Toward Dynamic Precision Scaling

Serif Yesil
University of Illinois at Urbana-Champaign

Ismail Akturk
University of Missouri-Columbia

Ulya R. Karpuzcu
University of Minnesota, Twin Cities

This article makes the case for dynamic precision scaling to improve power efficiency by tailoring arithmetic precision adaptively to temporal changes in algorithmic noise tolerance throughout the execution.

Approximate computing is a promising paradigm to enhance power efficiency by trading computation accuracy for performance or power. The intrinsic noise tolerance of the emerging recognition, mining, and synthesis (RMS) applications—which process massive but noisy input data by probabilistic algorithms—makes them particularly suitable to approximate computing. In this article, we make the case for tailoring the degree of approximation to changes in the application's noise tolerance within the course of execution. This is a generic paradigm that can be adapted to many approximation techniques, but we will use precision scaling as a case study. Approximate computing by (non-adaptive) precision reduction represents a well-explored area.[1,2] This doesn't apply to our focus: adaptive precision reduction. Specifically, based on the observation that applications exhibit varying degrees of sensitivity to noise during computation, we explore how tailoring the precision of computation adaptively to temporal changes in algorithmic noise tolerance can improve power efficiency.

Due to its analogy to dynamic voltage and frequency scaling (DVFS), we refer to this paradigm as dynamic precision scaling (DPS). Recall that DVFS can improve power efficiency by tracking temporal changes in the performance demand of the workload and by changing the operating voltage and frequency accordingly. Similar to DVFS, DPS tracks temporal changes in workload characteristics. However, in improving power efficiency, DPS rather tracks temporal changes in noise tolerance and adaptively decreases the arithmetic precision of noise-tolerant phases to obtain power savings at the same operating speed (or faster execution within the same power budget) while keeping the overall loss in accuracy due to precision reduction bounded. In this article, we conduct a limit study to quantify the power efficiency potential of DPS, by devising a proof-of-concept implementation.

DYNAMIC PRECISION SCALING

We envision a practical DPS implementation to comprise three basic modules: (1) an offline profiler to identify and demarcate application phases of different noise tolerance characteristics, (2) a runtime monitor to track temporal changes in workloads' noise tolerance, and (3) an accuracy

Published by the IEEE Computer Society
0272-1732/18/$33.00 ©2018 IEEE

controller to adjust the arithmetic precision "on the fly" accordingly. The differences in the design of these three modules give rise to different points in the DPS design space.

The offline profiler and runtime monitor should be able to capture fine-grain temporal changes in applications' noise tolerance. Because the noise tolerance of RMS applications is mainly algorithmic, software intervention is inevitable—for example, in the form of code annotations to demarcate varying degrees of noise tolerance. To communicate such annotations to the hardware, programming language extensions[2] may help. At the same time, noise tolerance is input-dependent, rendering profiling-based approaches such as ours necessary. The ideal solution may be hidden in yet-to-be-explored correlations between hardware-observable features (similar to performance counter outcome) and noise tolerance at the application level.

The accuracy controller design space spans software, hardware, or hybrid implementations, similar to DVFS controllers. For example, if the processor features functional units of reduced precision, the controller is in charge of scheduling (more) noise-tolerant phases to lower-precision arithmetic units. The processor may also accommodate functional units of reconfigurable precision to harvest power efficiency under DPS. In each case, a very stringent budget applies for the power and performance overhead of the accuracy controller.

The next section details a proof-of-concept DPS implementation that features an offline profiler along with a hypothetical runtime monitor and an accuracy controller.

Proof-of-Concept Implementation

Without loss of generality, we confine the proof-of concept DPS implementation to the floating-point data path. However, the DPS concept generally applies to the integer data path, as well, where the main complication comes from identification and, hence, exclusion of memory address calculations (i.e., pointer arithmetic) from approximation.

