Load Value Approximation: Approaching the Ideal Memory Access Latency

Joshua San Miguel and Natalie Enright Jerger
Department of Electrical and Computer Engineering
University of Toronto
joshua.sanmiguel@mail.utoronto.ca, enright@eecg.toronto.edu

Abstract

Approximate computing recognizes that many applications can tolerate inexactness. These applications, which range from multimedia processing to machine learning, operate on inherently noisy and imprecise data. As a result, we can trade-off some loss in output value integrity for improved processor performance and energy-efficiency. In this paper, we introduce load value approximation*. In modern processors, upon a load miss in the private cache, the data must be retrieved from main memory or from the higher-level caches. These data accesses are costly both in terms of latency and energy. We implement load value approximators, which are hardware structures that learn value patterns and generate approximations of the data. The processor can then use these approximate data values to continue executing without incurring the high cost of accessing memory. We show that load value approximators can achieve high coverage while maintaining very low error in the application's output. By exploiting the approximate nature of applications, we can draw closer to the ideal memory access latency.*

1. Introduction

Approximate computing is an emerging paradigm for energy-efficient processor design. It recognizes that a wide range of commercial, multimedia and scientific applications are inherently *approximate*. That is to say, they operate on noisy data and perform inexact computations. These applications—which range from audio and video processing to recognition and mining applications—can tolerate some error in their output values. This allows processor designers to trade-off data value integrity for better performance and low power. Recent work has proposed ideas for accelerating approximate computations [1, 10], relaxing synchronization [22, 26] and efficiently storing approximate data [15, 24].

Though these innovations improve energy-efficiency, the memory wall persists in modern multiprocessors. Significant attention has been paid to caches and networks-on-chip (NoCs), which lie on the path to memory, contributing to the latency of accessing application data. Unfortunately, high-performance caches and NoCs tend to be power-hungry, together consuming as much as 33% of the chip power budget [25]. Cache power [18] and NoC power [3] are expected to grow infeasibly high if current designs are naively scaled with increasing core counts. Our work aims to exploit the approximate nature of applications to minimize the latency and energy of accessing data in the memory hierarchy.

In this paper, we introduce *load value approximation*. Since many applications can tolerate inexactness, their memory data can be approximated. In traditional processors, upon a load miss in the private 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 NoC results in many cycles between the request and receipt of data by the processor. We propose the use of 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 to receive the data.

Load value approximation follows from previous work on load value prediction [4, 6, 12, 13, 14]. Applications have been found to exhibit value locality; they tend to reuse common values in memory. This is typically due to runtime constants and redundancy in real-world input data sets [13]. Unfortunately, load value prediction introduces significant complexity in supporting speculative values and performing rollbacks upon mispredictions. Martin et al. note that special care must be taken when implementing value prediction on multiprocessors to avoid breaking the memory consistency model [16]. Furthermore, traditional load value predictors tend to exhibit low coverage and accuracy with floating-point data [12]. Since floating-point values are represented with fine precision, even small variations in their values are treated as mispredictions. Our work aims to overcome these challenges. By approximating values instead of strictly predicting them, we can not only achieve greater coverage and accuracy but also eliminate the need for speculation and rollbacks. Load value approximation effectively enables us to approach the ideal latency of accessing memory.

Our work makes the following contributions:

- We propose load value approximation and investigate its feasibility on a diverse set of applications.
- We show that we can approximate the values of 88.80% of load instructions while maintaining low output error of 2.87% on average (no more than 7.02%).

2. Background

2.1. Approximate Computing

Approximate computing refers to applications whose outputs are not restricted to a single "correct" value. For example, recognition and mining applications usually produce a range of acceptable solutions instead of a unique solution [17]. Similarly, the outputs of audio, video and image processing can tolerate small variations that are not perceptible to the user [7].

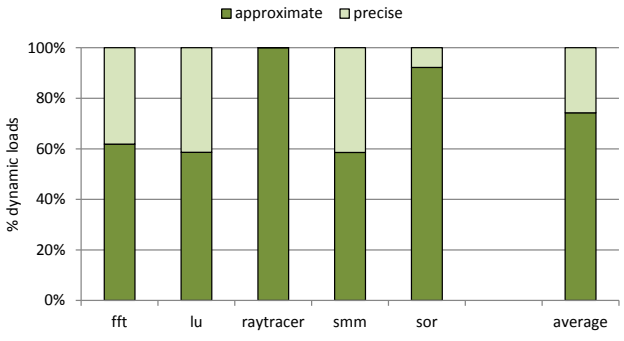


Figure 1: Approximate and precise dynamic loads.

