is not dropped, it is considered a normal cache miss, and its value is fetched from memory.

In our experiments, the drop rate is statistically identified by our compilation workflow and remains constant during program execution.

## D. Integrating RFVP into the Microarchitecture

The 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 caused by the original miss. In the microarchitecture, we use a simple Linear Feedback Shift Register (LFSR) 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 did not insert the *entire* cache line, the subsequent safe-to-approximate loads to the same cache line would produce another miss. Since RFVP never predicts nor drops *all* missing safe-to-approximate loads, the line would need to 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. Hence, our decision is to 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 on a cache line boundary (similar to [6]). Thus, we can ensure that any data value prediction does not contaminate precise load operations.

## III. Language and Software Support for RFVP

Our design principle for RFVP is to maximize the opportunities for performance and energy efficiency gains, while limiting the adverse effects of approximation on output quality.

### A. Providing Safety Guarantees

Safety is a semantic property of a program [6]. Therefore, only the programmer can reliably identify which instructions are safe to approximate. For example, EnerJ [6] provides language constructs and compiler support for annotating safe-to-approximate operations in Java. We rely on a similar technique. 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 our results confirm (shown later), RFVP still provides significant performance and energy gains because there are enough performance-critical loads that are safe to approximate.

### B. Targeting Performance-Critical Loads

After safe-to-approximate loads are identified, the next step is a profiling pass that identifies the subset of these loads that cause the largest fraction of cache misses. As prior work has shown [7], and our experiments corroborate, only a few load instructions cause most of the cache misses. In all of our GPU applications except similarityscore, at most 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 provides the opportunity to eliminate most of the cache misses. Furthermore, this focus 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.

### C. 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 final step 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, through profiling, 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 performance evaluation.

Altogether, these three 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.

## IV. 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. We use the *two-delta* stride predictor [8] due to its low complexity and reasonable accuracy as the base for multi-value prediction.[2] Empirically, the two-delta predictor provides a good tradeoff 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. However, this predictor cannot be readily used for multi-value prediction (for predicting the entire cache line), which is required for GPUs, as explained earlier.

### A. Value Predictor Design for GPUs

Here, we elaborate on the RFVP predictor design for multi-value, (*i.e.*, fill the entire cache line) 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 multiple 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 separate two-delta predictors, which is impractical.

In many GPU applications, the adjacent threads within a warp process data elements with some degree of value similarity, e.g. pixels of an image. Previous work [9] shows the value similarity between the neighboring locations in memory for GPGPU workloads. Furthermore, GPU bandwidth compression techniques (e.g., [10]) exploit this value similarity in GPGPU workloads to compress data with simple compression algorithms [11]. Our evaluation also shows significant value similarity between adjacent threads in the applications we study.

In our multi-value 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. The GPU predictor is indexed by the hash of the WarpID plus the load PC. This combination ensures the unique identification of the loads of each warp.

We design a predictor[3] that consists of *only two* specialized two-delta predictors. In order to perform the *entire cache line*[4] 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 with LRU replacement policy. As Figure 2 shows, for each row in the predictor, we keep the corresponding load's {WarpID,

[2]Each row in a two-delta predictor consists of three values: (1) the last value, (2) $\text{stride}_1$, and (3) $\text{stride}_2$. Please refer to [8] for the details of how the base two-delta predictor works.

[3]For measurements, we use a predictor that has 192 entries, is 4-way set associative, and consists of two two-delta predictors.

[4]In our GPU configuration (Table I), each cache line has 32 4-byte words.

PC} as the row tag. The load values will only be predicted if their {WarpID, PC} matches the row tag.

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

TABLE I: GPU microarchitectural parameters.

| |
|---|
| **Processor**: 700 MHz, SMs: 16, Warp Size: 32, SIMD Width: 8, 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**: 177.4 GB/sec |

**RFVP prediction mechanism.** When there is a match between {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 generates the entire cache line prediction by replicating the two predicted values for the corresponding threads. As Figure 2 shows, the Two-Delta (Th0–Th15) structure generates one prediction for threads with ThreadID=0–15, and the Two-Delta (Th16-Th31) for threads with ThreadID=16–31. Note that each of the two two-delta predictors works similarly as the baseline two-delta predictor [8]. Using this approach, RFVP is able to predict the *entire cache line* for each SIMD load access.[5]

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 later used 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 the predicted approximate values. We add a column to each two-delta predictor that tracks the last value of **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 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 the predicted approximate values (instead of random data) and avoids the 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* and 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]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 only passes 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.

