Load Value Approximation

Joshua San Miguel Mario Badr Natalie Enright Jerger
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
University of Toronto, Toronto, Canada
{*joshua.sanmiguel, mario.badr*}*@mail.utoronto.ca enright@ece.utoronto.ca*

***Abstract*—Approximate computing explores opportunities that emerge when applications can tolerate error or inexactness. These applications, which range from multimedia processing to machine learning, operate on inherently noisy and imprecise data. We can trade-off some loss in output value integrity for improved processor performance and energy-efficiency. As memory accesses consume substantial latency and energy, we explore *load value approximation*, a microarchitectural technique to learn value patterns and generate approximations for the data. The processor uses these approximate data values to continue executing without incurring the high cost of accessing memory, removing load instructions from the critical path. Load value approximation can also inhibit approximated loads from accessing memory, resulting in energy savings. On a range of PARSEC workloads, we observe up to 28.6% speedup (8.5% on average) and 44.1% energy savings (12.6% on average), while maintaining low output error. By exploiting the approximate nature of applications, we draw closer to the ideal latency and energy of accessing memory.**

I. Introduction

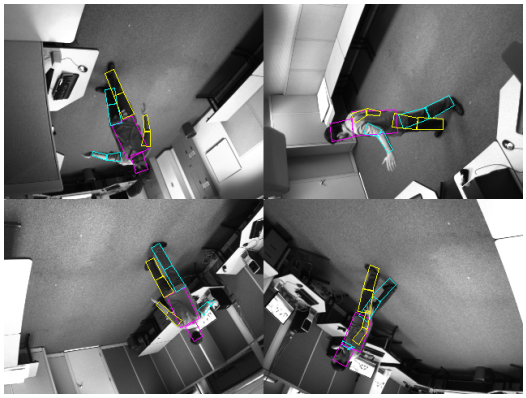
Approximate computing is an emerging paradigm for energy-efficient processor design. A wide range of commercial, multimedia and scientific applications are inherently *approximate*. They operate on noisy data and perform inexact computations. These applications – which range from image processing to recognition and mining applications [10, 25] – can tolerate some error in their output values. This allows architects to trade-off data value integrity for better performance and energy savings. Recent work approximates computations [1, 14], relaxes synchronization [33, 40] and efficiently stores approximate data [22, 36]. Our work exploits approximate applications to reduce both the latency and energy of accessing data in the memory hierarchy.

We propose *load value approximation*. Since many applications can tolerate inexactness, the values associated with cache misses can be approximated. In traditional processors, upon a load miss in the L1 cache, the data must be retrieved from the next-level caches or from main memory. Cache access combined with the long latency of traversing the network-on-chip (NoC) results in many cycles between the request and receipt of data by the processor. To overcome this latency, we use a load value approximator, a hardware mechanism that estimates memory values. By approximating the load value on a cache miss, the processor can immediately proceed without waiting for the cache response.

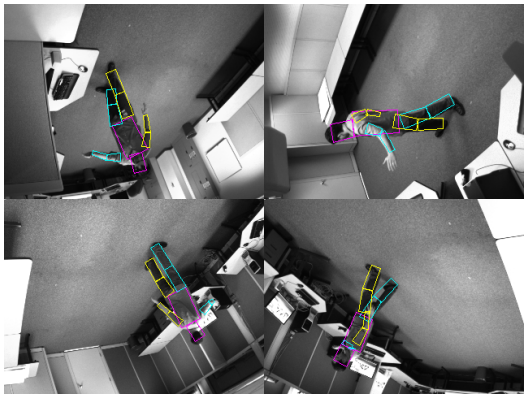
Load value approximation follows from work on load value prediction [6, 7, 16, 20, 21]. Applications exhibit value locality; they tend to reuse common values. This is typically due to runtime constants and redundancy in real-world input data [20]. Load value prediction introduces complexity due to supporting speculative values and performing rollbacks upon mispredictions. Our work targets applications that exhibit *approximate value locality*. Since inexactness is acceptable, rollbacks are eliminated; there is no need to re-execute a load instruction if the approximator generates a value that does not exactly match the value in memory. We show that without rollbacks, load value approximation still maintains low error in the final application output.

In addition, load value approximation allows for *relaxed confidence estimation*. In traditional value predictors, predictions are made only if confidence is high to minimize rollbacks. The confidence of a predictor increases if its predictions frequently match the actual values in memory. High confidence requirements limit the coverage of a value predictor; predictions that are not exact but are approximately close to the actual values are deemed mispredictions, thus decreasing confidence. Load value approximation achieves greater coverage by employing relaxed confidence windows; approximators can continue to generate values even if they are not exact, as long as they fall within some acceptable range. Relaxed confidence windows create a *performance-error* tradeoff for approximators.

Furthermore, load value approximation decouples the fetching of a cache block from the cache miss itself. Conventionally, an L1 cache miss results in fetching the block from the next-level cache or main memory. The processor cannot proceed until it receives the data. However, with load value approximation, the approximator generates a value so the processor can continue immediately. As a result, it is no longer necessary to fetch the block from memory. Note that this is fundamentally impossible with traditional value prediction since blocks must always be fetched to validate the correctness of the prediction. Although blocks are still fetched periodically to train the approximator, load value approximation eliminates the one-to-one ratio of cache misses to cache fetches. We refer to this ratio as the *approximation degree*. By foregoing the fetching of blocks upon cache misses, load value approximation saves substantial energy in the memory hierarchy through this *energy-error* tradeoff.



(a) Precise execution.



(b) Execution with load value approximation.

Figure 1: Output of bodytrack.

Contributions. Our work identifies and exploits the following novel properties of load value approximation:

- Simplified implementation relative to traditional value prediction due to the elimination of speculation and rollbacks while still reaping performance benefits;
- Relaxed confidence estimation to facilitate trade-offs along a performance-error spectrum;
- Approximation degree enabling trade-offs along an energy-error spectrum.

We extend our preliminary work on load value approximation [37] with an extensive approximator design space exploration and a detailed evaluation of performance and energy. We find that load value approximation can achieve up to 28.6% speedup (8.5% on average) and 44.1% energy savings (12.6% on average) with low output error; Figure 1 shows that the difference in application output with and without approximation is nearly indiscernible.

II. Load Value Prediction

The presence of value locality [20] has led to significant research in load value predictors [6, 7, 16, 21]. In these schemes, a load miss in the L1 cache still fetches the data from the next level of memory. However, instead of waiting for the data, the predictor generates a value and allows the processor to continue executing instructions speculatively. When the data arrives, the prediction is validated against the actual value. If they do not match, the processor must rollback the speculatively executed instructions.

When the predicted value is correct, the predictor increases its *confidence* for that value and the processor has avoided the cache miss latency. If the predicted value is incorrect, confidence is lowered and the processor must rollback and recompute with the actual value. Implementing such a scheme is complex, and requires managing speculative values while risking costly rollbacks for inaccurate predictions. Due to the long latencies of cache misses, processors need large buffers to store all speculative values since they must be validated later. Upon a misprediction, the processor must be able to quickly restore its registers and undo all speculative modifications to memory, either in the store queue or the L1 cache. Recent work re-examines the complexities associated with value prediction to move towards more practical implementations [30, 31]; this work improves confidence estimation and addresses timing issues for back-to-back occurrences of the same instruction. We address both of these issues in Section III. Load value prediction typically performs poorly for floating-point values [16]. Since predicted values need to be identical to the actual values, even small variations in floating-point precision result in costly rollbacks; we explore reduced precision for floating-point values in Section VII. Finally, in multiprocessors, it is possible for another thread to modify a speculative value, resulting in complications with the memory consistency model [24]; we defer our discussion of multiprocessor issues for load value approximation to Section VII. Many of the challenges associated with implementing load value prediction can be mitigated by taking advantage of the approximate nature of applications.