According to the IEEE 754 standard, a single (double) precision floating-point number occupies an 32(64)-bit register with one bit allocated for sign, 8(11) bits for exponent, and 23(52) bits for fraction (mantissa), respectively. A single precision floating-point number corresponds to $(-1)^{sign} \times 2^{exponent-127} \times 1.mantissa$. In the proof-of concept implementation, we reduce precision by omitting a subset of less significant bits of the mantissa.

The proof-of-concept implementation captures temporal changes in noise tolerance by tracking user-defined (and annotated) regions with a large number of floating-point operations. Each static floating-point-heavy region may have multiple dynamic instances, similar to static function definitions and dynamic function calls. Without loss of generality, these floating-point-heavy regions can take different forms, such as loop iterations or functions. In any case, the dynamic instances are dispersed in time, and each may feature a different degree of noise tolerance. We capture temporal changes in noise tolerance by tracking application behavior at region-granularity, by tracking changes in the noise tolerance of such dynamic instances.

The question now becomes how to identify the noise tolerance of these regions. To this end, the offline profiler module in the proof-of-concept implementation uses a two-step approach: The first step involves statistical fault injection; the second step involves post-processing of the resulting execution outcome. In the first step, for all floating-point operation destinations (register or memory values) in each dynamic instance of a region, we corrupt one mantissa bit at a time and record the corresponding accuracy loss at the application output. We repeat this experiment for all mantissa bits, by injecting both stuck-at-0 and stuck-at-1 faults. The proof-of-concept implementation uses noise-tolerant dynamic instances of regions as proxies for noise-tolerant phases of the application. Therefore, the accuracy loss observed in the end result per fault injection experiment serves as a proxy for the degree of noise tolerance of each dynamic instance of a region.

The post-processing at Step 2 can rely on different policies. We first devise a basic policy (DPS) following Algorithm 1; the key inputs of this algorithm are targetAccLoss (the maximum accuracy loss in the end result the application can tolerate) and the outcome of the first step of offline profiling (namely, the accuracy loss observed in the end result after injecting stuck-at-0 and

stuck-at-1 faults in each mantissa bit of each dynamic instance of a region). #bits specifies the number of mantissa bits subject to fault injection, and #regions, the number of dynamic instances of regions (which may correspond to multiple executions of the same region, different regions, or a mixture of both).

```
Input: targetAccLoss, #bits, #regions
Input: AccLossS0[1...#regions][1..#bits]
Input: AccLossS1[1...#regions][1..#bits]
Output: #omittedBits[1...#regions]

1. for i←1...(#regions-1) do
2.      targetBit←0;
3.      cummAccLoss←0;
4.      cummAccLoss_next←0;
5.      while cummAccLoss<targetAccLoss & targetBit≤#bits
           & cummAccLoss_next<targetAccLoss do
6.              if AccLossS0[i][targetBit], AccLossS1[i][targetBit]
                   AccLossS0[i+1][targetBit], AccLossS1[i+1][targetBit]
                    are valid then
7.                      err←max(AccLossS0[i][targetBit],AccLossS1[i][targetBit]);
8.                      err_next ← max(AccLossS0[i+1][targetBit],
                           AccLossS1[i+1][targetBit]);
9.                      cummAccLoss+←err ;
10.                     cummAccLoss next+←err_next;
11.             else
12.                     break;
13.             end
14.             targetBit++;
15.     end
16.     #omittedBits[i]←targetBit-1;
17. end
```

Algorithm 1. Basic DPS and dependency-aware DPS+ policies, with the latter highlighted.

We keep the fault injection information in two separate (#regions x #bits) matrices for stuck-at-0 (AccLossS0) and stuck-at-1 faults (AccLossS1). The algorithm output is the total number of (consecutive) mantissa bits (starting from the least significant) we can omit while the corresponding accuracy loss in the end result remains lower than targetAccLoss, on a per-region basis: #omittedBits.