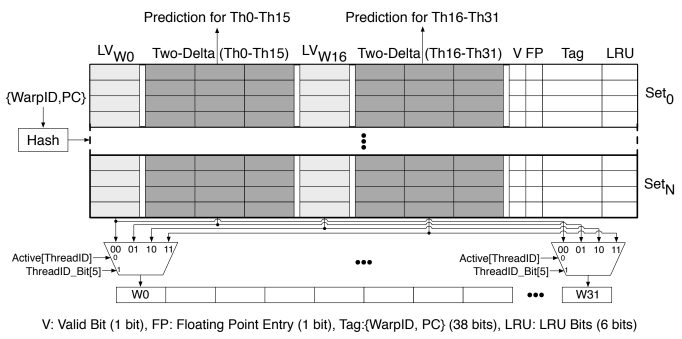


Fig. 2: Structure of the multi-value predictor for RFVP in GPUs. The GPU predictor consists of two two-delta and two last value predictors. The GPU predictor is also set-associative to reduce the conflicts between loads from different active warps. It produces predictions for full cache lines.

## V. Experimental Results

This section empirically evaluates the tradeoffs between performance, energy, and quality when RFVP is employed in a modern GPU. We use the cycle-level GPGPU-Sim simulator version 3.1 [12]. 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 (Table I) that closely models an Nvidia GTX 480 chipset (Fermi architecture). For each benchmark, we use 10 different inputs to evaluate RFVP. We define application-specific quality metrics to assess the quality of each application's output with RFVP. For image applications (fastwalsh, gaussian), we use image difference Root-Mean-Square Error (RMSE). For heartwall and particlefilter, we use average displacement. For backprop and stringmatch, we use average relative error and mismatch rate, respectively. In all other applications, we use RMSE as the quality degradation metric.

**Performance, energy, memory bandwidth, and quality**. Figure 3a 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 3b and 3c illustrate the energy reduction and the reduction in off-chip bandwidth consumption, respectively.

As Figures 3a and 3b 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 instructions 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 applications, including fastwalsh, particlefilter, similarityscore, s.srad2. In particular, the benefits are as high as 22% speedup and 20% energy reduction for particlefilter with strictly less than 1% quality loss.

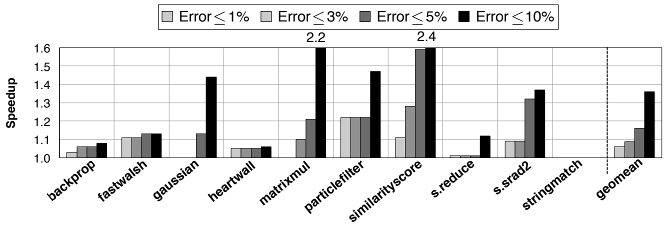
Comparing Figures 3a, 3b, and 3c shows that the benefits strongly correlate with the reduction in bandwidth consumption. This strong correlation suggests that RFVP is able to significantly improve both GPU performance and energy consumption by predicting load values and dropping memory access requests. 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 3c 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 (when all the memory accesses are L1 cache hits). Therefore, the benefits from RFVP are predictably limited even with significant bandwidth reduction. This case shows that the applications' performance sensitivity to off-chip communication bandwidth is an important factor in RFVP's ability to improve performance and energy efficiency.

Also, Figure 3 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 upcoming paragraphs. 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 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.

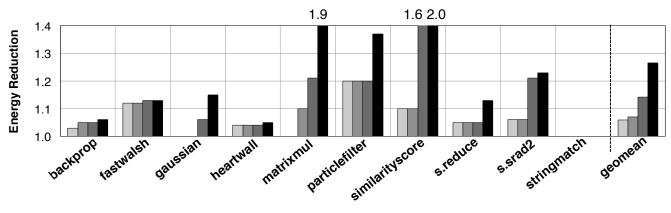
All applications but one benefit considerably from RFVP due to reduced off-chip communication. Particularly, the energy benefits are due to reduced runtime and fewer costly data fetches from off-chip memory. Overall, these results confirm the effectiveness of rollback-free value prediction in mitigating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

**Quality tradeoffs with drop rate**. Drop rate is RFVP's knob for navigating the quality tradeoffs. It dictates what percentage of the approximate load cache misses to value-predict and drop. 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%.