III. Load Value Approximation

Load value approximation estimates data values to eliminate the latency and energy of retrieving the data from memory. Figure 2 shows an overview of load value approximation. When a load X misses in the L1 data cache ❶, the load value approximator generates *X_approx* ❷.[1] The processor assumes this is the actual value of X and proceeds with its execution ❸a. A request is sent to the next level of the memory hierarchy to fetch the cache block containing *X_actual* ❸b. This request is off the critical path of the application's execution; the processor does not need to wait for the actual data. *X_actual* is then used to train the approximator for better accuracy ❹.

Section III-A describes the generation of approximate values ❷. To minimize error in the application's final output, we use confidence estimation to selectively make approximations only when the approximator is sufficiently accurate.

[1] Section IV describes how we select which loads to approximate.

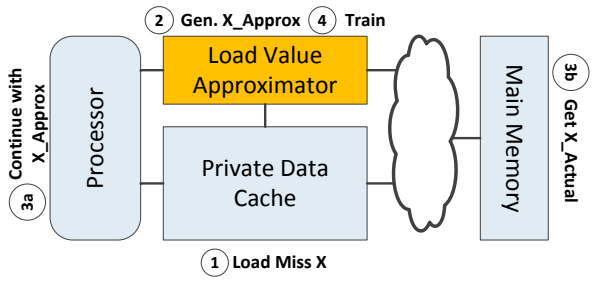


Figure 2: Load value approximation overview.

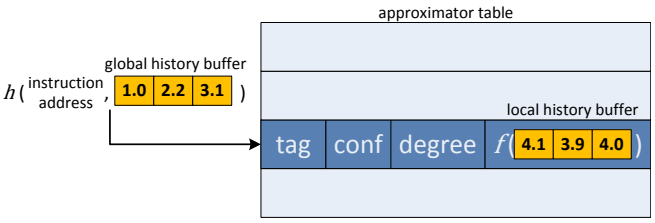


Figure 3: Approximator design.

In Section III-B, we relax confidence estimation to achieve high coverage; coverage is the percentage of loads that are approximated. Section VI shows that high coverage leads to large reduction in MPKI. Load value approximation can save energy by decoupling cache fetches ❸b from cache misses ❶. In conventional caches, a miss always fetches the block from the next-level in the memory hierarchy. However, with load value approximation, the fetch is optional; fetching trains the approximator ❹. By foregoing the fetch, we can trade-off approximator accuracy for energy savings in the memory hierarchy. Section III-C describes how we accomplish this by varying the approximation degree.

A. Approximator Design

Traditional value predictors use the values of previous loads to generate a prediction. Sazeides and Smith categorize value predictors as either computational or context-based [38]. Computational predictors take the previous values of a given load instruction and generate a prediction by computing some function, such as the stride between the values. Context-based predictors generate predictions by finding patterns in the value history. This taxonomy also applies to load value approximators.

We explore the design space of approximators that combine the computational and context-based ideas into a single hardware structure. Figure 3 shows the general structure of our load value approximator. The approximator consists of a global history buffer (GHB) and an approximator table. The GHB is a FIFO queue that stores the values accessed by the most recent load instructions. This provides the global context, which can improve accuracy by incorporating global control path information [27]. GHB entries are not approximate values but, rather, the precise values as loaded from memory. Specifically, all *X_actual* (Figure 2) values accessed by a processor are pushed into the GHB as part of the training process of the approximator ❹.

Upon a cache miss, to generate an approximate value, the values in the GHB are hashed together – using some hash function h (e.g., XOR) – along with the instruction address of the load. This hash value forms the context and represents a load value pattern. This hash value indexes the direct-mapped approximator table. Each entry in the table consists of a tag (the GHB hash value), a saturating confidence counter (Section III-B), a degree counter (Section III-C) and a local history buffer (LHB). Unlike the GHB which tracks all loads, the LHB stores the values accessed only by the previous loads that match the entry's tag. As a result, the LHB tracks the load values that immediately followed the GHB value pattern.

The example in Figure 3 shows that the last three loads issued by the processor had *X_actual* values of 1.0, 2.2 and 3.1. This value pattern maps to an entry in the approximator table. The LHB values (4.1, 3.9 and 4.0) are the *X_actual* values of the three previous loads that have come immediately after this value pattern in the past. An approximate value is then generated by employing some computation function f (e.g., average) on the values in the LHB.

Due to their fine precision, floating-point values have low *exact predictability*. For example, if a predictor generates the value 1.000 but the actual value in memory is 1.001, then a rollback is required since the values do not match exactly. To address this, predictors such as the finite context method [38], typically employ a selection mechanism that decides which of the values in the LHB is most likely to match the actual value. Load value approximation overcomes this challenge since it is not concerned with exact predictability; we simply compute the average of all the values for use as an estimate. Furthermore, it is difficult to achieve high coverage for floating-point values. Since GHB values are hashed together to form the index to the approximator table, small variations in precision result in similar values (such as 1.000 and 1.001) being mapped to different table entries. To address this, we can reduce the number of mantissa bits in the GHB, effectively improving floating-point value locality [1]. In our evaluations, we show that our approximator design achieves better coverage than traditional value predictors while maintaining low error.

Rollbacks. With traditional value predictors, *X_approx* is speculative and must be validated against *X_actual* ❹. If they are not identical, the processor must rollback its execution and repeat the load instruction. Load value approximation eliminates rollbacks since it does not need to validate the correctness of its approximations. Thus cache blocks are only fetched to improve the accuracy of the approximator ❹.

Value Delay. As in value prediction, an important challenge in implementing approximators is value delay [49]. Value delay occurs when the *X_actual* values of previous loads

have not yet arrived at the L1 cache and thus have not been inserted into the history buffers. The approximator must generate an estimate based on older (potentially stale) values in the GHB and LHB, limiting its accuracy. For example, consider a load miss on *A* that results in a value prediction. The processor saves its state and enters speculative execution. Now if we encounter a load miss on *B*, the prediction for *B* may be less accurate since the predictor has not yet been trained with the actual value of *A*. Value predictors typically only save the processor state on the first load miss (*A*) to keep complexity low [7]. Thus if *B* is mispredicted – which is likely to happen due to value delay – the processor rolls back to *A*, even if *A* was predicted correctly.

Value delay is more tolerable for load value approximation; stale values may affect the accuracy of the approximator but would not trigger rollbacks. In Section VI-C, we show that our approximators are resilient to high value delays. This enables load value approximation to exploit energy optimizations when fetching blocks into the L1 cache. Since approximation takes load misses off the critical path, low-energy techniques – using heterogeneous NoCs [26] and memory modules [32] – can be employed to fetch of approximate data. These types of energy saving techniques often sacrifice fetching latency; however, we can tolerate this since our approximators are resilient to high value delays.

B. Relaxed Confidence Estimation

Traditional value predictors use confidence estimation to improve accuracy and minimize costly rollbacks; however, confidence estimation typically decreases coverage. Conversely, approximators can employ relaxed confidence estimation to achieve high coverage while keeping error low. The extent to which approximation can be tolerated is called the *relaxed confidence window*. This window defines how close *X_approx* must be to *X_actual* to increment the confidence counter. For example, a confidence window of 10% means we increment the confidence counter only if *X_approx* is within $\pm$10% of *X_actual*. Traditional value prediction employs a confidence window of 0%, since the two values must be identical.

Each approximator table entry contains a saturating (signed) confidence counter. Upon a cache miss, if the counter is greater than or equal to 0, we make an approximation. When the block arrives at the cache, the approximator is trained; *X_actual* is pushed into both the LHB and GHB. The confidence counter is incremented by one if *X_approx* is approximately close enough to *X_actual* (decremented by one otherwise). The confidence counter could be adjusted by more than one depending on how far off the approximation is; this feature cannot be used in traditional value prediction since its correctness is a binary property. We leave the study of this optimization for future work.

