Architecture-Aware Approximate Computing

MUSTAFA KARAKOY, TOBB University of Economics and Technology, TURKEY
ORHAN KISLAL, Pennsylvania State University, USA
XULONG TANG, Pennsylvania State University, USA
MAHMUT TAYLAN KANDEMIR, Pennsylvania State University, USA
MEENAKSHI ARUNACHALAM, Intel, USA

Deliberate use of approximate computing has been an active research area recently. Observing that many application programs from different domains can live with less-than-perfect accuracy, existing techniques try to trade off program output accuracy with performance-energy savings. While these works provide point solutions, they leave three critical questions regarding approximate computing unanswered, especially in the context of dropping/skipping costly data accesses: (i) what is the maximum potential of skipping (i.e., not performing) data accesses under a given inaccuracy bound?; (ii) can we identify the data accesses to drop randomly, or is being architecture aware (i.e., identifying the costliest data accesses in a given architecture) critical?; and (iii) do two executions that skip the same number of data accesses always result in the same output quality (error)? This paper first provides answers to these questions using ten multithreaded workloads, and then, motivated by the negative answer to the third question, presents a program slicing-based approach that identifies the set of data accesses to drop such that (i) the resulting performance/energy benefits are maximized and (ii) the execution remains within the error (inaccuracy) bound specified by the user. Our slicing-based approach first uses backward slicing and then forward slicing to decide the set of data accesses to drop. Our experimental evaluations using ten multithreaded workloads show that, when averaged over all benchmark programs we have, 8.8% performance improvement and 13.7% energy saving are possible when we set the error bound to 2%, and the corresponding improvements jump to 15% and 25%, respectively, when the error bound is raised to 4%.

CCS Concepts: • **Computer systems organization → Multicore architectures**;

Additional Key Words and Phrases: Approximate computing, compiler, manycore system

ACM Reference Format:
Mustafa Karakoy, Orhan Kislal, Xulong Tang, Mahmut Taylan Kandemir, and Meenakshi Arunachalam. 2019. Architecture-Aware Approximate Computing. *Proc. ACM Meas. Anal. Comput. Syst.* 3, 2, Article 38 (June 2019), 24 pages. https://doi.org/10.1145/3326153

1 INTRODUCTION

Constraints imposed by hardware scaling and data and control dependencies across computations combined by compilers' inability to maximize computation parallelism limit the performance potential of multithreaded workloads running on emerging manycore systems. State-of-the-art

Authors' addresses: Mustafa Karakoy, TOBB University of Economics and Technology, Ankara, TURKEY, m.karakoy@yahoo.co.uk; Orhan Kislal, Pennsylvania State University, State College, PA, 16801, USA, kislal.orhan@gmail.com; Xulong Tang, Pennsylvania State University, State College, PA, 16801, USA, xzt102@psu.edu; Mahmut Taylan Kandemir, Pennsylvania State University, State College, PA, 16801, USA, mtk2@psu.edu; Meenakshi Arunachalam, Intel, Hillsboro, Oregon, 97124, USA, meena.arunachalam@intel.com.

2476-1249/2019/6-ART38 $15.00
https://doi.org/10.1145/3326153

research on computer architecture [14, 28, 29, 37], optimizing/parallelizing compilers [6, 17, 44, 45, 47] and runtime systems/OS [34, 35, 46] helps us extract increasingly more performance from modern architectures, but their impact is hampered by ever-growing application and hardware complexities. In particular, data accesses in modern architectures pose a significant bottleneck (from both performance and energy consumption angles) as they go through multiple layers in hardware, each with its own management strategy. Indeed, there is a growing concern that "memory wall" [55] can be the main factor preventing many important applications from achieving their full potential, even if we employ sophisticated code parallelization and data optimization techniques. Thus, there is a motivation for exploring revolutionary approaches instead of evolutionary ones.

Many workloads in different application domains can live with a "less-than-perfect" output quality. The application programmers are usually provided with metrics to evaluate the output quality of a particular application. For instance, in video encoding/decoding applications, Peak Signal to Noise Ratio (PSNR) is a metric that is widely used to measure the quality of lossy video compression. In this paper, we target the application domains where the programmers have the capability to determine the error bound of application outputs. Observing this, performance and energy benefits that arise from deliberate use of so-called "approximate computing" has recently been explored across software and hardware stacks [1, 8, 11, 15, 24, 38, 42, 50]. Approximate computing can be a promising solution in overcoming the inherent performance and energy scalability issues faced by current parallel hardware and software systems due to data accesses. For example, skipping some computations can save both computation cycles/energy and cache/memory cycles/energy, which could not be achieved by employing conventional code/data restructuring or hardware enhancements alone. In fact, dropping computations/data accesses can be one of the ways to take the performance/energy scalability of parallel workloads to levels that are not possible through evolutionary approaches.

While prior art [34, 36, 51] in approximate computing focused on point solutions that typically trade off accuracy (output quality) with performance and energy benefits, existing works do not target evaluating the maximum potential of approximate computing or proposing practical schemes that can come close to this potential. In particular, we believe that the prior research leaves three critical questions regarding approximate computing unanswered, especially in the context of dropping/skipping costly data (cache/memory) accesses in data intensive applications:

- What is the maximum potential of skipping (i.e., not performing) data accesses under a given inaccuracy (output quality) bound?
- Can we simply identify the data accesses to drop randomly, or is being architecture aware (i.e., identifying the "costliest" data accesses with respect to a given architecture) critical?
- Do two executions that skip the same number of data accesses always result in the same output quality (error)?

Motivated by these questions, we make two main **contributions** in this work:

- First, we explore the potential benefits of a form of approximate computing that drops select data accesses during the execution of parallel workloads on emerging network-on-chip (NoC) based manycore systems. The unique aspect of this evaluation is its "architecture awareness". That is, given a bound on inaccuracy (the minimum level of program output quality that can be tolerated by user/execution environment), we quantify the benefits of dropping the "costliest" data accesses (in our manycore architecture), as opposed to dropping data accesses "randomly". Our experiments with ten different multithreaded workloads indicate that *being architecture aware* in dropping data accesses pays off, resulting in 27% additional performance improvement, over randomly dropping the same number of data accesses. Unfortunately, our results also indicate that two different executions of a given application that drop the same number of data accesses

can result in quite different output quality values (errors), which makes it difficult to maximize performance under a given error bound.

- Second, motivated by this last observation above, we propose a "program slicing" based approach that identifies the set of data accesses to drop such that we (i) maximize the resulting performance/energy benefits and (ii) remain within the error (inaccuracy) bound specified by the user. Our slicing based approach first uses backward slicing and then forward slicing to decide the set of data accesses to drop. Our experimental evaluations of this slicing-based approach under multithreaded workloads and a cycle-accurate manycore simulator show that, when averaged over all ten benchmark programs we have, 8.8% performance improvement and 13.7% energy saving are possible when we set the error bound to 2%, and the corresponding improvements jump to 15% and 25%, respectively, when the error bound is set to 4%. We also tested a restricted version of our approach on a commercial manycore system, and observed 6.8% and 11.2% average performance improvements with error bounds of 2% and 4%, respectively.

The remainder of this paper is structured as follows. The next section presents our target manycore architecture, and discusses different types of potential "data localities" in this architecture. Section 3 explains the idea of "dropping data accesses" at a high level, and Section 4 presents our experimental setup and benchmark programs. Section 5 gives a quantitative analysis of the output quality– performance/energy saving tradeoff by varying the number and locality category of the data accesses to be dropped. Section 6 discusses our program slicing-based approach to approximate computing, and provides experimental evidence that shows its effectiveness in practice. Section 7 discusses the related work, and finally, Section 8 concludes the paper.

2 TARGET MANYCORE ARCHITECTURE, DATA ACCESS, AND DIFFERENT LOCALITIES

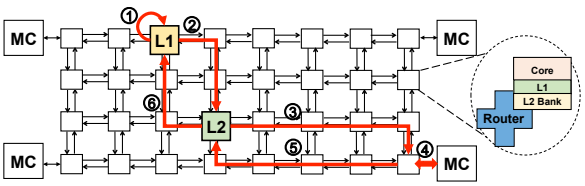
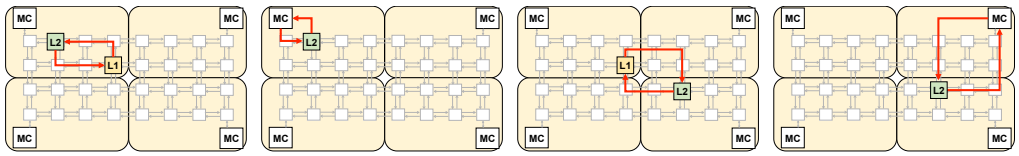


Fig. 1. Representation of a 4 × 8 NoC system with and SNUCA based memory access flow.

Since one of the goals of this work is to measure the limits of architecture-aware approximate computing, we need to explain the "cost of a data access" from an architecture viewpoint. Figure 1 shows the architecture of a state-of-the-art network-on-chip (NoC) based manycore system (similar to Intel KNL [40] and many emerging commercial manycore systems) and the flow of a data access in it under the SNUCA last-level cache (LLC) management policy[1]. First, the local L1 cache is looked up (①), and if a miss occurs, an L2 bank (which is determined based on the physical address of the data being requested) is accessed (②). If we hit in the target L2 bank, the data is read and transferred to the local L1 of the requesting core (⑥). If not, an off-chip memory access needs to be performed. For this, first, a target memory controller (MC) is determined (again, based on the physical address of the data being requested) and we access the target bank governed by that

[1]In SNUCA [13], each data block is statically mapped to an L2 cache bank (called its "home bank") based on its physical address. A core/node that requires a specific data item brings it from its home bank if it is in the cache. If not, the main memory is accessed, again using the physical address of the data. Note that although our target system has two levels of cache hierarchy (L2 being the last-level cache), our approach can work with cache hierarchies of any depth.