To exploit these types of applications, researchers have provided programming and ISA support for approximate computations [2, 9]. In our evaluations, we use EnerJ, a framework that allows programmers to declare data as either precise or approximate [23]. In applications provided by EnerJ, Figure 1 shows that on average, 74.3% of all dynamic loads access approximate data[1]. This demonstrates significant opportunity for employing load value approximation.

Since approximate data can tolerate loss in value integrity, they can be stored more efficiently. As demonstrated by drowsy caches [11], the supply voltage of SRAM cells can be reduced for low 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 [15]. Sampson et al. improve PCM performance and lifetime by reducing write precision and reusing failed cells for storing approximate data [24]. Though these techniques allow for more efficient storage, our work instead aims to minimize both the latency and energy of fetching data from memory.

Floating-point operations can be made more energy-efficient by using fewer mantissa bits, resulting in modest imprecision [27]. This also improves the value locality of floating-point data, since the loss in precision yields more identical values [1]. As we show later in Section 3, we exploit this imprecision to maximize the coverage and accuracy of load value approximation. Sreeram and Pande have explored approximate value locality [26]. However, their work targets approximate store instructions instead of loads in order 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 [22]. Our work instead focuses on the approximate value locality of load instructions.

2.2. Load Value Prediction

Value locality is the notion that data values are likely to be similar to previously seen values [13]. This has led to extensive research on designing load value predictors [4, 6, 12, 14]. In these schemes, upon a load miss in the L1 cache, a request is sent to fetch the data from the next level of memory. 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 roll back the speculatively executed instructions.

Implementing load value prediction is challenging; it introduces high complexity for managing speculative values and adds the risk of costly rollbacks when prediction accuracy is low. Due to the long latencies of cache misses, processors must be equipped with large buffers to store all speculative values since they need to be validated later. Upon a misprediction, the processor must also be able to quickly restore its registers and undo all speculative modifications to memory, either in the store queue or in the L1 cache. Furthermore, in multiprocessors, it is possible for another thread to modify a value that has been speculated, resulting in complications with the memory consistency model [16].

It is important for load value predictors to achieve both high coverage (*predicted loads / total loads*) and high accuracy (*correctly predicted loads / predicted loads*) to maximize the performance gain. Unfortunately, coverage and accuracy tend to be low when predicting floating-point values [12]. Since predicted values need to be exactly identical to the actual values, even small variations in floating-point precision result in costly rollbacks. These challenges of implementing load value prediction can be mitigated by taking advantage of the approximate nature of applications.

3. Load Value Approximation

Load value approximation estimates data values to save the processor from needing to retrieve the actual values from memory. These approximate values must be accurate to maintain low error in the application's output. Figure 2 shows an overview of load value approximation. When a load *X* misses in the private cache (1), the load value approximator generates *X_approx* (2). The processor assumes that this is the actual value of *X* and proceeds with its execution (3a). A request is still sent from the private cache to the next level of the memory hierarchy to retrieve *X_actual* (3b). This happens 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 (4).

The load value approximator effectively acts as an intermediary between the private cache and the next level of the memory hierarchy, accessing data on the processor's behalf. It is not a storage element but rather a learning mechanism that learns the values in memory. Unlike a cache, the approximator is tagged and indexed by either the instruction address, recently seen data values, or both. This allows load value approximation to scale well with the large data sets of current applications.

Unlike value prediction, load value approximation does not require speculative execution. Since approximate computing

[1] Simulation details can be found in Section 4.