C. Approximation Degree

A key feature of load value approximation is its ability to forego the fetching of blocks. In conventional caches, the ratio of fetches to misses is 1:1. This ratio also holds true for traditional value prediction. Even though value predictors allow the processor to continue executing upon a cache miss, the data block must still be fetched to validate the correctness of the prediction.

Load value approximation does not enforce precise execution. For this reason, upon a cache miss, if an approximation is made, it is possible to not fetch the data block at all. Recall in Figure 2 that *X_actual* serves no other purpose than to train the approximator for better accuracy ❹. Thus we can forego fetching a block by reusing the same approximate value multiple times. This effectively trades off approximation accuracy for better energy efficiency in the memory hierarchy. We control this trade-off through the *approximation degree*. The approximation degree specifies how many times we reuse a value generated by the approximator before we train it. For example, an approximation degree of 4 means that when a load misses in the cache, the approximator generates a value that will be reused for the next 4 load misses (in addition to the current miss). As a result, the data block does not need to be fetched until the last miss, since only then will the approximator be trained.

To implement the approximation degree, each entry in the approximator table contains a degree counter, shown in Figure 3. The degree counter is initialized to the maximum approximation degree and is decremented every time an approximation is made using this entry. Upon a cache miss, if an approximation is made and the counter is greater than 0, the data block is not fetched. This implies that the next approximation from this entry will return the same value. The cache block is only fetched if the degree counter is equal to 0, after which the entry is trained (both the LHB and GHB are updated with the fetched value) and the degree counter is reset to the maximum approximation degree.

Prefetching is an effective and widely used technique for reducing the cache miss rate. However, this reduction typically comes at the expense of more fetches, wasting energy when prefetching useless blocks. This trade-off is controlled by the *prefetch degree*. The prefetch degree specifies how many extra blocks to fetch on a cache miss. For example, a prefetch degree of 4 means that when a load misses in the cache, the prefetcher fetches up to 4 more blocks (in addition to the missed block) to try to anticipate the next 4 load misses. This results in a 5:1 ratio of fetches to misses.

The approximation degree effectively yields the inverse fetch-to-miss ratio as the prefetch degree. A prefetch degree of 4 yields 5:1 fetches to misses, while an approximation degree of 4 yields a ratio of 1:5. In Section VI-D, we compare load value approximation to prefetching. Though both techniques can reduce load miss rates, load value approximation saves substantial energy by minimizing the number of blocks fetched into the L1 cache.

IV. Identifying Approximate Data

Programming and ISA support have been proposed for identifying approximate computations [2, 13]. For example, the EnerJ framework allows Java programmers to declare data as either precise or approximate [35]. We leverage this prior work by annotating data that can use load value approximation; we make use of ISA extensions to indicate which loads are approximate. In this section, we discuss which regions of data can be approximated for a variety of applications from PARSEC 3.0 [3], which represent a broad range of modern and emerging application domains.

Load value approximation requires programmers to annotate their code. There are certain conditions where data that can be approximated, should not be. In addition, it is not necessary to approximate all data. In this section, we discuss guidelines and recommendations programmers should adhere to when annotating their code.

Control Flow: In general, one should not approximate data that directly affects an application's control flow. Approximating the value x for an application that contains the code $if(x == 42)$ will likely result in incorrect behaviour. However, restricting programmers from annotating all data that affects control flow is conservative [35]. For example, a condition that leads to a region of code that further updates approximate data can be approximated itself – as data affected by the condition is approximate, the evaluation of the condition may not significantly change the data values.

Divide-By-Zero: Data used in the denominator of a division operation should not be approximated. An approximation of zero will cause the program to crash; dividing by very small values will result in a much larger number than expected.

Memory Addresses: Memory addresses and pointers should never be approximated. Approximating memory addresses in an application that relies on pointer chasing, for example, could have catastrophic results.

Common Case: Programmers should focus their attention on expensive loops and functions rather than obscure and rarely visited corner cases. Identifying approximate data in frequently visited regions of code is the ideal scenario. This removes cache misses from the critical path and allows the application to continue rapidly with approximated data, rather than flooding the network with data requests. Profiling tools can be use to determine where the bulk of cache misses occurs. We use profiling and previous work [39] with PARSEC, to find frequently visited regions of code and then determine if the data being used can be approximated.

The guidelines above may apply to certain regions of code but not others. However, the data being used can span several functions and classes. Therefore, simply identifying data as approximate for the entire application may be unwise. It is beneficial to identify data as approximate for only small regions of code (such as the common case). We use this strategy for several benchmarks, which we discuss below.

A. Benchmarks

Canneal is a kernel of a CAD application that uses simulated annealing to place blocks on a two dimensional grid. These blocks are interconnected via *nets*; the annealer attempts to minimize the routing cost by randomly swapping blocks and recalculating the cost. Cost is a function of the distance between all blocks connected to (fan in) and from (fan out) a given block. A naïve approximation would annotate every block's <x,y> values. However, this is infeasible due to the number of blocks; it is also unnecessary. Significant load misses come when calculating the cost of the swap. As a result, we choose to annotate the integer <x,y> values only for cost functions. That is, we approximate the <x,y> values of blocks that fan in or out from the block in question. The output error for canneal is the difference between the final routing cost for the approximated and exact execution. The algorithm itself is a heuristic and inherently approximate, therefore small errors are tolerable.

Both **blackscholes** and **swaptions** are financial analysis applications that compute partial differential equations to determine the prices of a portfolio. In both cases, the input data consists of arrays of floating-point values that we target for approximation. The input set contains a lot of redundant values; for example, an underlying asset's current price in blackscholes' simlarge input set takes on four possible values, two of which occur over 98% of the time. Moreover, the input values are used repeatedly throughout computation but are not updated. Rather, they are copied to other variables which are updated separately. This makes the input values an ideal candidate for approximation. When the data set is larger than the cache size, approximation is a useful technique for reducing cache misses and improving performance. Upon completion, the benchmarks output a list of prices. In swaptions, we compare the error of each approximated price to its precise price, and then take an average where all prices are weighted equally. In blackscholes, the error is calculated as the percentage of prices with an error greater than 1%. Errors in options pricing are tolerable [17]. For example, blackscholes has *Black's approximation* to quickly determine an estimate of an option's price [4].

Bodytrack is a computer vision application that takes four camera feeds as its input and tracks a human body. It uses an annealed particle filter, which samples and resamples particles to estimate if a particular distribution is a body part. A likelihood function determines a particle's weight, which influences the quality of this estimation. The likelihood computation occurs every time step [12], which involves two error calculations that are performed in long loops. The **x264** benchmark encodes raw data into a H.264 format. Frames are divided into blocks, and the algorithm looks for previously encoded frames that are similar to the current block to estimate motion. This portion of code is frequently visited [39]. In both benchmarks, the approximated data are integer values of pixels. For bodytrack, we approximate the

value from the image map at an (x,y) point.

Bodytrack outputs vectors representing the position of the bodies. We perform a pair-wise comparison of the vectors from precise computation to approximate computation to determine the error. Figure 1 shows a visualization of the error as the actual image output (7.7% output error). For x264, we compare the peak signal-to-noise ratio and bit rate, weighted equally.

Ferret takes an image query and searches for similar images in its database. Images are divided into segments; a feature vector of floats describes each segment. We approximate these feature vectors. The benchmark calculates the distance between these segments to determine which images in the database are most similar to the query. We calculate the error by first determining the intersection of the images found by the approximate run to the precise search results, and then divide the size of this set by the size of the precise set of search results [39]. Note that this is a conservative error metric. Several images can satisfy a query; since the search algorithm is inherently approximate, only a subset of these images may be returned. When approximating, images may be returned that were not in the precise subset, despite satisfying the query. As a result, our error measurements for ferret are pessimistic.