(a) L2 hit (SQ). (b) row-buffer hit/miss (SQ). (c) L2 hit (DQ). (d) row-buffer hit/miss (DQ).

Fig. 2. Various data localities in an on-chip network based manycore. Note that the difference between a row-buffer hit and a row-buffer miss manifests itself in the off-chip memory.

controller via the on-chip network (③). Each bank is equipped with a row-buffer that holds the most recently accessed memory row (also called page); typically, a row-buffer hit takes much less time than a row-buffer miss, as the latter requires an access to the memory array itself. After the data is read (④), it is returned to L2 (⑤), the home bank for the requested data, and then to L1 (⑥).

In this architecture, at a high level, one can distinguish among at *least* four different types of *data localities*, depending on where the data being sought is found: L1 hit, L2 hit, row-buffer hit, and row-buffer miss. The actual latency numbers for these different localities may change from one architecture to another. More interestingly, the latency to be incurred in the last three cases may *not* be uniform, as the distance to target L2 bank and target MC may vary (on the network), depending on the relative locations of the requesting node/core and target L2 bank/MC. Therefore, not all L2 accesses (and similarly not all MC accesses) have the same latency, and consequently, one can have a spectrum of L2 and MC latencies from a given core's perspective. For the purposes of this work however, we consider seven different types of *localities*: L1 hit, L2 hit (SQ), L2 hit (DQ), row-buffer hit (SQ), row-buffer hit (DQ), row-buffer miss (SQ), and row-buffer miss (DQ). In this naming, SQ denotes the "same quadrant", i.e., when the requesting core and the target L2 are in the same quadrant of the network (in the case of an L2 hit) and when the requesting L2 and the target MC are in the same quadrant (in the case of an L2 miss). DQ on the other hand captures the case when the core (resp. L2) and target L2 (resp. MC) are in different quadrants. These cases are captured in Figure 2. In the rest of this paper, for brevity, these seven types of data localities (also called "categories", or "groups" in this work) are denoted using C_1 (for L1 hit), C_2, C_3, C_4, C_5, C_6, and C_7 (for row-buffer miss (DQ)). Clearly, everything else being equal, for a given data access in our manycore architecture, we would prefer to have C_i instead of C_{i+1} ($1 \leq i \leq 6$) from a data locality viewpoint. In fact, many previously-published software and hardware works targeting "data locality" (in the context of both single-core and manycore systems), try to move as many data accesses as possible, from C_j to C_i where $1 \leq i < j \leq 7$.

3 DROPPING DATA ACCESSES

While there exist different approaches [11] to approximate computing, each leading to a different tradeoff between performance/energy benefits and program output accuracy, in this work we use "skipping/dropping data accesses". Three other possible approaches are (i) dropping computations, (ii) dropping synchronizations, and (iii) reducing the number of bits used to implement individual data elements. In loop perforation [39], computations (some loop iterations) are dropped in a systematic fashion. While this computation skipping technique is quite effective in some applications, in general dropping an iteration means dropping a considerable number of data accesses (especially in the case of large loop bodies) along with the computations that operate on them, and consequently, its impact on the accuracy of the application can be very significant. As a result, one may prefer to employ a *finer granular* technique, e.g., data access skipping that is used in this work. An alternate

approach to approximate computing is to reduce the number of synchronizations in the application code. Many high-performance computing applications are implemented in a multithreaded fashion and execute using a very large number of threads (sometimes tens of thousands). In such cases, a barrier synchronization (which is meant to synchronize all threads) can be very costly latency wise. In [32], the barrier is opened right after a certain fraction of threads reach it, instead of waiting for all threads to reach it. One problem with this approach is that it may not be very effective in applications where there are not lots of synchronizations. Even in applications with lots of synchronizations, it is hard to predict the impact of relaxing barriers on program accuracy. The third alternate technique, reducing the number of bits to represent values of individual data elements [33], while can be effective in certain application domains, may not be the best fit for other applications that need full bit width. Motivated by these observations, in this work, we use "data access skipping", which is finer grain than heavy-handed computation skipping, and is applicable to programs with floating point (as well as integer) data and those with no/few synchronizations.

We want to emphasize that, while dropping data accesses can lead to program crash in certain cases, this is *not* expected to be the case in our target application domain. Basically, in this work, we focus on array and loop dominated programs from high performance computing and embedded image/video processing. These codes are generally written as a series of loop nests, and each nest having a large body of instructions. The specific execution path taken by the program is mostly a function of the input size but is not dependent on the actual values of the inputs. Thus, changes in the values of the data elements (which is an effect of our approach, as we supply a "value" for each data access we drop) does *not* cause an otherwise correct program to crash.

The goal behind data access dropping/skipping is to drop the right number of data accesses to maximize the performance/energy benefits and at the same time remain within the limits of "acceptable output inaccuracy" (output quality). Clearly, the latter is a function of the application/workload characteristics and can sometimes even change from one execution environment (e.g., user constraints, input) to another, for the same application program. This paper is based on a simple yet important observation:

> *If we are allowed to drop a certain amount of data accesses so that the program's output quality is still acceptable, to maximize performance/energy benefits, we may want to drop the "costliest" data accesses first.*

Considering our classification of data locality in Section 2, we want to first drop the accesses in the C_7 category; if we can continue to drop more data accesses (and still remain within the user-specified error bound), they should be picked from the C_6 category; and so on. That is, the observation above tells us that, in dropping data accesses, we should be "cost/architecture aware". However, there is *no* guarantee that, two executions that drop the same number of references with exactly the same cost would result in the same inaccuracy. In other words, just adopting, as our quality metric, the number of the data accesses to be dropped or their associated costs may not guarantee that the resulting output accuracy of the application program would be acceptable from a user's perspective. This is because the inaccuracy resulting from dropping data accesses is not only a function of the number of data accesses dropped but also a function of which *specific* data accesses are dropped.

Motivated by this observation, *the rest of the paper makes two major contributions.* First, we perform a quantitative analysis of the tradeoff between performance/energy benefits and inaccuracy (QoS) under varying amounts of data access skipping. The goal of this analysis is to demonstrate the importance of dropping the costliest data accesses (as opposed to, say, "randomly" selecting the data accesses to drop). We also investigate the variation in the program output qualities of two different executions of the same program, when in both the executions the same number of

Table 1. Target system configuration.

Manycore Size, Frequency	32 (4 × 8), 1 GHz
L1 Cache	16 KB; 8-way; 32 bytes/line
L2 Cache	512 KB/core; 16-way; 64 bytes/line
Coherence Protocol	MOESI
Router Overhead	3 cycles
Page Size	2 KB
On-Chip Network Frequency	1 GHz
Routing Strategy	XY-routing
DRAM	DDR3-1333 (9-9-9); 250 request buffer entries; 4 MCs 1 rank/channel; 16 banks/rank
Row-Buffer Size	2 KB
Address Distribution across LLCs	64 bytes
Address Distribution across banks	64 bytes
Epoch Length	256 cycles

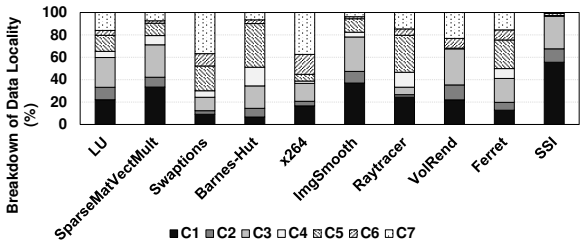


Fig. 3. Data locality breakdown of our multithreaded application programs.

(but different) data accesses are dropped. Our results show that this variation can be quite high. Motivated by this, we then propose a "program slicing" based strategy that drops the right set of data accesses to guarantee the inaccuracy bound specified by the user/programmer. Unless stated otherwise, *when we say "a data access is dropped", we mean that the corresponding access is not performed and instead a value of "0" is assumed for that access.* Later, we discuss an alternate strategy (based on application profiling) to supply the missing values.[2]

4 BENCHMARKS AND EXPERIMENTAL SETUP

The program slicing technique (and the required code analysis) used in this work is implemented within the LLVM framework [19]. Most of our experiments are conducted using a cycle-accurate manycore simulator. More specifically, in our experiments, we use the GEM5+McPAT [3, 20] combination to collect performance and energy statistics. Note that the reported energy numbers include the energy consumed by all CPU components, caches, TLBs as well as main memory. The important parameters of the modeled on-chip network based manycore system along with their default values are listed in Table 1. Later in our experiments we modify the values of some of these parameters to conduct various sensitivity experiments. We preferred a simulator-based analysis, as it is difficult in current architectures to be able to drop data accesses at a fine granularity (dynamic instance based) and it is also difficult to isolate the performance of data accesses. Nevertheless, we also tested a restricted version of our approach on a state-of-the-art manycore architecture [40], and report the collected results in Section 6.5.

[2]We want to emphasize that only the mentioned alternative strategy uses "profiling"; our original approach that supplies "0" for the dropped accesses does not need profiling.

Table 2. Our multithreaded workloads and their salient characteristics.