Each step of the algorithm processes a different dynamic instance of a region (Line 1). Starting from the least significant bit, we check the accuracy loss in the end result under the corruption of each mantissa bit (targetBit): If the corresponding AccLossS0(1) entries are valid—meaning the fault injection experiment did not result in Inf or NaN (Line 5)—we extract the maximum of accuracy loss under stuck-at-0 and stuck-at-1 (Line 7). The basic DPS policy accumulates this maximum accuracy loss in the end result due to the corruption of each mantissa bit in isolation (in cummAccLoss from Line 9), as we consider more mantissa bits for omission. cummAccLoss serves as a running estimate for the actual accuracy loss in the end result. Accordingly, the policy keeps evaluating higher-order mantissa bits for omission as long as cummAccLoss remains below targetAccLoss (Line 5). We can then use the output of the basic DPS policy captured by Algorithm 1, #omittedBits (over all regions, in other words, phases), to tune the precision of the workload "on the fly."

Algorithm 1's main bottleneck is cummAccLoss, the estimate of cumulative accuracy loss in the end result of the application if we omit multiple mantissa bits, on a per-region basis (Line 9). This is because the actual (runtime) impact of each omitted mantissa bit on the accuracy loss in the end result may not always be additive. Therefore, in the worst case, if we omit multiple mantissa bits following Algorithm 1—as captured by #omittedBits—we may eventually observe a higher accuracy loss in the end result than targetAccLoss. A refined version of the basic DPS

policy—DPS+—can address this, as depicted in Algorithm 1, with the difference from DPS highlighted.

Both algorithms process all dynamic region instances within the course of execution; the order of the executed regions in AccLossS0(1) and #omittedBits arrays reflect their execution order in the offline profiling run. For the representative set of RMS benchmarks we experiment with later in the article, we observe that dynamic region instances, following each other in dynamic control flow, are often also data-dependent. DPS+ leverages this insight by limiting the precision reduction of a dynamic region instance to the precision reduction of its following region instance in execution (and processing) order. In this manner, we enforce that the (reduced) precision of a producer's output data matches (does not exceed) the maximum acceptable precision of the input data of its (immediate) consumer. DPS+ still cannot provide mathematical guarantees (as data-dependent regions are not always executed back to back), but can effectively enforce runtime accuracy loss (in the end result) to remain below targetAccLoss.

EVALUATION SETUP

Throughout the evaluation, we refer to the proof of-concept implementation as DPS+.

Benchmarks

To quantify the power efficiency potential of DPS, we deploy a representative, relatively floating-point-heavy set of RMS applications from PARSEC, Rodinia, and Gapbs suites: Blackscholes (BS), a partial differential equation (PDE) solver for stock options (while the application domain is arguably not suitable for approximation, we keep BS as a PDE example); Fluidanimate (FA), an n-body simulator; Hotspot (HS), a numerical thermal simulator; Particlefilter (PF), a medical imaging application; and PageRank (PR), an iterative graph algorithm implementation. For each benchmark, we identify floating-point heavy regions (which represent functions for the majority) in the application code, where the actual computation takes place.

All benchmarks output (possibly multi-dimensional) numeric values. To quantify the accuracy loss in the end result, we use mean relative error (the average relative error over all data points in the output) with respect to the full-precision outcome. The data points in BS are final stock option prices; in FA, final positions of bodies; in HS, temperature values; in PF, the final position of the object being tracked; and in PR, the PageRank values.

Simulation Infrastructure

We implement the proof-of-concept offline profiler (compromising statistical fault injection) and DPS policies as an extension to the Pin-based approximate computing framework iACT.[4] During offline profiling, we inject two types of faults in the mantissa: We set one mantissa bit (out of 23 for single; 52, for double precision) to 0 (stuck-at-0) or 1 (stuck-at-1) at a time. Our tool instruments all floating-point arithmetic and load/store instructions for DPS+. In accordance with the energy model presented in Shao *et al.*,[5] we experimented with a Xeon-Phi-like system of 32-Kbyte (512-Kbyte) L1 (L2) data cache.