Fluidanimate simulates a fluid in a volume so that its surface can be animated in interactive applications (such as video games). The benchmark uses particles to model the state of the fluid. A particle's movement depends on the density fields of other particles, which are updated at every time step. Particles are partitioned into cells so that only particles in the current and neighbouring cells are considered when updating densities. We approximate the data of each particle (e.g. location, density, etc) during the regions of code that calculate densities and determine particle acceleration. The final error is calculated as the percentage of particles that are in the same cell as they are in with precise execution.

V. Methodology

We employ a two-phase methodology to evaluate load value approximation. The first phase is a wide design space exploration of different approximator implementations (Section V-A). The second phase measures the performance and energy of load value approximation in full-system multiprocessor simulation (Section V-B). We perform our evaluations on applications from PARSEC 3.0 [3], each configured with 4 threads.

A. Design Space Exploration

First, we explore the approximator design space using Pin, a dynamic binary instrumentation framework [23]. We leverage the cache simulator in Pin to model 64 KB private L1 data caches. We then implement our approximators alongside the caches and compare them against value predictors and prefetchers. Benchmarks are run with the simlarge input set.

Benchmark	L1 MPKI	Instruction count variation
blackscholes	0.93	0.99%
bodytrack	4.93	0.05%
canneal	12.50	1.25%
ferret	3.28	0.60%
fluidanimate	1.23	0.17%
swaptions	4.92E-05	0.00%
x264	0.59	2.37%

Table I: Precise L1 MPKI and variation in dynamic instruction count when employing load value approximation.

Our Pin simulator catches all load instructions that access approximate memory locations, as annotated in Section IV. Blackscholes, ferret, fluidanimate and swaptions approximate floating-point data, while bodytrack, canneal, and x264 approximate integer data. We then directly clobber the return values of these loads with our approximated values, dynamically altering the execution of the application. This allows us to measure the error due to approximation in the final outputs of the applications. We utilize the error metrics described in Section IV. All of our error results pertain to the application's final output, not the individual error of each load instruction. In approximate computing, output error below 10% is generally acceptable [36, 41].

Pin allows us to rapidly evaluate performance, energy and output error across the vast approximator design space. For performance, we measure the total load misses per kilo-instructions (MPKI), normalized to that of the precise execution without any approximations. The L1 MPKI of precise execution is shown in Table I. In our evaluation, MPKI incorporates both precise and approximate data. We consider an approximated value to be a cache hit since the value is immediately available to the processor; thus even though the value still needs to be fetched from memory, effective MPKI is reduced. For energy, we use reduction in L1 cache line fetches as a first-order approximation of energy savings. These metrics serve as proxies for estimating the performance and energy of load value approximators in our design space exploration. All measurements are averaged from 5 simulation runs. Our current work focuses on load misses; store misses are generally off of the critical path of execution [19]. Table I also lists the variation in dynamic instructions executed when employing approximation; variation is low across all workloads.

B. Full-System Simulation

In the second phase of our methodology, we use full-system simulation to evaluate performance and energy savings for load value approximation after having pruned the design space with Pin simulations. We use FeS2, a cycle-level x86 simulator [28], configured as in Table II. We use BookSim [18] to model the network-on-chip. All applications execute up to one billion user-mode instructions. Note that the L1 cache size is scaled down; our full-system simulations use simmedium input sets instead of simlarge.

Full System Configuration

Processor	4 IA-32 cores, 2 GHz, 4-wide OoO, 32-entry ROB
Private L1 cache	16 KB, 8-way, 1-cycle latency, 64 B blocks
Shared L2 cache	512 KB distributed, 16-way, 6-cycle latency
Main memory	1 GB, 160-cycle latency
Cache coherence	MSI protocol
Network-on-chip	2×2 mesh, 3-cycle routers
Technology node	32 nm

Baseline Approximator Configuration

Approximator table	512 entries
Confidence bits	4 [min -8, max 7]
Confidence window	+/- 10%
Context hash function	XOR(PC, GHB)
Global history buffer	0 entries
Computation function	AVERAGE(LHB)
Local history buffer	4 entries
Tag bits	21
Value delay	4 load instructions
Approximation degree	0

Table II: Configuration parameters used in evaluation.

For energy, we use CACTI [42] to measure the dynamic energy consumption of the caches, main memory and the approximator tables; overheads of the approximator tables are factored into the energy results presented.

VI. Evaluation

In this section, we explore several design considerations for load value approximators: global history buffer size (Section VI-A), relaxed confidence thresholds (Section VI-B), value delay (Section VI-C) and approximation degree (Section VI-D). Our design space exploration is based on three metrics: misses-per-kilo-instructions (MPKI), blocks fetched into the L1 cache (fetches), and output error. We then evaluate performance and energy of load value approximation with full-system simulation (Section VI-E).

We compare load value approximation (LVA) to load value prediction (LVP) and prefetching. For all evaluations, our LVP implementation is *idealized*; for LVP, we assume that a value is correctly predicted as long as one of the values in the LHB matches the precise value in memory. A typical predictor uses a mechanism to select the value most likely to match the precise value in memory – the selection could be wrong even though the correct value is in the LHB. We assume a perfect selection mechanism for LVP to represent an upper bound on LVP's ability to reduce MPKI.

Table II details our baseline configuration for LVA. The local history buffer (LHB) has four entries; LVP has four values that can match the precise value in memory. To generate an approximation, LVA takes an average of the four entries in the LHB. We tried different LHB functions such as strides and deltas and found average to be most accurate; exploration of more complex functions is left for future work. We assume a relaxed confidence window of ±10% only for floating-point data. We do not employ confidence for integer data (unless otherwise noted). We use

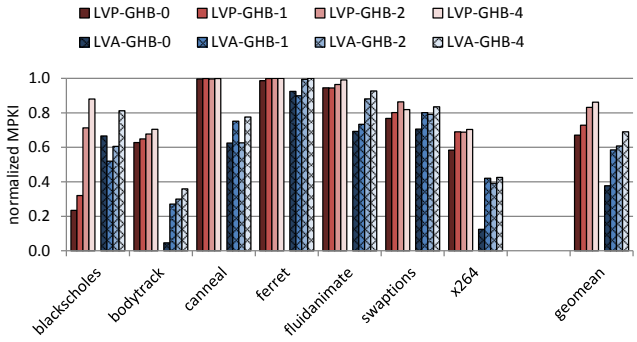


Figure 4: LVA vs. an idealized LVP for different GHB sizes.

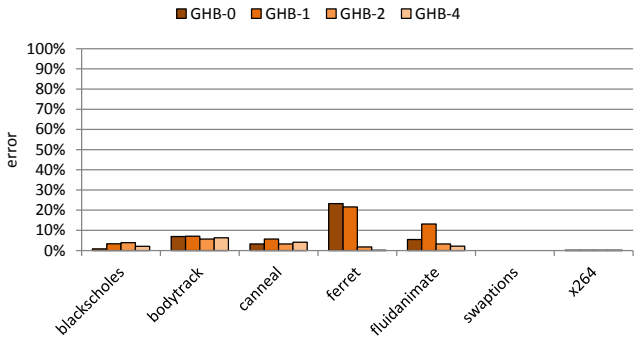


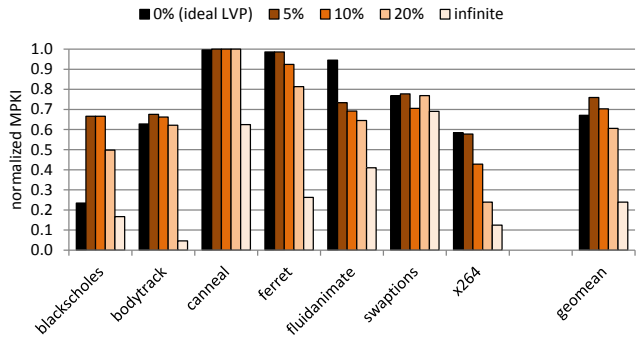
Figure 5: Output error of LVA for different GHB sizes. Note the near zero error for swaptions and x264.