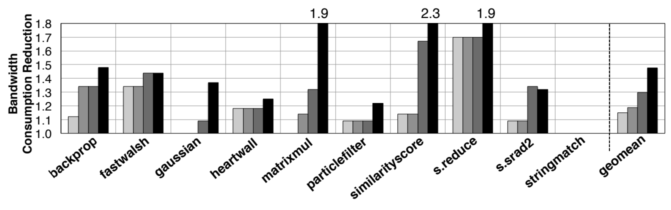
Figure 4 illustrates the effect of drop rate on speedup (Figure 4a), energy reduction (Figure 4b), and quality degradation (Figure 4c). As the drop rate increases, so do the performance and energy benefits. However, the benefits come with some cost in output quality. The average speedup ranges from 1.07× with a 12.5% drop rate, to as much as 2.1× with a 90% drop



(a) Speedup



(b) Energy Reduction



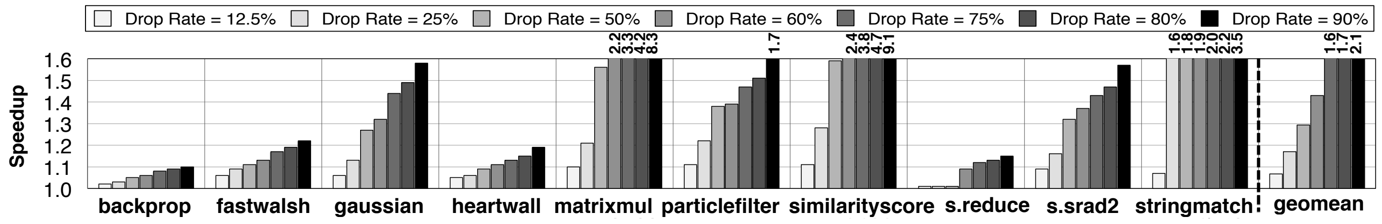
(c) Memory Bandwidth Consumption Reduction

Fig. 3: RFVP's (a) performance improvement, (b) energy reduction, and (c) memory bandwidth consumption reduction for at most 1%, 3%, 5%, and 10% quality degradation in a modern GPU (described in Table I).

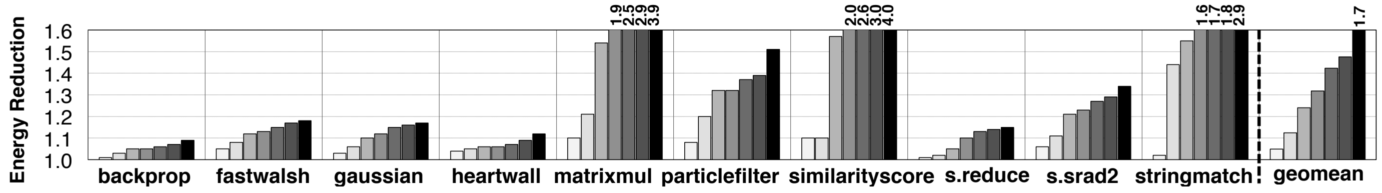
rate. Correspondingly, the average energy reduction ranges from 1.05× to 1.7× and the average quality degradation ranges from 6.5% to 31%.

Figure 4c 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. Its input data set contains only English words with very low value locality. Furthermore, the application outputs is 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 found 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.

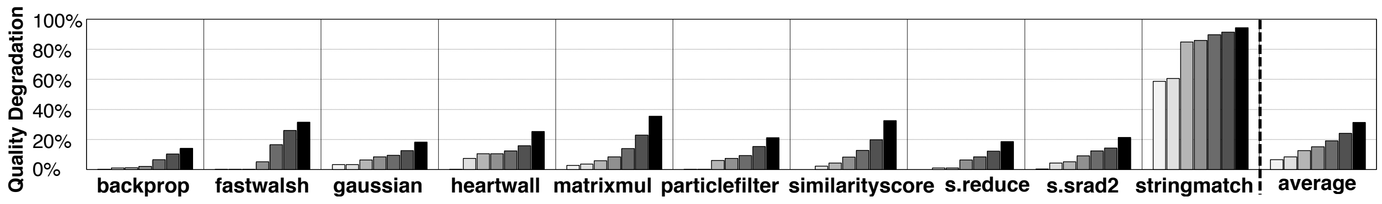
As Figure 4 shows, each application tolerates the effects of RFVP approximation differently. For some applications, such as gaussian and fastwalsh, as the rate of approximation (drop rate) increases, speedup, energy reduction and quality loss gradually increase. In other applications such as matrixmul and similarityscore, the performance and energy benefits increase sharply while the quality degradation increases gradually. For example, in similarityscore, increasing the drop