Benchmark	Source	Error (Quality) Metric	Input Dataset Size (MB)
LU	Splash-2 [54]	Relative difference from standard output	565.8
SparseMat VectMult	local	Relative difference from standard output	521.2
Swaptions	Parsec [2]	Relative difference from standard output	487.5
Barnes-Hut	Splash-2 [54]	Relative difference from standard output	663.3
x264	Parsec [2]	PSNR	702.6
ImgSmooth	local	PSNR	712.4
Raytracer	Splash-2 [54]	Pixel Difference	762.5
VolRend	Splash-2 [54]	Pixel Difference	522.9
Ferret	Parsec [2]	Classification accuracy	698.7
SSI	local	Ranking accuracy	608.7

In this study, we used 10 multithreaded benchmark programs[3]. In our experiments, only one multithreaded application is executed at a time (using all 32 cores). The second column in Table 2 gives the sources of our benchmark programs and the third column gives the error (output quality) *metric* used for each benchmark. In this column, "Relative Difference" is the root-mean-square error of the output. Note that, the same error metrics have also been used by prior research on approximate computing [7]. For *x264* and *ImagSmooth*, we used Peak Signal to Noise Ratio (PSNR) as error metric to measure the quality of output images. For *Ferret*, we used classification accuracy (calculated by the average of similarity differences). It can be observed from this column that our workloads employ a variety of error (quality) metrics. Finally, the last column of this table gives the total amount of input dataset in MBs. We start by presenting the breakdown of data accesses for each benchmark program we have into the seven locality categories ($C_1 \cdots C_7$) explained earlier in Section 2. The results plotted in Figure 3 indicate that our applications exhibit great variety in terms of the data locality they exhibit. For example, locality behaviors of *ImgSmooth* and *SSI* are quite well, with very few data accesses fall in the $C_4 - C_7$ range. In comparison, applications such as *Barnes-Hut* and *x264* exhibit poor data locality, with nearly most of their data accesses falling in the $C_4 - C_7$ range.

5 EVALUATION OF THE PERFORMANCE/ENERGY BENEFITS AND OUTPUT INACCURACY TRADEOFF

In this section, we evaluate our benchmark programs in terms of **four different metrics** while varying the fraction of data accesses to be dropped: *memory access latency improvement*, *overall performance improvement*, *energy savings*, and *output error* (*inaccuracy*). All these metrics are given as relative (percentage) values with respect to the **default** execution where "no" data access is dropped (in this default case, the program output quality is assumed to be perfect). Recall that the third column of Table 2 gives the error (quality) metric for each benchmark. In the x axes of the graphs presented below, "5% of C7" refers to an execution where 5% of the data accesses in the C_7 category (as defined in Section 2) are dropped. Similarly, "100% of C7 + 5% of C6" captures an execution where all of the C_7 type of data accesses as well as 5% of the C_6 type of data accesses are dropped, and so on. Note that the percentage performance (energy) improvement values represent the percentage reduction/saving over the execution time (energy consumption) of the default execution. Also, each result presented below in Figures 4, 5, 6, and 7 represents the "median values" collected from 20 different executions. Finally, in an execution where we are to drop a certain

[3]These benchmarks have been fully optimized for parallelism and data locality using O3 compiler flag.

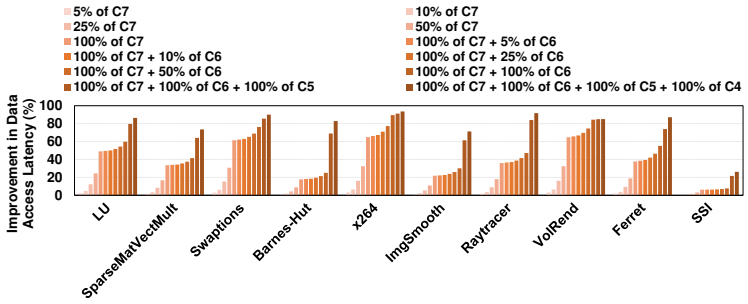


Fig. 4. Improvements in data access latencies.

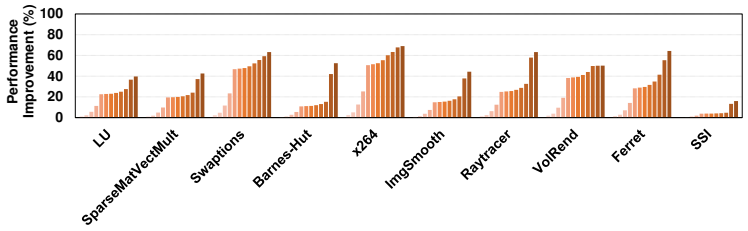


Fig. 5. Performance improvements. The legend is the same as in Figure 4.

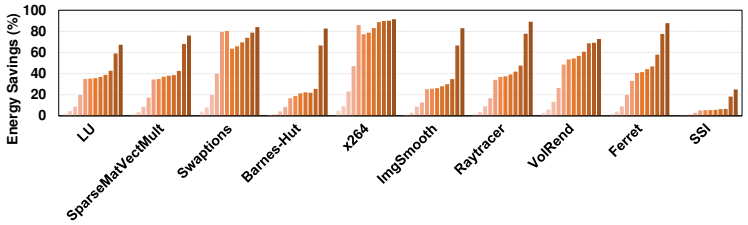


Fig. 6. Energy savings. The legend is the same as in Figure 4.

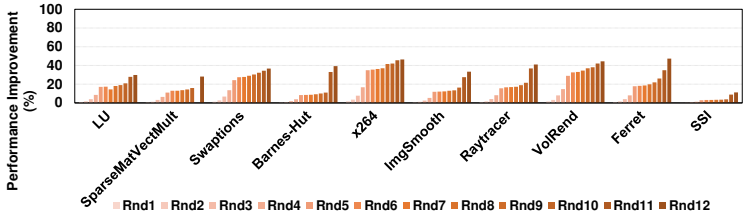


Fig. 7. Performance improvements.

fraction of data accesses, those accesses are determined *randomly*. For example, if, say, we are dropping 10% of the C_7 type of data accesses, that 10% accesses are selected randomly (from among all C_7 type of accesses).

In the discussion below, we try to answer three important questions: (i) *how much performance and energy can one save by dropping the "costliest" memory accesses?*; (ii) *how do these benefits compare to an alternate strategy that simply drops the data accesses "randomly", i.e., without considering which*

locality category they belong to?; and (iii) *is the magnitude of performance and energy savings obtained via data access skipping a function of only the amount of the data accesses to drop, or also of the data accesses themselves?*

The graph in Figure 4 gives the percentage improvement in the average "data access latency" under different execution scenarios (the x-axis represents the fraction and category of the data accesses dropped). As expected, data access latencies drop, as we increase the number of data accesses to drop. However, improvement trends vary across different workloads. For example, *SSI* does not exhibit significant improvements except in the last two executions. In contrast, *LU* and *VolRend* show reasonable improvements in memory access latencies, even when the number of data accesses dropped is not too high. While data access latency is certainly an important metric, one would ultimately be interested in overall application performance, which is plotted in Figure 5. This graph gives the percentage reduction in the parallel execution time (over the default execution). Clearly, different applications get different benefits due to the different contributions of the memory accesses in them to their overall performance. Still, one can observe a common pattern where the performance benefits are initially not very high but they show a significant jump beyond a certain point. For example, in *VolRend*, the performance benefits jump significantly as we move from "50% of C_7" to "100% of C_7". A similar jump can be observed in *x264* and *Barnes-Hut* as well.

Figure 6 plots the energy benefits (given as percentage reduction in system-wide energy consumption over the default execution) when dropping various number of data accesses (as specified on the x-axis). We see that these results are in general higher than the performance improvement results presented in Figure 5. This is primarily because, in the default execution, a costly data access can overlap with other (ongoing) data accesses or computations. Consequently, dropping such an access may not return much benefits in practice. In contrast, while performance overhead of a data access can be hidden in parallel execution, its energy overhead cannot be; and, as a result, dropping a data access gives more energy benefits than performance benefits, when compared to the default execution. Overall, the results presented in Figures 5 and 6 clearly indicate that *dropping costly memory accesses can give significant performance and energy benefits.*

Let us now try to answer the second question raised above, namely, whether dropping the costliest accesses (as opposed to other accesses) is really important, as far as the performance and energy benefits are concerned. Figures 7 and 8 give, respectively, the performance and energy benefits (again with respect to the default execution), when we select the data accesses to drop *randomly* from all data accesses[4]. In these two charts, each point on the x-axis corresponds to an execution that drops the "same number" of data accesses as the corresponding point in Figures 5 and 6. For example, "Rnd2" in Figure 7 drops the same number of data accesses as "10% of C7" in Figure 5, and "Rnd11" drops the same number of data accesses as "100% of C7 + 100% of C6" in Figure 5. Comparing the results in Figures 7 and 8 with the corresponding results in Figures 5 and 6 clearly indicates that *one needs to be very careful in selecting the data accesses to drop,* as can also be seen from the geometric mean results plotted in Figures 9 and 10. For example, the execution with "10% of C_7" drops generates about 27% additional performance improvement and 31% additional energy improvement compared to randomly dropping the same number of data accesses.

We next focus on output quality of our application programs and quantify it when dropping various number of costly data accesses. There can be different metrics to quantify the output quality of an application, and in general two applications may have different metrics to quantify their output qualities. As stated earlier, in this work, we used the metrics listed under the third

[4]It is to be noted that, in the earlier case (Figures 5 and 6) where we dropped the costliest references, we use random selection *only* from a given category. For example, when we talked about dropping 10% of the C7 category of data accesses, that 10% is selected randomly (but *only* from the C7 category). In contrast, here, the data accesses to be dropped are selected randomly from *all* data accesses.

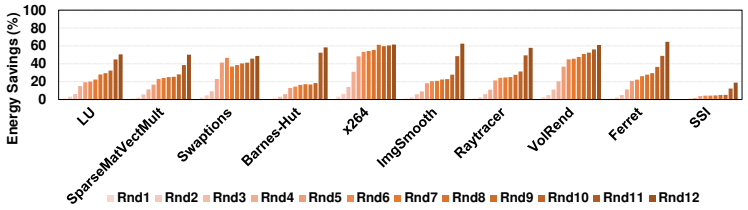


Fig. 8. Energy savings.