512-entry approximator tables; to reduce hardware overhead, Section VII-A shows that even smaller tables are effective.

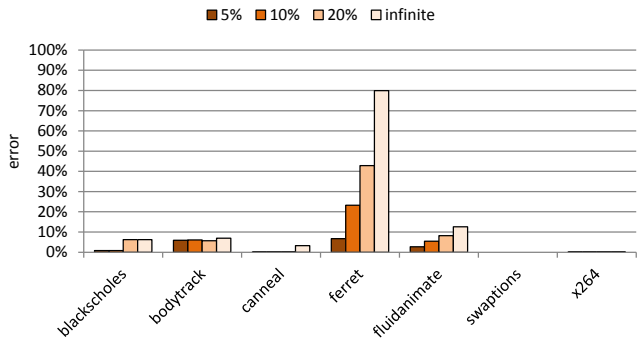
A. The Effects of History

Different numbers of values stored in the global history buffer (GHB) affect how loads index the approximator. Figure 4 compares our baseline LVA with LVP for varying GHB sizes. On average, LVA achieves lower MPKI than LVP because we are not restricted to exact predictability. By tolerating some error (confidence window of ±10%), LVA still generates accurate approximations by taking the average of the values in the LHB (without the need for a selection mechanism). Figure 5 shows the impact of GHB size on output error. Output error is around or below 10% for all applications except for ferret, whose output error tends to be pessimistic (as discussed previously in Section IV).

MPKI tends to increase with the GHB size. Hashing more GHB values generates more unique values to index the approximator table. This is particularly challenging for floating-point values (e.g., fluidanimate) since their fine precision reduces the redundancy of floating-point values that we encounter (we show how to improve this in Section VII-B). Though we use 512-entry approximator tables, larger tables can be used to reduce destructive aliasing by allowing similar floating point values to map to different entries. However, LVA performs well even when using only



(a) MPKI.



(b) Output Error.

Figure 6: Performance and error for varying approximator confidence windows.

the PC to index into the approximator table (GHB 0), demonstrating that our approximator need not be complex to maintain low output error. In fluidanimate, LVA is accurate with GHB 0, correlating a single property (e.g. position) of a particle with its neighbouring particles. With larger GHB sizes, LVA attempts to correlate different properties (e.g. position, acceleration, density); such complex relationships are difficult to capture, yielding high error. Future work can explore more complex approximators, though we find that simple, low-overhead approximators are sufficient for most applications.

B. Relaxed Confidence Estimation

While LVP requires that the predicted value be identical to the value in memory (a confidence window of 0%), LVA allows for relaxed confidence. Figure 6 shows how different confidence windows affect MPKI and error for our baseline design. In these figures, both floating-point and integer data values employ confidence. The main tradeoff is between performance and output integrity. We can relax the confidence window to tolerate more error in approximated values. However, this does not mean that all approximated values have high error. Since the approximation degree is 0, values in the LHB are always updated. Relaxed confidence changes how often we provide approximate data – with infinitely relaxed confidence, the confidence counter is never decremented, and data is always approximated according to precise values in the LHB.

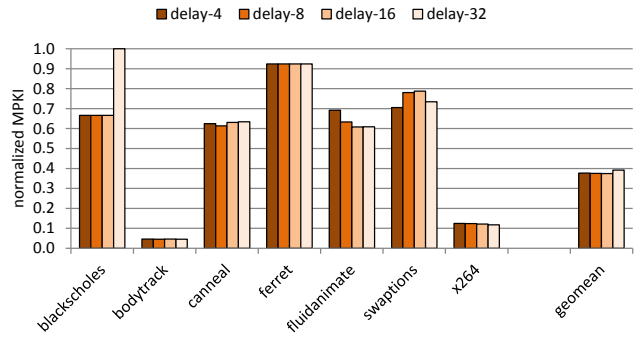
This study has some important takeaways. In x264, approximated pixel values occur during motion estimation, which determines if two frames are similar. x264 sees significant reductions in MPKI with almost no impact on error because a pixel has a finite range of values. By taking the average of the LHB values, the approximate value cannot be outside this range, minimizing error. For this reason, we find that integer values are generally more amenable to approximation and thus do not employ confidence for integer data in our baseline LVA configuration. Conversely, ferret sees increasing error with relaxed confidence estimation because we approximate vectors of floating-point data, with no discrete range or apparent pattern. Even with an average of the precise values of the LHB, errors occur in the approximation because different feature vectors are loaded by a single PC; with a GHB size of zero, this leads to increased error.

C. Value Delay

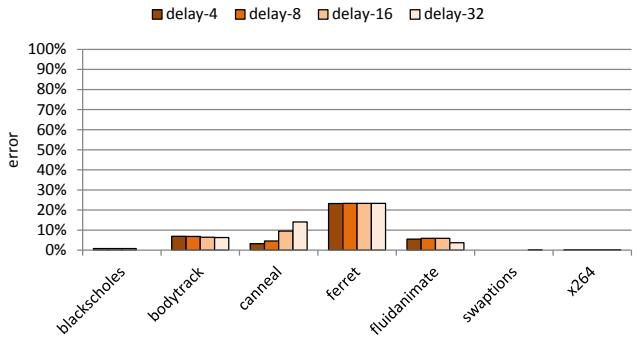
Variations in value delay show us the impact of approximating with stale data. We assume a value delay of 4, which means that, upon generating an approximation on an L1 data cache miss, the actual block from memory does not arrive at the cache until 4 load instructions later. Note that this is a conservative assumption; from our full-system simulations, we find that average value delay tends to only be $\sim$1.

LVA inherently tolerates inexactness; stale data due to value delay does not have a significant impact on either performance or error. Figure 7 shows the impact of different value delay assumptions on MPKI and error. MPKI is affected by value delay because the approximator is working with stale data, skewing its confidence calculations that determine whether or not data should be approximated. When the approximator data becomes too stale (as in blackscholes with value delay 32), the confidence mechanism may reject all approximations, yielding no MPKI reduction and no error. For swaptions and x264, output error is near zero even for value delays of 32 load instructions. In fact, for all benchmarks except canneal, value delay does not have a significant impact on output error. In canneal, <x, y> positions are constantly being swapped by the simulated annealing, which means that having stale data can impact the cost functions, which in turn determine whether two blocks are swapped. The data annotated in other benchmarks are less inter-dependent, hence we see negligible variation in output error.

LVA's resilience to value delay provides opportunities for throughput and energy optimizations, since LVA takes the fetching of the cache block off of the critical path of execution. As a result, approximated cache blocks can be deprioritized in the NoC [11, 26]. In addition, low energy

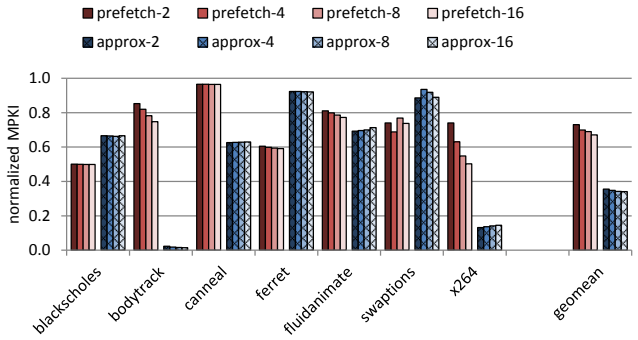


(a) MPKI.