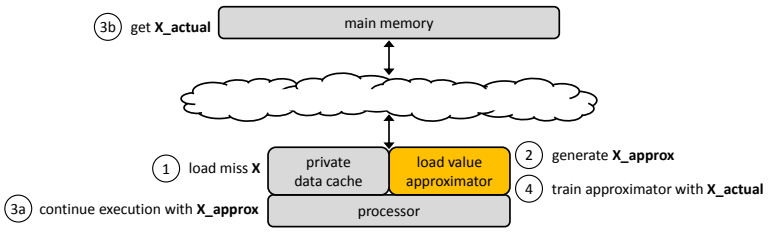


Figure 2: Load value approximation overview.

applications can tolerate output error, the values generated by the approximator do not need to be validated against the actual values in memory. This eliminates rollbacks and takes the memory access off the critical path, allowing the processor to execute with near-ideal memory latency. This also offers opportunity for energy savings. Low-power techniques—such as heterogeneous NoCs [19] and memory modules [21]—can be employed when accessing approximate data in memory since they are off the critical path. Furthermore, as shown in Figure 2 (4), the main purpose of fetching approximate data is to train the approximator. By employing confidence estimators, we can selectively decide not to fetch the data at all if the approximator is deemed sufficiently accurate, thus saving energy. These ideas will be explored in future work.

3.1. Context-Based Approximator

In this paper, we focus on how to design an accurate load value approximator that can maintain low output error. In traditional value predictors, the values of the most recent loads provide the *context* for deriving the prediction. The context can serve either as an index into a prediction table or as a value buffer from which we select a prediction [4]. In our approximator design, global context (values from all load instructions) serves as the index while local context (values per load instruction) serves as the value buffer. Figure 3 shows the general structure of our load value approximator. A global history buffer (GHB) stores the most recent load values in the processor. This provides the global context, which has been shown to improve accuracy since it incorporates global control path information [20]. Our approximator uses a direct-mapped table, indexed and tagged by a hash h of the instruction address and the values in the GHB. Each table entry contains a local history buffer (LHB), which stores the most recent load values for the entry's tag. This provides the local context and is used in some function f to generate an approximate value. The processor then uses this value to continue executing.

Global Context. The global context and the instruction address are hashed together to index into the table. Any hash function h can be used; for example, h can be the exclusive-or of the instruction address with each value in the GHB. The ideal size of the GHB varies per application; we evaluate different GHB sizes in Section 5.1. It can be challenging to train the approximator with floating-point context. Due to their fine precision, floating-point values that are approximately

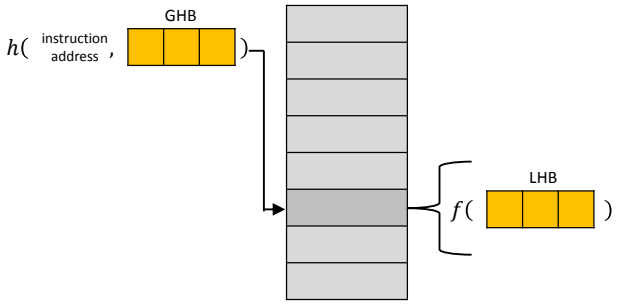


Figure 3: Context-based load value approximator.

similar but not exactly identical end up indexing into different table entries. To address this, we can reduce the number of mantissa bits in the GHB to improve floating-point value locality [1]. Though this introduces imprecision in the global context, it can map approximately similar values to the same table entries.

Local Context. To generate an approximate value, each entry stores the tag of hash h and the local context, which tracks the most recently seen values for that tag. The ideal LHB size may vary; we evaluate different sizes in Section 5.2. In a traditional load value predictor, a selection mechanism simply picks one of the values in the LHB as the prediction [5]. Load value approximation extends this to use any function f to generate a more educated estimate. For example, f can compute the average of the LHB values. Unlike value prediction, value approximation does not trigger a rollback if the generated value does not exactly match the actual value in memory; the generated value only needs to be approximately close enough that the error in the application's final output is low.

Value Delay. As in value prediction, an important challenge in implementing context-based approximators is value delay [29]. Value delay occurs when the actual values of preceding loads have not yet been retrieved and inserted into the history buffers. As a result, the approximator must generate an estimate based on older (potentially stale) values in the GHB and LHB, limiting its accuracy. This is a significant problem for traditional load value predictors. Say we have a load miss on A; a value prediction is made, and the processor saves its state and enters speculative execution. Now if we encounter a load miss on B, the value prediction for B may be less accurate since the predictor has not yet been trained with the actual value of A. Unfortunately, value predictors typically only save

Word size	8 bytes
Table size	2048 entries
Value delay	4 load instructions
Hash h	XOR (inst addr, GHB[0], GHB[1], ...)
Function f	AVERAGE (LHB[0], LHB[1], ...)