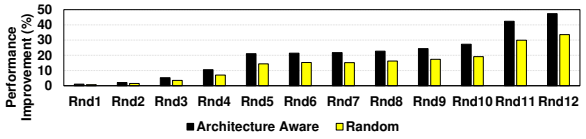


Fig. 9. Geometric mean of performance improvements across all applications. Note that, each point on the x-axis represent different entities for locality aware and random schemes. In the latter, it represents random selections of the data accesses to be dropped. In the former, it represents the corresponding point in Figure 5.

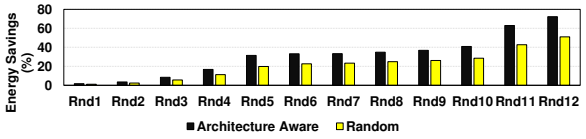


Fig. 10. Geometric mean of energy savings across all applications. Note that, each point on the x-axis represent different entities for locality aware and random schemes. In the latter, it represents random selections of the data accesses to be dropped. In the former, it represents the corresponding point in Figure 6.

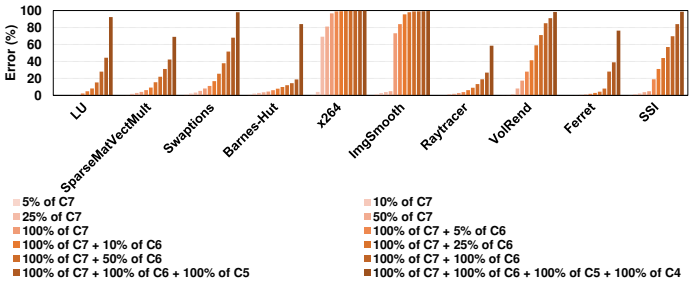


Fig. 11. Error (program output inaccuracy) values.

column of Table 2. The y-axis in Figure 11 represents the percentage error experienced over the default execution (reduction in output quality). As in the case of performance behavior, in most benchmarks, the error jumps significantly beyond a point (which depends on the benchmark). For instance, in *Ferret*, the error moves from 8% to 28%, as we move from "100% of C7 + 50% of C6" to "100% of C7 + 100% of C6".

We now turn our attention to the third question raised above. When considering both performance improvements (Figure 5) and corresponding errors (Figure 11), one might think that, it could be possible to decide the "ideal number of data accesses" to skip in order to save as much performance and energy as possible while staying within an acceptable error bound. As an example, in *Swaptions*,

Table 3. Distribution of output errors in *LU*.

	5% of C7	10% of C7	25% of C7	50% of C7	100% of C7	100% of C7 +5% of C6	100% of C7 +10% of C6	100% of C7 +25% of C6	100% of C7 +50% of C6	100% of C7 +100% of C6	100% of C7+100% of C6 +100% of C5	100% of C7+100% of C6 +100% of C5+100% of C4
minimum	0.02	0.06	0.09	0.14	0.2	1.13	2.26	3.87	8.28	19.68	28.06	89.2
median	0.05	0.09	0.15	0.25	0.33	2.33	4.84	8.16	15.27	28	44.4	92.3
maximum	1.9	3.34	5.87	8.11	11.14	14.72	16.67	18	24.45	41	62.22	98

Table 4. Distribution of output errors in *VolRend*.

	5% of C7	10% of C7	25% of C7	50% of C7	100% of C7	100% of C7 +5% of C6	100% of C7 +10% of C6	100% of C7 +25% of C6	100% of C7 +50% of C6	100% of C7 +100% of C6	100% of C7+100% of C6 +100% of C5	100% of C7+100% of C6 +100% of C5+100% of C4
minimum	0.13	0.25	0.77	4.31	12.05	18.27	28.45	45.04	56.12	68.56	78.16	94.02
median	0.23	0.54	1.34	8.13	17.41	28.01	41.4	59	71.11	85.05	91	98.34
maximum	2.22	3.16	4.49	12.68	24.24	37.78	53.7	72.67	84.21	96.16	97.71	99.12

if the maximum error that can be "tolerated" is 10%, we can see that "100% of C7" is the right amount to drop, and this leads to about 46.7% improvement in overall application performance. A similar analysis can be carried out for any application program and any given error bound. However, in practice, this approach may *not* be easy to implement. This is because, as indicated earlier, the results presented in Figure 6 represent the "median values" from 20 different executions. Table 3 and Table 4 give the "distribution" of the errors from our 20 different experiments for *LU* and *VolRend*, respectively. The error results are given in these tables as minimum, median and average error values. One important observation from these results is that, there are "significant variations" across error values. To be more specific, two different executions of an application that drop the same number of data accesses from the same set of categories (localities) can lead to significantly different errors in the output of an application. Consequently, it may *not* be possible to simply look at the performance and error plots, and determine the right amount/type of data access(es) to drop. Furthermore, this result also means that, it is *not* trivial to put a bound on error when a certain number of data accesses are dropped. Motivated by these observations, the next section proposes and experimentally evaluates a "program slicing" based strategy to determine the data accesses to drop such that (i) *performance benefits are maximized* and (ii) *we remain within a user-specified error bound.*

6 PROGRAM SLICING-BASED DATA ACCESS REMOVAL

We now discuss our program slicing-based approach employed to decide the set of variables to drop such that the performance is maximized and the accuracy loss is within a specified error bound. At a high level, our approach computes, using backward and forward slicing (explained below) as well as target error metric, the performance gains when accuracy of output elements are relaxed. Among all possible relaxations, our approach then selects the one that generates the maximum performance benefits while staying within the specified error bound.

6.1 Program Slicing Basics

Program slicing is a source code analysis strategy that identifies code segments relevant to a given program point in the target code base. The initial idea was proposed by Weiser for the purpose of simplifying debugging [53]. Slicing is also used in program differencing, optimization, maintenance, and flowback analysis [48]. While the original concept ensures that a code slice is a syntactically valid program, more recent algorithms relax this constraint to provide more accurate (smaller) slices. There are two main forms of program slicing. *Static slicing*, which is the one employed in this work, uses only statically measurable information and it is applicable for any execution of the

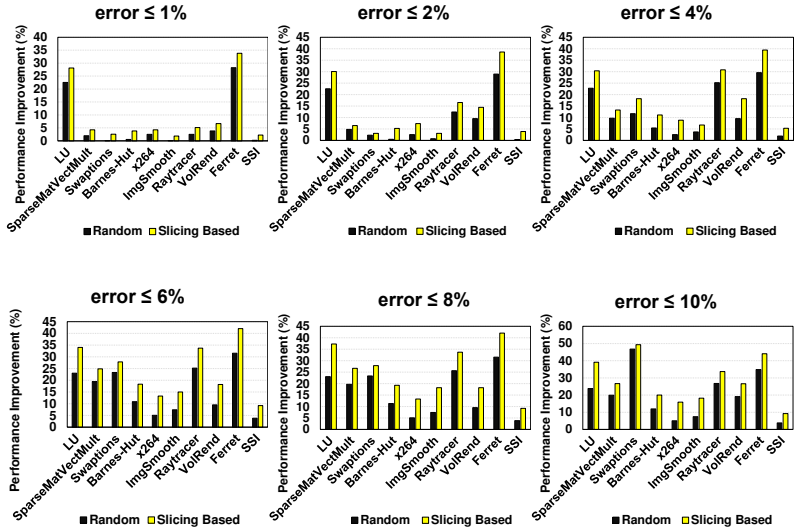


Fig. 12. Performance improvements.

program. On the other hand, *dynamic slicing* is tailored to a particular execution of the program which requires a fixed set of input values [18].

To explain how a code slice is obtained, it is important to define the following *dependency* concepts. Let P_1 and P_2 be two program points. P_2 has a "data dependency" on P_1 if there exists a variable v that satisfies all of the following conditions: (i) P_1 modifies the value of v; (ii) P_2 reads the value of v; and (iii) There is an execution path from P_1 to P_2 on which the value of v is not overwritten. Note that, slicing captures "indirect dependencies" as well. For instance, in code sequence $\{S1 : u = v + 1; S2 : w = u - b; S3 : t = w * w - 1;\}$, the indirect dependence between statement S1 and statement S3 is also captured (in addition to two direct dependencies, one between S1 and S2, and the other between S2 and S3). On the other hand, P_2 is said to have a "control dependency" on P_1 if at least one of the following conditions is true: (i) P_1 is a condition statement, and reaching P_2 depends on P_1; (ii) P_1 is a function call, and P_2 is one of the parameters of the call; (iii) P_1 is a function call, and P_2 is the entry point of a function called by P_1; and (iv) P_1 is a function entry point, and P_2 is one of the formal parameters, one of the declarations, or one of the top-level statements in the body of the function.

A program slice can be expressed as a set of data and control dependencies. There are two "directions" a program slice can be extracted since a dependency requires two actors. *Backward Slice* of a given point is the union of the points it depends on. Similarly, *Forward Slice* of a point is the union of its dependent points. Since both of these are calculated are using the same dependency relations, our explanation below mostly focuses on backward slicing.

Figure 13(a) shows a sample code fragment, and Figure 13(b) shows a static backwards slice from line 14, variable x (denoted as $< 14, x >$). $< 14, x >$ has a data dependency on lines 6 and 8 as they modify the value of x. These two lines are controlled by the if statement (lines 5 and 7) and the whole block is controlled by the while statement (lines 4 and 13). Finally, the while condition is data dependent on lines 1, 3, and 12. The remaining lines (2, 9, 10, 15, and 16) are safely omitted from the slice. A dynamic backward slice from the same point may or may not be the same as its static counterpart. Assuming the value of n is 1; line 8 will *not* be included in the slice, since the control never reaches that point in the first place.