(a) Speedup



(b) Energy Reduction



(c) Quality Degradation

Fig. 4: Exploring (a) speedup, (b) energy reduction, and (c) quality trade-offs with different drop rates.

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, for applications whose performance is most sensitive to memory bandwidth (see Figure 1).

## VI. Related Work

To our knowledge, this paper 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 line.

Below, 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 paper defines a new technique that mitigates the memory subsystem bottlenecks of long access latency and limited off-chip bandwidth.

Some of the existing techniques include (a) loop perforation [1], (b) computation substitution [9], (c) precision scaling [6], and (d) approximate circuit synthesis [3]. Most of these techniques (1) operate at the coarse granularity of a loop body or a function call, (2) are agnostic to and unaware of micro-architectural events, and (3) are explicitly invoked by the code. In contrast, rollback-free value prediction (1) operates at the fine-granularity of a single load instruction, (2) is triggered by microarchitectural events, and (3) does not require direct and explicit runtime software invocation.

**Value prediction.** RFVP takes inspiration from prior work that explores exact value prediction [8]. However, our work fundamentally differs from traditional value prediction techniques because it does not check for mispredictions and does not recover from them.[6]

**Load value approximation.** In [15], we introduced the RFVP technique for conventional CPU based systems to lower the effective memory access latency. Later, in a concurrent effort [16], San Miguel et al. proposed a technique that uses value prediction without checks for misprediction to address the memory latency bottleneck in CPU based systems. This work differs from our previous [15] and the concurrent [16] work as follows: (1) we specialize our techniques for GPU processors, targeting mainly the memory bandwidth bottleneck, (2) we utilize the value similarity of accesses across adjacent threads in GPUs to develop a low-overhead multi-value predictor for an entire cache line, and (3) we drop a portion of cache-miss load requests to fundamentally reduce the memory bandwidth demand in GPUs.

## VII. Conclusions

This paper introduces Rollback-Free Value Prediction (RFVP) and demonstrates its effectiveness in tackling two major memory system bottlenecks–limited off-chip bandwidth (*bandwidth wall*) and long memory access latency (*memory wall*), with a focus on GPU-based systems. RFVP predicts

[6]Note that a previous work by Zhou and Conte [13] and use of value prediction in runahead mode [14] are recovery-free value predictors, but they explore purely speculate nature of execution as apposed to error tolerance of programs.

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. The drop rate is a knob that controls the tradeoff between quality of results and performance/energy gains. Our extensive evaluations show 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%. 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.

## VIII. Acknowledgements

We thank anonymous reviewers of IEEE D&T as well as past anaonymous 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

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

[2] 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.

[3] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, J. Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan, "Axilog: Language support for approximate hardware design," in *DATE*, 2015.

[4] Y. Luo, S. Govindan, B. P. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *DSN*, 2014.

[5] S. W. Keckler, W. J. Dally, B. Khailany, M. Garland, and D. Glasco, "GPUs and the future of parallel computing," *IEEE Micro*, no. 5, 2011.

[6] 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.

[7] 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.

[8] R. J. Eickemeyer and S. Vassiliadis, "A load-instruction unit for pipelined processors," *IBM Journal of Research and Development*, 1993.

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

[10] N. Vijaykumar, G. Pekhimenko, A. Jog, A. Bhowmick, R. Ausavarungnirun, C. Das, M. Kandemir, T. C. Mowry, and O. Mutlu, "A case for core-assisted bottleneck acceleration in GPUs: enabling flexible data compression with assist warps," in *ISCA*, 2015.

[11] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: practical data compression for on-chip caches," in *PACT*, 2012.

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

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

[14] O. Mutlu, H. Kim, and Y. N. Patt, "Address-value delta (AVD) prediction: Increasing the effectiveness of runahead execution by exploiting regular memory allocation patterns," in *MICRO*, 2005.

[15] 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 *PACT*, August 2014.

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