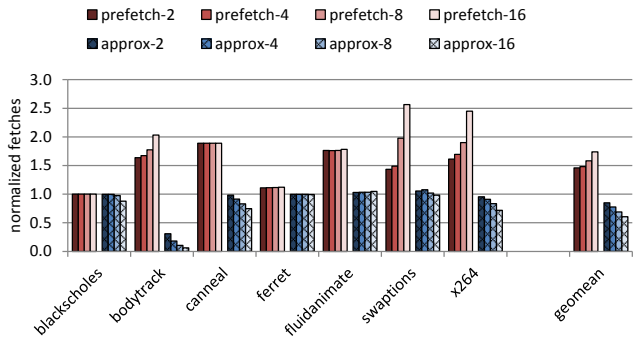


(b) Output Error.

Figure 7: Performance and error for varying value delays.



(a) MPKI.



(b) Number of Fetches.

Figure 8: MPKI and number of fetches for varying approximation and prefetch degrees.

techniques can be employed in the NoC [46] and memory hierarchy [9, 32] since the cache block does not need to be fetched with low latency.

D. Approximation Degree

A fundamental advantage of LVA is that it can selectively decide not to fetch a block into the L1 cache. Approximation degree determines how often it should fetch loads (i.e. train); load miss fetches in between training loads can be cancelled (i.e. never issued to the memory system). For example, an approximation degree of 4 means that every 5^{th} load to the same entry in the approximator is used to train that entry. The result is that the number of fetches and misses are decoupled. Conversely, with precise execution, the number of blocks fetched is synonymous with the number of misses.

To compare against LVA, we implement a GHB prefetcher that uses local delta correlation and next-line prefetching [29], with 2048 entries for the GHB and 2048 entries for the index table (which contain pointers into the GHB). We use 2048 entries to make the comparison fair, since our approximators contain 512 entries with 4 LHB values each. GHB prefetching works better than conventional prefetching tables because it naturally eliminates stale data by prioritizing the most recent accesses [29]. Prefetching degree determines how many prefetches (based on the prefetcher's pattern) to issue on a cache miss. In our results, prefetching is applied to all data, not just data that has been annotated as approximate. The number of fetches is a reflection of both useful and extraneous prefetches, while useful prefetches lead to an MPKI reduction.

Figure 8 shows how approximation degree and prefetch degree affect MPKI and the number of fetches. Prefetching reduces MPKI at the expense of an increase in fetches. Conversely, LVA reduces both MPKI and the number of fetches. The tradeoff for LVA is in output error (Figure 9); higher approximation degree leads to less frequent training of the approximator which increases error. On average, LVA (with approximation degree of 16) reduces the number of blocks fetched by over 39%, while prefetching (also degree of 16) increases the number of fetched blocks by 73%.

For canneal, the reduction in MPKI by prefetching is low and does not change despite different degrees. The random nature of canneal does not result in a discernible pattern that the prefetcher can exploit. Prefetching with a higher degree further reduces the MPKI for x264 and bodytrack. A higher prefetching degree brings more useful image data into the cache. However, using LVA we simply approximate this image data and, instead of fetching more data than necessary as a prefetcher would, significantly reduce the number of fetches compared to precise execution. Prefetching reduces MPKI at the expense of extra energy consumption due to more memory accesses; in today's power-constrained

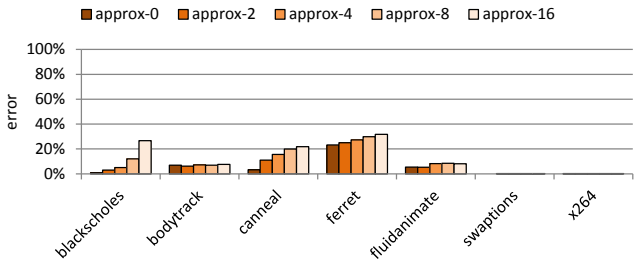


Figure 9: LVA output error with different approximation degrees.

computing environment, reducing MPKI with *fewer* memory accesses is a significant win.

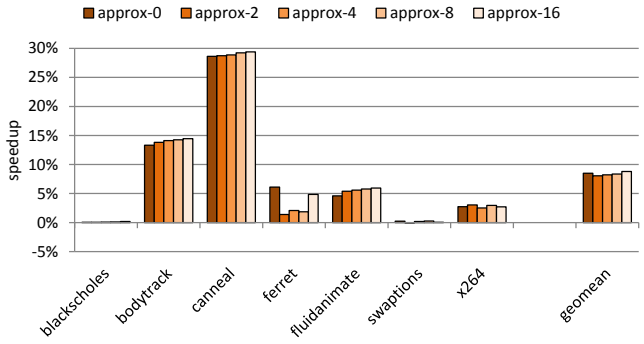
E. Full-System Performance and Energy

Though MPKI and blocks fetched provide estimates of performance and energy, we further evaluate LVA in full-system simulation. This provides deeper insight on the impact of LVA in the presence of out-of-order scheduling, contention and non-uniform memory access latencies. This also allows our approximators to operate with realistic value delays (∼1 on average), as opposed to the conservative value delay of 4 that we assumed in our design space exploration.

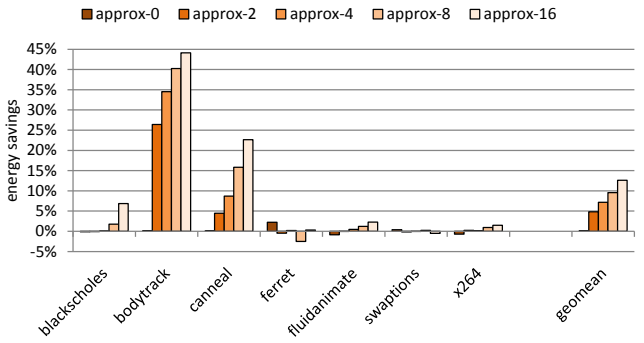
Figure 10a shows how LVA affects overall application performance, with varying approximation degrees. LVA improves performance by 8.5% on average (as much as 13.3% and 28.6% for bodytrack and canneal), with an approximation degree of 0. LVA reduces average L1 miss latency by 41.0%. Bodytrack and canneal see 72.5% and 48.3% reductions in miss latency. Bodytrack sees a larger improvement in L1 miss time, yet canneal sees a larger overall speedup; the cost computation in canneal's simulated annealing algorithm is so simple that the out-of-order processor is unable to fully mask the miss latency. For applications with high miss rates (e.g., bodytrack, canneal and fluidanimate), performance improves as approximation degree increases. Reducing the number of blocks fetched alleviates contention in the memory hierarchy. With approximation degree 16, interconnect traffic is reduced by 37.2% on average, lowering the average L1 miss latency by 47.2%.

Figure 10b shows that higher approximation degrees yield greater dynamic energy savings in the memory hierarchy. With approximation degree 16, the average number of blocks fetched from the L2 cache and main memory is reduced by 33.7% and 12.1%. This lowers the energy per L1 miss by 30.2%, resulting in 12.6% overall energy savings on average (up to 44.1% for bodytrack).

Summary. Unlike prefetchers which improve performance at a cost of more energy from fetches, load value approximation changes this tradeoff by exploring a third axis: output error. By allowing for some error in the application output, LVA achieves performance and energy improvements simultaneously. Figure 11 shows how LVA affects the energy-



(a) Speedup.



(b) Energy Savings.

Figure 10: Full-system performance and energy for varying approximation degrees.

delay product (EDP) of L1 cache misses, under varying approximation degrees. These results are normalized to precise execution. Some applications are computationally intensive (e.g., swaptions) and some have data values that are difficult to approximate (e.g., ferret), making them less amenable to LVA. On average though, LVA exploits the inexactness of applications to reduce the L1 miss EDP by 41.9%, 53.8% and 63.8%, with approximation degrees of 0, 4 and 16. These all yield speedups of ∼8.5% on average, with energy savings of 7.2% and 12.6% for approximation degrees 4 and 16. By tuning the approximator design parameters (such as relaxed confidence and approximation degree), load value approximation improves both performance and energy by allowing for some (tolerable) noise in the application output.