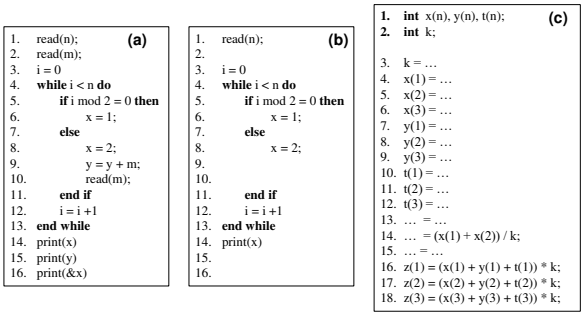


Fig. 13. (a) An example code fragment to illustrate program slicing. (b) Static backward slice from Figure 13a for line 14, variable x. (c) An example code segment.

Note that a backward slice may contain lines after that particular point (and vice versa for forward slice). For example, a backward slice from $< 9, m >$ will include line 10, as its result does effect line 9 in a different iteration of the while loop. In addition, it is important to note the distinction between values and their addresses. As an example, $< 16, \&x >$ does not depend on lines 6 or 8 (or any other line) since the actual address of the variable does not change and the value is irrelevant for this particular point in the program.

The calculation of a forward slice follows the data and control dependency chains in the *reverse direction*. For example, static forward slice of $< 2, m >$ contains lines 9 and 15. Line 9 depends on the value of m and, even though m it is overwritten at line 10, this dependency propagates to line 15 through the value of y. Forward slicing is a useful method to classify points based on their potential impact on the rest of the program. This classification may lead to various optimizations, e.g, which variable should have less precision?, which calculation should be approximated?, etc.

Program slicing has been extended to handle iteration spaces and multi-dimensional arrays by Pugh and Rosser [30]. In our work, we closely follow their implementation of program slicing. Specifically, we generate, without completely unrolling the loop nests, an iteration space slice, if the loop bounds and array subscript expressions are affine functions of outer loop bounds and symbolic constants. In more detail, the backward iteration space slicing takes, as input, the dependency information (extracted by the compiler) and determines all statement instances (in loops) that need to be executed to produce the values of a given array access. Consequently, a slice obtained in this way can be thought of as a "dependence chain" that can be used to reach all statement instances which can affect the result. In the next section, we give the technical details of our slicing-based compiler algorithm.

6.2 Slicing for Approximate Computing

Algorithm 1 gives the pseudo-code of our slicing-based compiler algorithm. The complexity of the algorithm is $O(NM^2)$ where N is the number of outputs and M is the number of statements involved in backward slicing and forward slicing. At a high level, our algorithm first extracts a backward slice for each element of the output data structure (line 36). After that, for each left-hand-side variable involved in any statement in the backward slice, it determines the forward slice (line 37). If that forward slice includes any other statement other than the one we started the backward slice from, we mark that variable as "non-approximatable". The remaining variables on the other hand (i.e., the variables that can be approximated/skipped if we accept error in the output element in question) constitute the "approximation set" of that output data element (line 38). Then, for each data element (access) in this set, we estimate its cost in terms of the number of cycles it

is expected to take in the target architecture (using the *Cost_Evaluation* function) (line 56), and subsequently find the total number of cycles that can potentially be *saved* if we accept error in that output element. This process is then repeated for each output element, and after all the output elements have been processed, we have the "approximation set" of each output data element. In the next step, we go over all variables in "approximation set" and identify the k most beneficial output data elements (accesses) to drop, where k is determined by the total number of output data elements and the acceptable error value specified by the user (err). We want to emphasize that our approach guarantees that the user-specified error bound is satisfied, i.e., among all options, we pick k elements (to drop) that satisfy the error bound and generate maximum savings. Therefore, our program slicing based approach is also "cost (architecture) aware" in that it drops that data accesses that return the most benefits. Note also that, since we carry out our compiler analysis at a data element granularity (actually, if desired, its granularity can be tuned), we catch the opportunities for dropping data accesses even if the same data element is accessed by different static load instructions in the code.

In this work, we implemented the *Cost_Evaluation* function by extending a cache miss estimation scheme called CME (cache miss equations [9]). CME is originally designed to give a detailed representation of *cache* behavior, including conflict misses, in loop-oriented application programs.[5] We extended the original CME implementation to capture coherence misses (in multithreaded execution), misses in cache hierarchies of any depth, as well as row-buffer hits/misses, and as a result, *our approach can classify any given data access in the target application program into one of the seven data locality categories (C_1 through C_7), described earlier in Section 2.* Our experiments with this CME implementation indicated that its accuracy (in correctly classifying data accesses into our seven categories) was about 89%, when averaged over all 10 benchmark programs in our experimental suite. That is, CME is highly accurate in identifying the category to which a data access is mapped.

To illustrate how we employ program slicing in identifying the costly data accesses to drop, let us consider the sample code fragment shown in Figure 13(c). In this code fragment, x, y, and t arrays as well as parameter k are assumed to be inputs, and array z is assumed to be output. For illustration purposes, let us assume that we can afford only one of $z(1)$, $z(2)$ and $z(3)$ to have a wrong value (i.e., an error bound of 33%). If the accesses to be dropped are determined randomly, there is *no* guarantee that we will achieve our 33% error target (this holds true even if we drop the costliest accesses). For example, if we happen to drop, say, $\{ x(1)$ and $y(2) \}$ or just k, *both* $z(1)$ and $z(2)$ will result in error (giving an overall error rate of 67%, which is *not* acceptable per the user-specification).

Our backward slicing-based approach, on the other hand, operates differently. It starts with $z(1)$, and extracts the backward slice for it, which includes statements 3, 4, 7, and 10. Then, starting with each of these statements, we perform a separate forward slicing, and determine that $x(1)$, $x(2)$, and k are used in statement 14 as well. Consequently, we identify $y(1)$ and $t(1)$ are the *only* data elements (accesses) that can be safely dropped (i.e., they constitute the "approximation set" for $z(1)$), if we accept that $z(1)$ is allowed to have an erroneous value (instead of its correct value). Note that, dropping $y(1)$ and $t(1)$ will force $z(1)$ to have an erroneous value, but it will *not* affect the correctness of either $z(2)$ or $z(3)$ (in this way, the specified error target, 33%, would be satisfied). Similarly, we do the backward slice and forward slice for $z(2)$ and $z(3)$ as well, and identify their approximation sets. At this point, we have, for each of the output data elements, the set of data accesses that can be dropped if that output element is allowed to have an erroneous value.

[5]CME uses reuse vectors across loop statements to generate linear Diophantine Equations. It then uses the solutions of the resultant Diophantine Equations to predict the hit/miss status of the data references.

Algorithm 1 Slicing for Dropping Data Accesses

```
INPUT: Output variables (Oset); Error bound (err);
OUTPUT: Variables that can be approximated
function BACKWARD_SLICING(v, s)
    output ← ∅
    /** start from point s in the program */
    for each statement s_i do
        if s has read after write (RAW) dependence with s_i then
            output ∪ s_i
        else if s_i has control dependence with output or s then
            output ∪ s_i
        end if
    end for
    return output
end function

function FORWARD_SLICING(Mset, s)
    output ← ∅
    /** start from modification points Mset */
    for s_i from Mset do
        for each statement s_j where j ≠ i do
            if s_j has read after write (RAW) dependence with s_i and s_j ≠ s then
                /** s_j is an intermediate reuse of variable v*/
                output ∪ s_i
            end if
        end for
    end for
    return output
end function

/** Initialization */
Iset ← ∅ /** Input variables set is initialized to empty */
DropSet ← Oset
while DropSet ≢ ∅ do
    o_i from DropSet
    s_i = get_statement(o_i)
    Iset ← get_inputs(s_i)
    for Each v_i in Iset do
        modification ← BACKWARD_SLICING(v_i, s_i)
        intermediate ← FORWARD_SLICING(modification, s_i)
        approximate ← modification ∩ intermediate
        if approximate ≠ ∅ then
            remove oi from DropSet
            break
        end if
        for each s_k in approximate do
            if v_i is the output variable of s_k then
                /** v_i is added as approximatable */
                DropSet ∪ v_i
            else
                remove v_i from Iset
            end if
        end for
    end for
    remove oi from DropSet
end while
/** select outputs to be dropped based on cost evaluation */
while ErrorCalculate(Oset) < Err and each o_i from Oset do
    MostCostlyOuput = Cost_Evaluation(approximate_o_i)
    Drop MostCostlyOutput
end while
```

After all the output elements have been processed in this fashion, we next choose the output element that should actually be approximated (i.e., allowed to have a wrong value).[6] We choose this output element to be approximated as the one with the most potential savings. In our example

[6]We want to emphasize that, since in this example our error target is 33%, we are allowed to have a wrong value for only one of the three output elements. If on the other hand our error bound was 67%, two output elements would be allowed to have wrong values.

code, suppose that our *Cost_Evaluation* function indicates that $z(1)$ and $z(2)$ have a cost of 50% $C1$ + 50%$C2$, whereas $z(3)$ has a cost of 25% $C1$ + 25% $C3$ + 50% $C5$. In this case, our approach decides that (only) $z(3)$ should be *approximated* and consequently $x(3)$, $y(3)$ and $t(3)$ should be *dropped*.

To sum up, our approach, using backward and forward slicing, estimates, for each output data element, what would be the total savings if we are allowed to error in that output element. It then identifies the k output elements (only 1 element in our example above) that collectively return the maximum savings. Once the corresponding set of data accesses to be skipped are determined, the compiler marks and passes them to hardware, which in turn simply drops them and return, instead, a value of '0' for each of them (our compiler also uses loop unrolling in cases where some (dynamic) instances of a given static load are to be dropped whereas some other instances of the same static load are not to be dropped).