Table 1: Default parameters for load value approximator.

the processor state on the first load miss (A) to keep complexity low [6]. Thus if B is mispredicted—which is likely to happen due to value delay—the processor must rollback to A, even if A was predicted correctly. Value delay is more tolerable for load value approximation since rollbacks are eliminated. However, value delay still affects the accuracy of the approximator; we evaluate this in Section 5.3.

4. Methodology

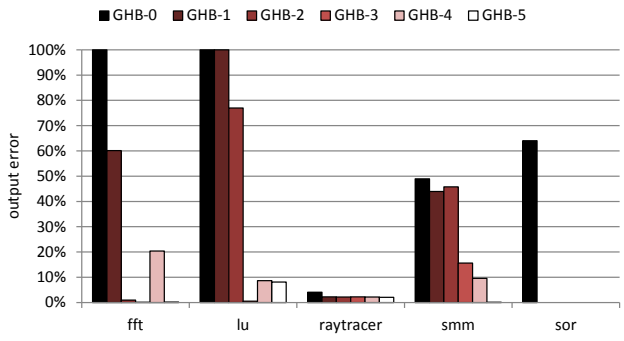
We implement load value approximation using the EnerJ framework [23]. We evaluate benchmarks provided with EnerJ—fft, lu, raytracer, smm and sor—which are representative of common scientific and multimedia workloads in approximate computing. We use the default inputs with two exceptions: we input a musical audio sample for fft and a DNA electrophoresis sparse matrix [8] for smm. We evaluate the quality of each benchmark's final output using the error metrics defined by Sampson et al. [23], which measure the differences of the final output values when running with and without load value approximation. For our EnerJ applications, output error below 10% is generally acceptable [24]. Note that all of our error results pertain to the application's final output, not the individual error of each load instruction.

The applications in the EnerJ framework have already been modified with programmer annotations to distinguish approximate data from precise data. We modify the EnerJ runtime: upon a load access to approximate data, the runtime returns a value generated by our approximator instead of the actual value in memory. In our evaluations, we invoke the approximator only on load accesses to arrays stored in DRAM; we ignore data structures that are small enough to fit in the private L1 cache. These loads make up 74.3% of all dynamic loads (both precise and approximate), as shown previously in Figure 1.

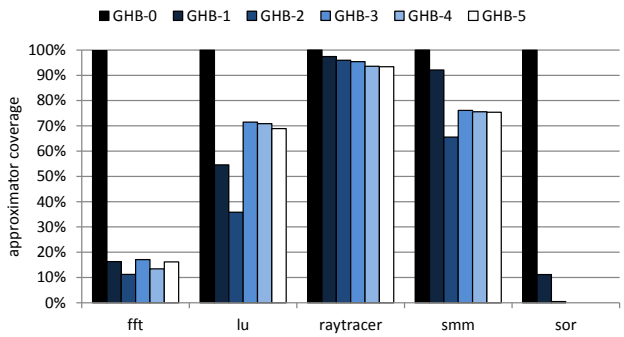
Table 1 lists our default parameters for implementing the load value approximator. Though we use exclusive-or and average for h and f respectively, other functions may be used for approximation. To simulate a value delay of 4, we impose the restriction that each invocation of the approximator is unaware of the values of the last 4 loads. We explore different value delays in Section 5.3. Our load value approximator uses a 2048-entry table, a typical size used in value predictor implementations [4, 6].

5. Evaluation

We evaluate the feasibility of implementing load value approximation. We explore the approximator design space and show



(a) Output error.



(b) Approximator coverage.

Figure 4: Varying size of GHB (LHB-1).

that it is possible to achieve high coverage while maintaining low output error. Since the approximator table is tagged, coverage is defined as the fraction of approximate loads that hit in the table. This ensures that destructive aliasing only affects coverage and not output error. Our results suggest that there is great potential in employing load value approximation to minimize the latency and energy of accessing memory.