VII. Discussion

This section discusses load value approximation overheads, opportunities for further improvement and issues related to multiprocessors.

A. Hardware Overhead

Our baseline approximator design uses ∼18 KB or ∼10 KB of storage when considering 64- and 32-bit datatypes for the LHB values. Though we use 512-entry tables, hardware overhead can be reduced further. Figure 12 shows the number of static load instructions that access

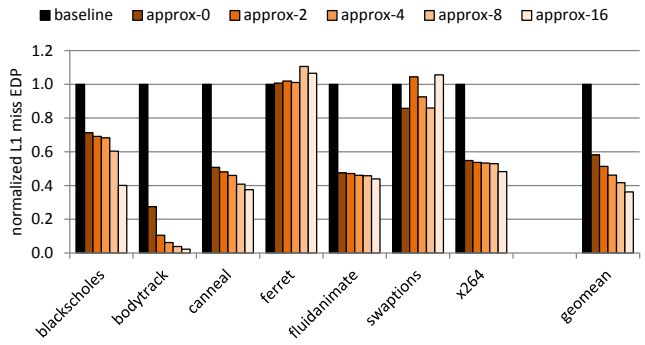


Figure 11: L1 miss EDP of LVA for different approximation degrees.

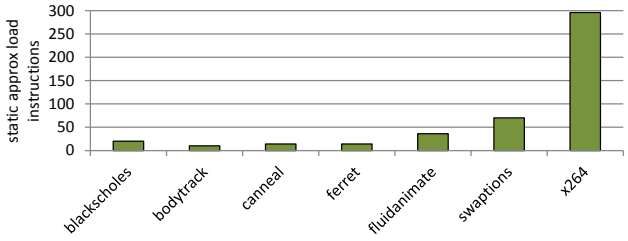


Figure 12: Number of static (distinct) PC values for approximate loads.

approximate data. Since we are not approximating all application data, the number of static load instructions to approximate data is small. This is the main reason why a GHB size of 0 works well in our evaluation (Section VI-A). The approximator table only needs to be large enough to store at most 300 entries (x264) and even lower for other applications. This demonstrates that load value approximation is feasible even on a small hardware budget.

B. Exploiting Floating-Point Precision

Because we are working with approximate data, the precision of traditional floating-point values is unnecessary. Though we used full precision floating-point values in our evaluations, it can be beneficial to reduce the number of bits of the floating-point mantissa used by the approximator. This improves our hash function with GHB sizes greater than zero for floating-point data. In Figure 13, we show the impact on fluidanimate's MPKI when we reduce the mantissa by up to 23 bits using a GHB size of 2 (we disable confidence to omit its effect on coverage). As we remove more bits, MPKI goes down. The error for fluidanimate remains low (around 10%) since fluidanimate can tolerate imprecision in its force and density calculations.

C. Memory Consistency

Maintaining memory consistency introduces significant complexity for traditional value prediction [24]. However,

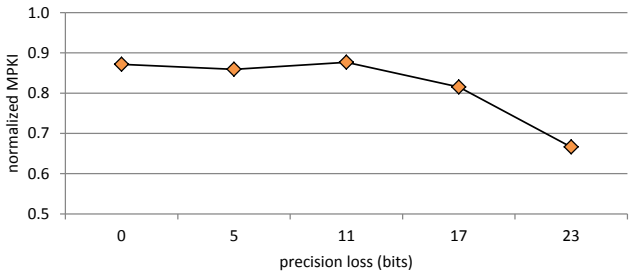


Figure 13: MPKI (normalized to precise execution) for varying floating-point (single) precision loss in fluidanimate.

this problem fundamentally does not apply in the context of load approximation. We take inspiration from prior work [33, 34, 40] that relaxes synchronization and memory ordering to improve performance with approximate computing. As our approximate values are consumed by the processor and true values are only used to train the prediction, the global order of these memory operations is not meaningful; we cannot maintain strict consistency across approximated values on multiple processors. Ordering of stores and all non-approximate memory operations still follows the consistency model of the underlying hardware. If consistency for a particular application is a critical concern, this application is unlikely to be a candidate for approximation.

VIII. Related Work

This work extends our preliminary study of load value approximation [37]. As discussed in Section II, prior research explores load value prediction [6, 7, 16, 20, 21]. We present novel extensions – elimination of rollbacks, relaxed confidence estimation and approximation degree – to trade-off output integrity for better performance and energy savings. In addition to load value prediction, we discuss some related research in approximate computing and data below.

Approximate Computing. Research in approximate computing spans all layers of the stack from circuits [8, 45] to architecture to programming languages. Architectural techniques include novel accelerator designs [14, 41]. Work in programming languages include programming frameworks to identify approximate data [5, 35], ISA extensions to leverage approximation [13] and compiler support [34]. We focus on a microarchitectural application of approximate computing; our technique has low overhead and does not require radical or complex changes to the processor pipeline leading to a low barrier to entry.

Approximate Value Locality. Floating-point operations can be made more energy-efficient by using fewer mantissa bits, resulting in modest imprecision [44]. This improves the value locality, since the loss in precision yields more identical values [1]. Physics-based animations are inherently error tolerant [48]; dynamically adapting floating point precision can accelerate physics simulation [47]. Sreeram and

Pande have explored approximate value locality [40] in the context of approximate store instructions to reduce conflicts in software transactional memory. This is similar to relaxed synchronization, which selectively allows races to occur, thus trading off output error for faster execution [33]. Our work instead focuses on the approximate value locality of load instructions. Recent work also proposes eliminating rollbacks for value prediction [43] but does not explore relaxed confidence mechanisms nor energy savings.

Approximate Data Storage. There has been significant research on storing approximate data more efficiently. Drowsy caches [15] reduce the supply voltage of SRAM cells to save power at the cost of potential bit failures. Approximate data in DRAM can be refreshed at lower rates, saving energy while increasing the likelihood of data corruption [22]. PCM performance and lifetime can be improved by reducing write precision and reusing failed cells for storing approximate data [36]. These techniques yield more efficient storage, while our work reduces both the latency and energy of fetching data from memory.

IX. Conclusion

We present load value approximation and evaluate its feasibility across a diverse set of applications. By generating approximate values on load misses, we can avoid the high latency and energy of fetching data from memory. From our design space exploration and full-system simulations, we find that load value approximation achieves up to 28.6% speedup (8.5% on average) and 44.1% energy savings (12.6% on average) in the memory hierarchy, while maintaining low output error. Load value approximation takes its inspiration from value prediction; yet there are fundamental differences provided by the approximate computing paradigm that lead to novel microarchitectural choices. We explore relaxed confidence estimation and selective fetching via approximation degree to exploit the performance-error and energy-error tradeoffs. Load value approximation opens up new possibilities for achieving both high performance and energy-efficiency in future processors.

Acknowledgment

The authors thank the anonymous reviewers for their thorough suggestions on improving this work. The authors also thank the members of the Enright Jerger research group for their feedback. This work is supported by a Bell Graduate Scholarship, the Natural Sciences and Engineering Research Council of Canada, the Canadian Foundation for Innovation, the Ministry of Research and Innovation Early Researcher Award and the University of Toronto.

References

[1] C. Alvarez, J. Corbal, and M. Valero, "Fuzzy memoization for floating-point multimedia applications," *IEEE Transactions on Computers*, 2005.

[2] W. Baek and T. M. Chilimbi, "Green: a framework for supporting energy-conscious programming using controlled approximation," in *Proc. Conf. Programming Language Design and Implementation*, 2010.

[3] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[4] F. Black, "Fact and fantasy in the use of options," *Financial Analysts Journal*, pp. 36–72, 1975.