It is important to note that, our slicing-based approach takes the error metric of the target program explicitly into account. In fact, error metric is as input to our approach along with the target architecture description and the application program to be optimized. As can be seen from Table 2, while our application programs have a variety of error metrics, they can be divided into three main groups: (i) metrics that are based on the difference between the correct output and approximated output (e.g., pixel difference and relative difference from standard output), (ii) metrics that are based on some value *derived* from the difference between the correct output and approximated output (e.g., PSNR), and (iii) metrics that *cannot* be straightforwardly correlate with the difference between the correct output and approximated output (e.g., classification accuracy and ranking accuracy). Note that (i) naturally translates to the number of output data elements (e.g., number of pixels) approximated; that is, for each approximated output value, we can simply assume that the resulting value is wrong. For (ii) on the other hand, our current implementation assumes the worst case scenario. For example, if the error bound is specified as PSNR (and since PSNR is a function of MSE (mean square error)), we assume that, for each output value approximated, its contribution to MSE is the maximum possible value. That is, when we drop data accesses, our approach assumes the worst impact that doing so can cause in PSNR. The third type of error metrics (iii) is more challenging to handle, as there is a no direct (easy) relationship between the approximated value and classification/ranking accuracy. In this case, our current implementation trains a model that captures this correlation and uses it in deciding how many output elements to approximate.

6.3 Experimental Evaluation of the Slicing-Based Approach

To demonstrate the effectiveness of our proposed slicing based approach to approximate computing, we give in Figure 12 plots showing how much performance improvement our approach and random selection of the data accesses bring *under the same error bound*. The error bound in these experiments takes values from {1%, 2%, 4%, 6%, 8%, 10%}. In obtaining the results captured by the first bar for each application, we used the graph in Figure 5 discussed earlier. More specifically, *for each error bound tested, we identified the maximum performance savings possible*. The second bar in each plot in Figure 12 represents the results from our slicing based approach under the same error bound. This "iso-error" analysis clearly indicates that, instead of randomly dropping the costly data references, employing a slicing based approach can save significantly more performance, the geometric means of the respective improvements with the random strategy and slicing based approach being 3.6% and 8.8%, respectively, when the error bound is set to 2%. The corresponding performance improvements are 12.9% and 21.5%, respectively, when the error bound is set to 6%. The corresponding energy savings under the same error bounds are plotted in Figure 14 (the first bar for each application is derived from Figure 6). It can be seen that, the general trends in energy savings are similar to those in performance improvements. Specifically, when the error bound is set

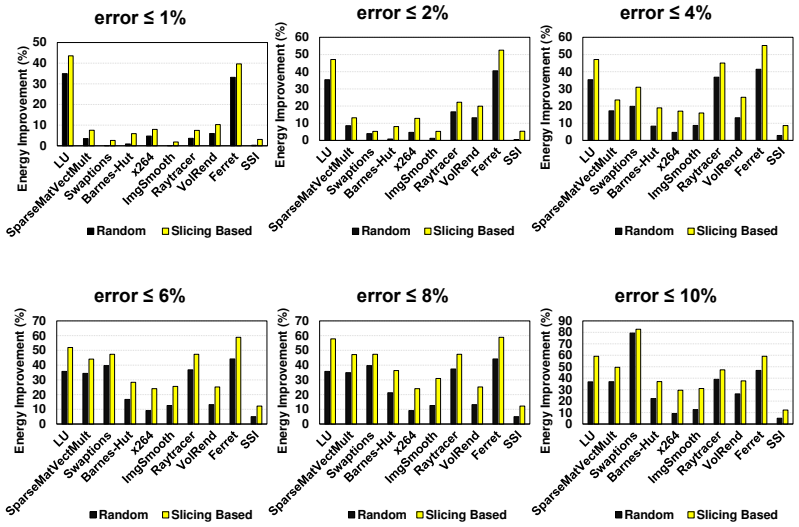


Fig. 14. Energy savings.

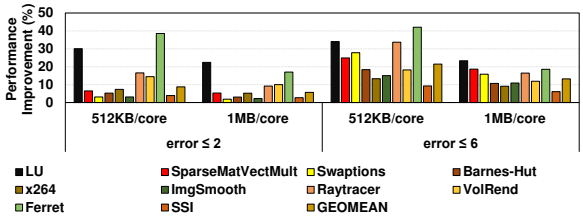


Fig. 15. Impact of the last-level (L2) cache capacity.

to 2%, randomly dropping the costly data accesses and our slicing based scheme generate 5.6% and 13.6% energy savings (geometric means). The improvements jump to 20% and 33.2%, respectively, when the acceptable error bound is increased to 6%.

Next, we change the values of some of the experimental parameters listed in Table 1 and perform a sensitivity study of our slicing based approach. We want to emphasize that, in each of these experiments, the value of only one parameter is changed; the remaining variables maintain their original values shown in Table 1. For ease of presentation, we present our results for only two error bounds (2% and 6%). Also, we present only the performance improvements, as the energy savings follow very similar trends. The first parameter whose value is changed in the last-level cache (L2) capacity, as it directly influences the breakdown of data accesses among our locality groups (C_1 through C_7). Recall from Table 1 that the L2 cache capacity used so far in our experiments was 512KB/core. The results with 1MB/core L2 capacity are plotted in Figure 15 (the results with the 512KB/core are reproduced here for ease of comparison). We see that, a larger last-level cache capacity lowers our savings a bit. This can be expected, as a larger cache capacity captures more data requests, which in turn reduces the number of accesses to the main memory (which also means reduced number of accesses on the on-chip network). The reductions in our savings are more pronounced in benchmarks such as *LU* and *Ferret*, as they are dominated by L2 misses more, compared to the other benchmarks.

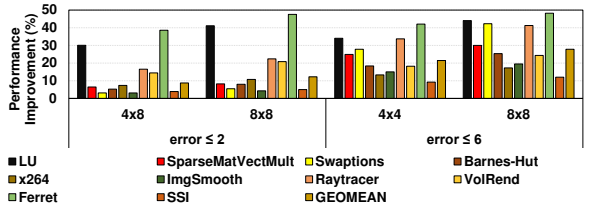
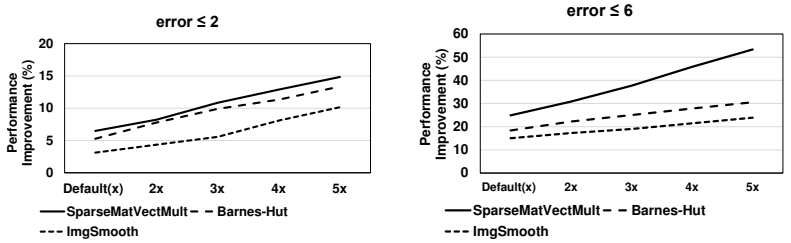


Fig. 16. Impact of network (machine) size.



(a) (b)

Fig. 17. Result with increased input sizes. The x-axis denotes the input size. The default input size for an application, given in Table 2, is denoted using 'x'.

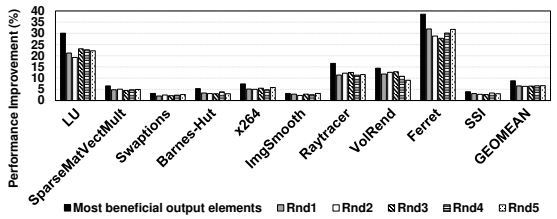


Fig. 18. Performance improvements with six different executions (each bar corresponds to a different execution that drops the same number of elements).

The second parameter we study is the number of cores (network size). Compared to our default value of 4×8, the results with an 8×8 machine size (64 cores), given in Figure 16, exhibit better savings. This is because a larger network increases the time-to-data in the "default execution", which is the primary target of our approximation oriented approach. It is to be noted that, while the default network size generates a geometric mean of improvement of 8.8% in the case of 2% error bound and 21.5% in the case of 6%, the larger network size (8×8) takes these values to 12.3% and 27.8%, respectively.

The next parameter we study is the input dataset size. Unfortunately, in only three of our benchmark programs, we were able to increase the default input data size given in the last column of Table 2 safely. The results presented in Figures 17a and 17b reveal that the effectiveness of our slicing based approach increases as we increase the input size. This is mainly because a bigger input puts more pressure on all on-chip resources (including caches, network and memory controllers) in our target manycore, which in turn renders our optimization more important. For example, compared to the default input size of 663.3 MB, when the input size of *Barnes-Hut* is increased to 1989.9 MB (3x), its performance improvement increases from 18.4% to 25%. Similarly, when we

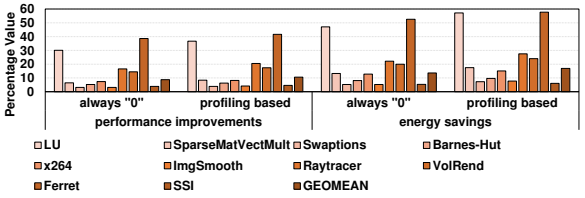


Fig. 19. Comparison of two different ways of supplying the value of a dropped data access.

increase the input size of *ImgSmooth* 4 times, the performance improvement moves from 15% to 25.5%.

Recall that our approach automatically identifies the set of "most beneficial" output data elements (accesses) to approximate (and costliest accesses to drop), given a user-specified error bound and a target multithreaded application code. We now quantify the behavior of a "simpler" version of our slicing based approach that selects the output elements whose values to be approximated *randomly* - instead of picking up the most beneficial ones. That is, instead of ranking the output elements from the most beneficial to approximate to the least beneficial and selecting the k most beneficial one, this new version just selects k output elements randomly. The results in Figure 18 indicate that the selecting the output elements (accesses) that return the highest benefits is critical.