5.1. Global Context

Figures 4a and 4b show the output error and approximator coverage while varying the GHB size. The LHB size is kept constant at 1, which implements a last value approximator. With a GHB size of 0, the approximator table is indexed only by the instruction address. This scenario yields the highest coverage since the number of static load instructions is typically small enough to fit inside the 2048-entry table. For benchmarks fft and sor, a GHB size greater than 0 yields too much destructive aliasing and thus limits coverage. This is due to poor floating-point value locality, as discussed in Section 3.1. For the other benchmarks, the coverage remains high, and the output error generally declines as the GHB size increases. This is expected since a larger GHB provides more global context information from which the approximator can make a more accurate estimate. However, the GHB must not be too large since older values become stale and can actually degrade the accuracy, as can be seen with lu for GHB sizes greater than 3.

benchmark	GHB size	LHB size	approximator size	approximator coverage	output error	predictor accuracy
fft	0	2	49.152 kB	99.77%	7.02%	0.00%
lu	3	1	32.792 kB	71.48%	0.55%	99.90%
raytracer	1	1	32.776 kB	97.44%	2.26%	92.26%
smm	5	1	32.808 kB	75.40%	0.01%	99.70%
sor	0	2	49.152 kB	99.92%	4.50%	0.02%

Table 2: Best approximator configurations for each benchmark.

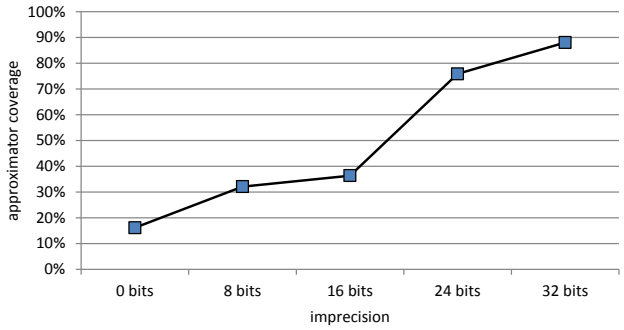


Figure 5: Approximator coverage in fft (GHB-5, LHB-1), varying number of bits removed from mantissa.

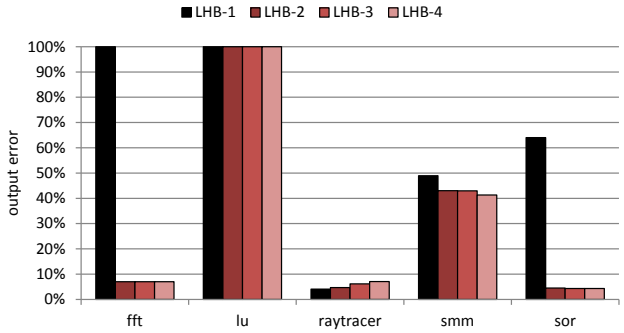


Figure 6: Output error, varying size of LHB (GHB-0).

As mentioned earlier, approximately similar floating-point values end up mapping to different entries in the table. With an infinite-size table, this is not a problem. However, with only 2048 entries, this can result in low coverage due to destructive aliasing. Figure 5 shows how coverage in fft can be improved by reducing the number of mantissa bits in the GHB (assuming double-precision floating-point values). Though not shown, the output error stays below 10% for all imprecision levels in the figure. The GHB and LHB sizes are fixed at 5 and 1 respectively. With full precision (0 bits removed), the approximator covers only 16.15% of all approximate loads. By removing 32 bits from the mantissa, we can dramatically increase the coverage to 88.02% while maintaining low output error.

5.2. Local Context

Figure 6 shows how the output error varies with LHB size. The GHB size is kept constant at 0; the approximator table is indexed only by the instruction address, and thus coverage (not shown) is nearly 100% in all cases. Increasing the LHB size tends to improve accuracy since more local context is available for making approximations. For lu and smm though, the error remains high regardless of the LHB size. This is because the GHB size is held at 0; these benchmarks benefit from more global context, as shown earlier in Figure 4. Note that too much local context can lead to stale data. This is evident in raytracer where a last value approximator (LHB-1) is sufficient for low error. Some benchmarks—such as fft and sor—benefit immensely from looking beyond just the last value. This is because the function f allows us to make better estimates (in this case, taking the average) using recently seen values. This is not possible with traditional value prediction since computing an estimate is far less likely to exactly match the actual value in memory.

Table 2 lists the best configurations (from our experiments) for each of the benchmarks. The table shows the corresponding output error and approximator coverage. Notice that there is no single ideal configuration for all benchmarks. This is expected since each application exhibits its own data value patterns. To address this, load value approximators can employ dynamic or hybrid schemes to adapt to the application [28]. We leave this to future work.