[5] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain<T>: A first-order type for uncertain data," in *Proc. of the Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2014.

[6] M. Burtscher, "Improving context-based load value prediction," Ph.D. dissertation, University of Colorado, 2000.

[7] L. Ceze, K. Strauss, J. Tuck, J. Torrellas, and J. Renau, "CAVA: using checkpoint-assisted value prediction to hide L2 misses," *ACM Transactions on Architecture and Code Optimization*, 2006.

[8] A. Chandrakasan and R. Brodersen, "Minimizing power consumption in digital CMOS circuits," *Proc. of the IEEE*, vol. 83, no. 4, pp. 498–523, 1995.

[9] N. Chatterjee, M. Shevgoor, R. Balasubramonian, A. Davis, Z. Feng, R. Illikkal, and R. Iyer, "Leveraging heterogeneity in DRAM main memories to accelerate critical word access," in *Proc. of the Int. Symp. on Microarchitecture*, 2012.

[10] V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Analysis and characterization of inherent application resilience for approximate computing," in *Proc. Int. Design Automation Conference*, 2013.

[11] R. Das, O. Mutlu, T. Moscibroda, and C. R. Das, "Aergia: exploting packet latency slack in on-chip networks," in *Proc. Int. Symp. Computer Architecture*, 2010.

[12] J. Deutscher, A. Blake, and I. Reid, "Articulated body motion capture by annealed particle filtering," in *IEEE Conference on Computer Vision and Pattern Recognition*, vol. 2, 2000, pp. 126–133.

[13] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Architecture support for disciplined approximate programming," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2012.

[14] ——, "Neural acceleration for general-purpose approximate programs," in *Proc. Int. Symp. Microarchitecture*, 2012.

[15] K. Flautner, N. S. Kim, S. Martin, D. Blaauw, and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power," in *Proc. Int. Symp. Computer Architecture*, 2002.

[16] F. Gabbay, "Speculative execution based on value prediction," Technion - Israel Institute of Technology, EE Department Technical Report 1080, 1996.

[17] P. Glasserman and N. Merener, "Cap and swaption approximations in libor market models with jumps," *Journal of Computational Finance*, vol. 7, no. 1, pp. 1–36, 2003.

[18] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A detailed and flexible cycle-accurate network-on-chip simulator," in *Proc. Int. Symp. Performance Analysis of Systems and Software*, 2013.

[19] S. Khan, A. Alameldeen, C. Wilkerson, O. Mutlu, and D. Jimenez, "Improving cache performance by exploiting read-write disparity," in *Proc. Int. Symp. High-Performance Computer Architecture*, 2014.

[20] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 1996.

[21] S. Liu and J. Gaudiot, "Potential impact of value prediction on communication in many-core architectures," *IEEE Transactions on Computers*, 2009.

[22] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn, "Flikker: saving DRAM refresh-power through critical data partitioning," in *Int. Conf. Architectural Support for Programming Languages and Operating Systems*, 2011.

[23] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proc. Conf. Programming Language Design and Implementation*, 2005.

[24] M. M. K. Martin, D. J. Sorin, H. W. Cain, M. D. Hill, and M. H. Lipasti, "Correctly implementing value prediction in microprocessors that support multithreading or multiprocessing," in *Proc. Int. Symp. Microarchitecture*, 2001.

[25] J. Meng, S. Chakradhar, and A. Raghunathan, "Best-effort parallel execution framework for recognition and mining applications," in *Proc. Int. Parallel and Distributed Processing Symposium*, 2009.

[26] A. K. Mishra, O. Mutlu, and C. R. Das, "A heterogeneous multiple network-on-chip design: an application-aware approach," in *Proc. Int. Design Automation Conference*, 2013.

[27] T. Nakra, R. Gupta, and M. L. Soffa, "Global context-based value prediction," in *Proc. Int. Symp. High-Performance Computer Architecture*, 1999.

[28] N. Neelakantam, C. Blundell, J. Devietti, M. M. K. Martin, and C. Zilles, "FeS2: a full-system execution-driven simulator for x86," poster presented at Int. Conf. Architectural Support for Programming Languages and Operating Systems, 2008.

[29] K. J. Nesbit and J. E. Smith, "Data cache prefetching using a global history buffer," *Micro, IEEE*, vol. 25, no. 1, pp. 90–97, 2005.

[30] A. Perais and A. Seznec, "EOLE: Paving the way for an effective implementation of value prediction," in *Proc. of the Int. Symp. on Computer Architecture*, 2014.

[31] ——, "Practical data value speculation for future high-end processors," in *Proc. of the Int. Symp. on High Performance Computer Architecture*, 2014.

[32] S. Phadke and S. Narayanasamy, "MLP aware heterogeneous memory system," in *Int. Conf. on Design, Automation and Test in Europe*, 2011.

[33] L. Renganarayana, V. Srinivasan, R. Nair, and D. Prener, "Programming with relaxed synchronization," in *Proc. Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*, 2012.

[34] M. Samadi, J. Lee, D. Jamshidi, A. Hormati, and S. Mahlke, "SAGE: Self-tuning approximation for graphics engines," in *Proc. of the Int. Symp. on Microarchitecture*, 2013.

[35] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman, "EnerJ: approximate data types for safe and general low-power consumption," in *Proc. Conf. Programming Language Design and Implementation*, 2011.

[36] A. Sampson, J. Nelson, K. Strauss, and L. Ceze, "Approximate storage in solid-state memories," in *Proc. Int. Symp. Microarchitecture*, 2013.

[37] J. San Miguel and N. Enright Jerger, "Load value approximation: Approaching the ideal memory access latency," in *Workshop on Approximate Computing Across the System Stack*, 2014.

[38] Y. Sazeides and J. Smith, "The predictability of data values," in *Proc. Int. Symp. Microarchitecture*, 1997.

[39] S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard, "Managing performance vs. accuracy trade-offs with loop perforation," in *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of software engineering*, 2011, pp. 124–134.

[40] J. Sreeram and S. Pande, "Exploiting approximate value locality for data synchronization on multi-core processors," in *Proc. Int. Symp. Workload Characterization*, 2010.

[41] R. St. 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 *Proc. of the Int. Symp. on Computer Architecture*, 2014.

[42] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," Technical Report HPL-2008-20, HP Labs, 2008.

[43] B. Thwaites, G. Pekhimenko, H. Esmaeilzadeh, A. Yazdanbakhsh, O. Mutlu, J. Park, G. Mururu, and T. Mowry, "Rollback-free value prediction with approximate loads," poster presented at Int. Conf. Parallel Architectures and Compilation Techniques, 2014.

[44] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar, "Reducing power by optimizing the necessary precision/range of floating-point arithmetic," *IEEE Transactions on VLSI Systems*, 2000.

[45] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan, "SALSA: Systematic logic synthesis of approximate circuits," in *Design Automation Conference*, 2012.

[46] J. Won, X. Chen, P. V. Gratz, J. Hu, and V. Soteriou, "Up by their bootstraps: Online learning in artificial neural networks for CMP uncore power management," in *Proc. Int. Symp. on High Performance Computer Architecture*, 2014.

[47] T. Yeh, P. Faloutsos, S. Patel, M. Ercegovac, and G. Reinman, "The art of deception: Adaptive precision reduction for area efficient physics acceleration," in *Int. Symp. on Microarchitecture*, Dec 2007.

[48] T. Yeh, G. Reinman, S. J. Patel, and P. Faloutsos, "Fool me twice: Exploring and exploiting error tolerance in physics-based animation," *ACM Transactions on Graphics*, December 2009.

[49] H. Zhou, J. Flanagan, and T. M. Conte, "Detecting global stride locality in value streams," in *Proc. Int. Symp. Computer Architecture*, 2003.