Our next set of experiments investigate what happens if we could supply a value other than 0 for the data accesses dropped. While one can potentially adopt different strategies for that, in this work, we employ a "profiling" based strategy. More specifically, we first profile the application program being optimized (using different inputs where available) and identify, for each data access, the most frequently-occurring value. After that, during the execution, if we decide to drop a data access, we use its profile value. The performance and energy saving results with this alternate strategy are plotted in Figure 19 clearly show that, this enhanced strategy brings additional improvements over our default strategy of supplying always "0" for a dropped data access. In other words, there is some scope for further optimization over our baseline slicing-based strategy. One can certainly approach to (or even exceed) the results achieved by this profiling based strategy using "value prediction" [22]. In fact, we also implemented and tested another version of our approach that employs value prediction (based on the implementation given in [22]) to supply the dropped value. Our experiments indicated that the value prediction-based version performs worse than the profiling based version; so, we do not report here the results with the value prediction-based version. We also need to mention that there is a significant difference between our approximate computing strategy (even when it employs value prediction for supplying the skipped values) and the conventional value prediction-based performance optimization (which does not employ approximation at all). In the latter, the value is predicted and supplied to the execution but in the background the load is performed anyway, and if the predicted value and the actual value are the same, the processor simply continues, but if they are different, the processor rolls back the execution and *recomputes* the relevant computations with the actual value. In contrast, in our approximate computing-based approach, once the value is predicted, the execution continues with it, and it is not checked at all whether the prediction was correct.

6.4 Correlated Drops

So far in our discussion, we skipped data accesses solely based on the performance benefits they can potentially bring, in an isolated fashion. In reality, data accesses can be related to one another in different ways. For example, data accesses on the right hand side of a given assignment statement are related in the sense that, if one of them is dropped, others could be dropped as well, as doing so is unlikely to further worsen the accuracy (compared to the case where only one data access is

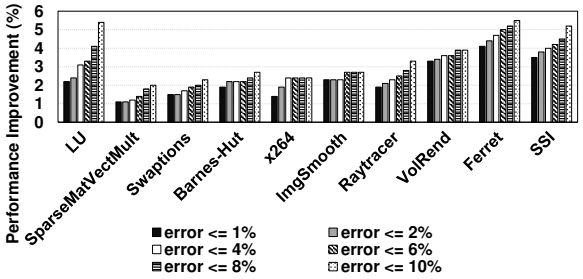


Fig. 20. Performance improvements when exploiting correlated data accesses.

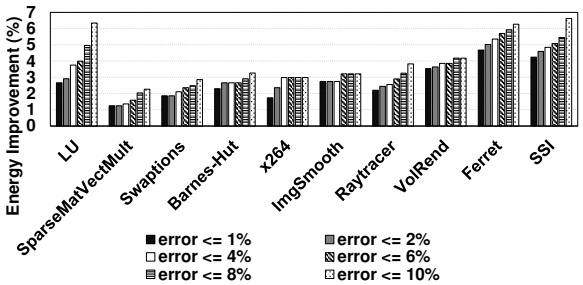


Fig. 21. Energy improvements when exploiting correlated data accesses.

dropped). For example, in $u = v + w + t;$, if v is dropped, going further and dropping w and t as well would probably not make too much difference from an accuracy viewpoint. Similarly, in a multi-statement scenario, $\{u = v + w; \cdots ; t = u + s;\}$, if v is dropped, s could be dropped as well without worsening accuracy any further (as u would be distorted anyway).

Motivated by these observations, we also performed experiments with a modified version of our slicing based approach to approximate computing. More specifically, under a given error bound, we exploited the correlation between different data accesses (as explained above) by dropping even more data accesses (in addition to those determined by our slicing-based approach), without worsening the accuracy beyond what is observed with the slicing-based scheme. We give the *additional* performance and energy improvements this new slicing-based version of our approach brings, *over our original slicing-based approach*, in Figures 20 and 21. Each bar in these two plots represent the geometric mean value across all applications we evaluated. One can see from the results presented in Figures 20 and 21 that, taking advantage of the correlations among different data accesses brings performance improvements ranging between 1.1% and 5.5%, and energy savings ranging between 1.8% and 6.7% (depending on the error bound), both over our original slicing-based scheme.

6.5 Results on Intel Manycore

Note that, it is not in general possible to drop individual data accesses in current commercial manycore systems. This is mainly because there is not a direct architectural/microarchitectural support for that. However, to have an idea about the potential of our approach in a commercial setting, we implemented and tested a "restricted version" of our approach on an Intel manycore system (the technical details of this system can be found elsewhere [40]). In this restricted version, once we identify the data accesses to be dropped, we map them to the static load instructions in the code. For example, a given data element, say A[5][3], can be accessed by 3 different static load

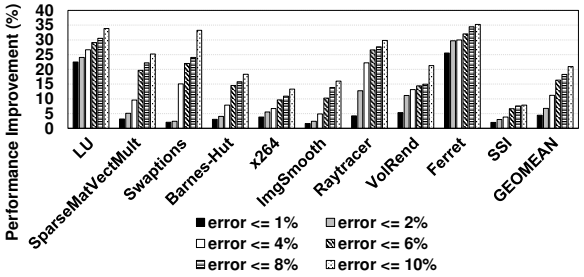


Fig. 22. Performance improvements on Intel KNL.

instructions in the code. If (i) such a load instruction does not access any other data element, or (ii) all the other data elements it accesses are also marked "to be dropped" by our slicing-based compiler analysis, we statically (at the assembler level) replace this load instruction such that it becomes a load-immediate instruction that loads "0" (provided that doing so does not cause the application to exceed the specified error bound). Clearly, this approach is not as strong as our original approach as the latter can drop accesses to a data element even if the load that accesses it also accesses other (potentially non-dropped) data elements. The performance results collected on Intel Xeon Phi are presented in Figure 22, under different error bounds. It can be observed 6.8% and 11.2% average performance improvements with error bounds of 2% and 4%, respectively. These results indicate that our approach is effective for current manycore systems as well.

7 RELATED WORK

We now discuss the prior work related to this paper in three categories: (i) hardware-based approaches, (ii) OS-based approaches, and (iii) compiler-based approaches.

Hardware Support: Hedge et al. explored low-energy digital signal processing as well as algorithmic noise tolerance schemes [12]. Venkataramani et al. developed Quara, a quality-programmable vector processor where the ISA provides accuracy levels for instructions [51]. Mohapatra et al. investigated voltage over-scaling (VOD) and proposed two techniques, dynamic segmentation and delay budgeting, to improve the scalability of VOD [26]. MACACO has been proposed by Venkatesan et al. for analyzing the behavior of an approximate circuit and measuring its error metrics such as worst-case error, average-case error, error probability, and error distribution [52]. Chippa et al. proposed a three-step framework to reduce the energy usage that includes characterizing the application for approximation, creating a unique hardware with scaling mechanisms and using a feedback-control technique at runtime to modulate dynamic scaling [5]. Nepal et al. created ABACUS, a tool for generating approximate circuits from arbitrary designs [27]. Kim et al. proposed an approximate adder that incorporates a parallel carry prediction strategy and an error correction logic to enhance machine learning (ML) applications [16], and Miguel et al. explored load value approximation for learning value patterns at a micro architectural level [23]. Esmaeilzadeh et al. proposed an ISA extension to support approximate operations and storage [7]. Compared to these prior studies, our slicing-based strategy is much less intrusive, as the only hardware support we need is to enable skipping of the data accesses marked by the slicer.

OS Support: Grigorian et al. proposed light-weight checks for enabling dynamic error analysis [10]. Sui et al. developed an ML method to learn the cost and error models of an approximation problem and tune the underlying approximation setup accordingly [43]. Renganarayana et al. proposed a relaxed synchronization method for reducing the overheads of parallel applications [31]. St. Amant et al. introduced a unified concept that exposes the analog circuits to the compiler to ensure that the approximations by the hardware can be tolerated by the software efficiently [41]. The ApproxIt

tool by Zhang et al. was developed to guarantee a certain level of quality with its approximations [56]. Li et al. explored the error resiliency of soft computations via error injections and proposed a recovery method based on checkpoints [21].

Compiler Support: Paraprox and SAGE by Samadi et al. have been developed for enhancing the approximate computing capabilities of GPUs [34, 35]. Esmaeilzadeh et al. investigated a neural network-based approach for accelerating general-purpose programs [8]. Vassiliadis et al. presented a mathematical model for determining the significance of code segments and used this model for determining beneficial regions to approximate [49]. In comparison, Mishra et al. developed iACT toolkit which provides an approximate memorization framework as well as a runtime and a simulated hardware test bed [25]. Sampson et al. introduced approximate data types to Java with their extension EnerJ [36], and Carbin et al. proposed a code classification technique for identifying critical regions [4]. Our work is different from these studies as we employ program slicing to identify the set of data accesses to drop.

8 CONCLUDING REMARKS

This paper makes two main contributions. First, it investigates the potential benefits of a form of approximate computing that drops/skips select data accesses in the executions of multithreaded workloads on merging manycore systems. The unique aspect of this evaluation is its "architecture awareness". That is, given a bound on program output error (inaccuracy), we quantify the benefits of dropping the costliest data accesses (in the target architecture), as opposed to dropping data accesses randomly. Our experiments with ten different multithreaded workloads indicate that being architecture aware in dropping data accesses pays off, resulting in 27% additional performance improvement (on average) over randomly dropping the same number of data accesses. Second, it presents a program slicing-based approach that identifies the set of data accesses to drop such that (i) the resulting performance/energy benefits are maximized and (ii) the execution remains within the error (inaccuracy) bound specified by the user. Our experiments with this approach reveal 8.8% performance improvement and 13.7% energy saving (on average) are possible when we set the error bound to 2%, and these improvements increase to 15% and 25%, respectively, when the error bound is raised to 4%. Our ongoing work includes extending the proposed approach to nonvolatile memory (NVM) based systems.

ACKNOWLEDGMENT

We thank Murali Annavaram for shepherding our paper. We also thank the anonymous reviewers for their constructive feedback. This research is supported in part by NSF grants #1526750, #1763681, #1439057, #1439021, #1629129, #1409095, #1626251, #1629915, and a grant from Intel.