Table 2 shows that it is possible to implement load value approximators that achieve high coverage (88.80% on average) while keeping error low (2.87% on average). The table also lists the prediction accuracy, if we were to use our approximator as a traditional value predictor instead. To be considered accurate for a traditional value predictor, the predicted value must exactly match the actual value in memory. Despite imperfect prediction accuracies—in fact, nearly 0% for fft and sor—the output error remains very low. This shows how load value approximation is able to exploit the approximate nature of applications. Though traditional load value prediction achieves high accuracy with lu, raytracer and smm, load value approximation is a more attractive solution since it avoids the high cost and complexity of supporting speculative values and rollbacks.

5.3. Value Delay

So far, our results have assumed a value delay of 4 load instructions; the values of the 4 most recent loads are not visible to the approximator. Figure 7 shows how larger value delays impacts output error, with the best approximator configurations. With a value delay of 12, the average output error increases from 2.87% to 4.84%. This is expected since older values in

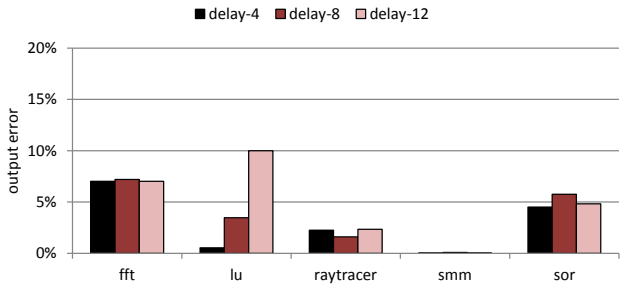


Figure 7: Output error (best approximator configurations), varying value delay.

the GHB and LHB are likely to become stale, thus making the approximator less accurate. Despite this, we see that it is still possible to maintain low output error (within 10%) even in the presence of very large value delay.

6. Future Work

In this paper, we focus primarily on the design of load value approximators and assess the feasibility of load value approximation on different applications. In future work, we first plan to implement dynamic and hybrid schemes to expand our design space [28]. We will then focus on evaluating load value approximation using full-system simulations. This will allow us to quantify the performance speedup and assess how closely we approach the ideal memory access latency. Furthermore, as discussed in Section 3, we will implement low-power techniques—such as heterogeneous NoCs [19] and memory modules [21]—for accessing approximate data. We can then quantify the energy savings of load value approximation. Approximate computing introduces endless research opportunities for exploiting inexactness and rethinking how approximate data is accessed from memory.

7. Conclusion

We present load value approximation and evaluate its feasibility across a diverse set of applications. By generating approximate values, we can avoid the high latency and energy of fetching data from memory. We explore the design of load value approximators and show that we can achieve high coverage of dynamic loads (88.80% on average) while keeping the output error very low (2.87% on average). Load value approximation opens up new possibilities for designing high-performance, energy-efficient processors to accomodate the immense growth of data sets in emerging applications.

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] S. Y. Borkar. Future of interconnect fabric: a contrarian view. In *Proc. Int. Workshop on System Level Interconnect Prediction*, 2010.

[4] M. Burtscher. *Improving context-based load value prediction*. PhD thesis, University of Colorado, 2000.

[5] M. Burtscher and B. G. Zorn. Exploring last n value prediction. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, 1999.

[6] 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.

[7] 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.

[8] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software*, 2011.

[9] 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.

[10] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *Proc. Int. Symp. Microarchitecture*, 2012.

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] 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.

[18] Y. Meng, T. Sherwood, and R. Kastner. On the limits of leakage power reduction in caches. In *Proc. Int. Symp. High-Performance Computer Architecture*, 2005.

[19] 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.

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

[21] S. Phadke and S. Narayanasamy. MLP aware heterogeneous memory system. In *Proc. Conf. Design, Automation and Test in Europe*, 2013.

[22] 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.

[23] 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.

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

[25] A. Sharifi, A. K. Mishra, S. Srikantaiah, M. Kandemir, and C. R. Das. PEPON: performance-aware hierarchical power budgeting for NoC based multicores. In *Proc. Int. Conf. Parallel Architectures and Compilation Techniques*, 2012.

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

[27] 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.

[28] K. Wang and M. Franklin. Highly accurate data value prediction using hybrid predictors. In *Proc. Int. Symp. Microarchitecture*, 1997.

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