REFERENCES

[1] Akturk, I., Khatamifard, K., and Karpuzcu, U. R. On quantification of accuracy loss in approximate computing.

[2] Bienia, C., Kumar, S., Singh, J. P., and Li, K. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *PACT* (2008).

[3] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sen, R., Sewell, K., Shoaib, M., Vaish, N., Hill, M. D., and Wood, D. A. The gem5 simulator. *ACM SIGARCH Computer Architecture News* (2011).

[4] Carbin, M., and Rinard, M. C. Automatically Identifying Critical Input Regions and Code in Applications. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*.

[5] Chippa, V. K., Venkataramani, S., Chakradhar, S. T., Roy, K., and Raghunathan, A. Approximate Computing: An Integrated Hardware Approach. In *Asilomar Conference on Signals, Systems and Computers* (2013).

[6] Ding, W., Tang, X., Kandemir, M., Zhang, Y., and Kultursay, E. Optimizing Off-chip Accesses in Multicores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2015).

[7] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Architecture support for disciplined approximate programming. In *ASPLOS* (2012).

[8] Esmaeilzadeh, H., Sampson, A., Ceze, L., and Burger, D. Neural acceleration for general-purpose approximate programs. In *MICRO* (2012).

[9] Ghosh, S., Martonosi, M., and Malik, S. Cache miss equations: a compiler framework for analyzing and tuning memory behavior. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1999).

[10] Grigorian, B., and Reinman, G. Improving coverage and reliability in approximate computing using application-specific, light-weight checks. *First Workshop on Approximate Computing Across the System Stack (WACAS)* (2014).

[11] Han, J., and Orshansky, M. Approximate computing: An emerging paradigm for energy-efficient design. In *18th IEEE European Test Symposium (ETS)* (2013).

[12] Hegde, R., and Shanbhag, N. R. Energy-efficient signal processing via algorithmic noise-tolerance. In *Proceedings of the International Symposium on Low Power Electronics and Design* (1999).

[13] Huh, J., Kim, C., Shafi, H., Zhang, L., Burger, D., and Keckler, S. A NUCA substrate for flexible CMP cache sharing. *IEEE Transactions on Parallel and Distributed Systems* (2007).

[14] Kandemir, M., Zhao, H., Tang, X., and Karakoy, M. Memory Row Reuse Distance and Its Role in Optimizing Application Performance. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)* (2015).

[15] Kayiran, O., Jog, A., Pattnaik, A., Ausavarungnirun, R., Tang, X., Kandemir, M. T., Loh, G. H., Mutlu, O., and Das, C. R. uC-States: Fine-grained GPU Datapath Power Management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)* (2016).

[16] Kim, Y., Zhang, Y., and Li, P. An energy efficient approximate adder with carry skip for error resilient neuromorphic VLSI systems. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (2013).

[17] Kislal, O., Kotra, J., Tang, X., Kandemir, M. T., and Jung, M. Enhancing computation-to-core assignment with physical location information. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (2018).

[18] Korel, B., and Laski, J. Dynamic program slicing. *Information Processing Letters* (1988).

[19] Lattner, C., and Adve, V. LLVM: A compilation framework for lifelong program analysis and transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO)* (2004).

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

[21] Li, X., and Yeung, D. Exploiting soft computing for increased fault tolerance. In *In Proceedings of Workshop on Architectural Support for Gigascale Integration* (2006).

[22] Lipasti, M. H., Wilkerson, C. B., and Shen, J. P. Value locality and load value prediction. In *ASPLOS* (1996).

[23] Miguel, J. S., Badr, M., and Jerger, N. E. Load value approximation. In *MICRO* (2014).

[24] Misailovic, S., Carbin, M., Achour, S., Qi, Z., and Rinard, M. C. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications* (2014).

[25] Mishra, A. K., Barik, R., and Paul, S. iact: A software-hardware framework for understanding the scope of approximate computing. In *Workshop on Approximate Computing Across the System Stack (WACAS)* (2014).

[26] Mohapatra, D., Chippa, V. K., Raghunathan, A., and Roy, K. Design of voltage-scalable meta-functions for approximate computing. In *Design, Automation Test in Europe* (2011).

[27] Nepal, K., Li, Y., Bahar, R. I., and Reda, S. Abacus: A technique for automated behavioral synthesis of approximate computing circuits. In *Proceedings of the Conference on Design, Automation and Test in Europe* (2014).

[28] Pattnaik, A., Tang, X., Jog, A., Kayiran, O., Mishra, A. K., Kandemir, M. T., Mutlu, O., and Das, C. R. Scheduling Techniques for GPU Architectures with Processing-In-Memory Capabilities. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation (PACT)* (2016).

[29] Pattnaik, A., Tang, X., Kayiran, O., Jog, A., Mishra, A., T. Kandemir, M., Sivasubramaniam, A., and Das, C. R. Opportunistic Computing in GPU Architectures. In *Proceedings of the 46th International Symposium on Computer Architecture* (2019).

[30] Pugh, W., and Rosser, E. Iteration space slicing and its application to communication optimization. In *Proceedings of the 11th International Conference on Supercomputing* (1997).

[31] Renganarayana, L., Srinivasan, V., Nair, R., and Prener, D. Programming with relaxed synchronization. In *Proceedings of the ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability* (2012).

[32] Rinard, M. C. Unsynchronized techniques for approximate parallel computing.

[33] Rubio-GonzÃąlez, C., Nguyen, C., Nguyen, H. D., Demmel, J., Kahan, W., Sen, K., Bailey, D. H., Iancu, C., and Hough, D. Precimonious: Tuning assistant for floating-point precision. In *2013 SC - International Conference for High Performance Computing, Networking, Storage and Analysis (SC)* (2013).

[34] Samadi, M., Jamshidi, D. A., Lee, J., and Mahlke, S. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS* (2014).
[35] Samadi, M., Lee, J., Jamshidi, D. A., Hormati, A., and Mahlke, S. Sage: Self-tuning approximation for graphics engines. In *MICRO* (2013).
[36] Sampson, A., Dietl, W., Fortuna, E., Gnanapragasam, D., Ceze, L., and Grossman, D. Enerj: Approximate data types for safe and general low-power computation. In *PLDI* (2011).
[37] Shrifi, A., Ding, W., Guttman, D., Zhao, H., Tang, X., Kandemir, M., and Das, C. DEMM: a Dynamic Energy-saving mechanism for Multicore Memories. In *Proceedings of the 25th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2017).
[38] Sidiroglou, S., Misailovic, S., and Hoffmann, H. Managing performance vs. accuracy trade-offs with loop perforation. In *Proc. ACM SIGSOFT symposium* (2011).
[39] Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. Managing Performance vs. Accuracy Trade-offs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011).
[40] Sodani, A., Gramunt, R., Corbal, J., Kim, H.-S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y.-C. Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro* (2016).
[41] St. Amant, R., Yazdanbakhsh, A., Park, J., Thwaites, B., Esmaeilzadeh, H., Hassibi, A., Ceze, L., and Burger, D. General-purpose code acceleration with limited-precision analog computation. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture* (2014).
[42] Stanley-Marbell, P., and Rinard, M. Efficiency limits for value-deviation-bounded approximate communication. *IEEE Embedded Systems Letters* (2015).
[43] Sui, X., Lenharth, A., Fussell, D. S., and Pingali, K. Proactive control of approximate programs. In *ASPLOS* (2016).
[44] Tang, X., Kandemir, M., Yedlapalli, P., and Kotra, J. Improving Bank-Level Parallelism for Irregular Applications. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2016).
[45] Tang, X., Kislal, O., Kandemir, M., and Karakoy, M. Data movement aware computation partitioning. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (2017).
[46] Tang, X., Pattnaik, A., Jiang, H., Kayiran, O., Jog, A., Pai, S., Ibrahim, M., Kandemir, M., and Das, C. Controlled Kernel Launch for Dynamic Parallelism in GPUs. In *Proceedings of the 23rd International Symposium on High-Performance Computer Architecture (HPCA)* (2017).
[47] Tang, X., Taylan Kandemir, M., Karakoy, M., and Arunachalam, M. Co-Optimizing Memory-Level Parallelism and Cache-Level Parallelism. In *Proceedings of the 40th annual ACM SIGPLAN conference on Programming Language Design and Implementation* (2019).
[48] Tip, F. A survey of program slicing techniques. *Journal of programming languages* (1995).
[49] Vassiliadis, V., Riehme, J., Deussen, J., Parasyris, K., Antonopoulos, C. D., Bellas, N., Lalis, S., and Naumann, U. Towards automatic significance analysis for approximate computing. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (2016).
[50] Venkataramani, S., Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Quality programmable vector processors for approximate computing. In *Proc. of International Symposium on Microarchitecture (MICRO)* (2013).
[51] Venkataramani, S., Chippa, V. K., Chakradhar, S. T., Roy, K., and Raghunathan, A. Quality programmable vector processors for approximate computing. In *MICRO* (2013).
[52] Venkatesan, R., Agarwal, A., Roy, K., and Raghunathan, A. Macaco: Modeling and analysis of circuits for approximate computing. In *Proceedings of the International Conference on Computer-Aided Design* (2011).
[53] Weiser, M. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering* (1981).
[54] Woo, S. C., Ohara, M., Torrie, E., Singh, J. P., and Gupta, A. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News* (1995).
[55] Wulf, W. A., and McKee, S. A. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News* (1995).
[56] Zhang, Q., Yuan, F., Ye, R., and Xu, Q. Approxit: An approximate computing framework for iterative methods. In *The ACM/EDAC/IEEE Design Automation Conference* (2014).

Received February 2019; revised March 2019; accepted April 2019