Energy Model

To model energy, we use the sum of products of energy per instruction (EPI) and number of instructions (#instructions), over each instruction category in dynamic region instances. EPI estimates come from measured data from Shao *et al.*,[5] which categorize instructions according to the sources of operands as register file (RF), L1, L2, and memory. Accordingly, to calculate the energy consumption of each dynamic region instance, we determine the number of instructions in each (operand source-based) category following the classification in Shao *et al.*[5] We deploy the scaling model from Tong *et al.*[6] to model how energy consumption of individual operations changes as a function of the number of mantissa bits omitted. Tong *et al.*[6] shows that in floating-

point multiplication, which represents one of the most energy-hungry floating-point operations, processing mantissa bits can consume more than 80 percent of the total energy.

EVALUATION

Application Characteristics

Fine-grain temporal changes in the noise tolerance of RMS applications motivates DPS+. Figure 1 verifies this insight, where we plot the outcome of the fault injection experiments to measure the sensitivity of each floating-point-heavy region to noise. The x-axis depicts which mantissa bit we corrupt. The y-axis shows the dynamic region instances, numbered in the order of execution. Therefore, the y-axis represents the time. This figure captures the (relative) accuracy loss in the end result, as induced by the corrupted bit, for each dynamic region instance, in gray scale: Black indicates a totally inaccurate result with a relative accuracy loss of 1; white indicates no accuracy loss. As expected, we observe darker regions (less noise tolerance) as we move right on the x-axis (as we corrupt more significant mantissa bits).

We further observe that the accuracy loss indeed changes with time (across the y-axis) for all applications, least pronounced for HS in Figure 1(c). For the rest of the applications (most clearly for FA and PF), the accuracy loss (as a proxy of noise tolerance) shows recurring patterns over time. PR (Figure 1(e)), on the other hand, exhibits less noise tolerance as we move up on the y-axis (in later stages of execution). This is because, relying on iterative refinement, PR has less opportunities to recover from noise in later stages of execution.

For the benchmarks where each region corresponds to a function, differences in the noise tolerance between different dynamic instances of the very same (static) region stem from differences in the function inputs across calls. For instance, BS has a single computational kernel. Each (dynamic) call to this kernel processes different inputs. Figure 1(a) reveals the differences in accuracy loss among these calls due to inputs along the time (y-) axis. Input data values also have an impact. When working with smaller values, corruptions in less significant bits can also induce noticeable accuracy loss in the end result. This effect is most visible in Figure 1(a). Putting it all together, our observations so far all point to various opportunities for DPS.

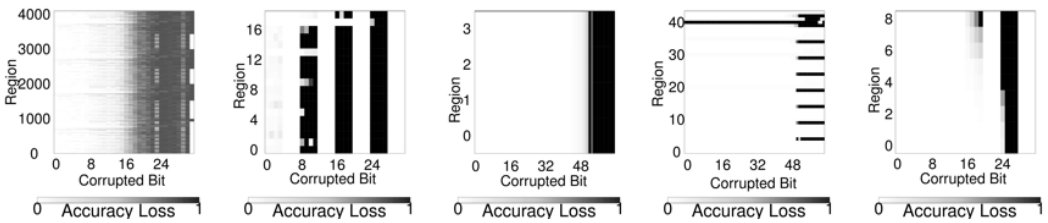

Figure 1. Statistical fault injection outcome to extract temporal noise tolerance.

Static Precision Scaling

As a baseline for comparison, we next look into the accuracy loss and energy consumption if we impose a fixed degree of precision reduction, statically, throughout the entire execution. We refer to this policy as static precision scaling (SPS). As SPS does not differentiate between noise tolerance of dynamic region instances, the least noise-tolerant instance dictates the final accuracy loss. Under SPS, we accordingly omitted (ceiling of) 5 to 75 percent of mantissa bits. Figure 2 captures the outcome. We observe that SPS can render sizable energy savings (Figure 2(b)), particularly as we omit more than 10 percent of the bits. However, for BS and FA, the energy savings are accompanied by an excessive accuracy loss, as revealed in Figure 2(a). According to Figure 1(b), FA can tolerate corruption in higher order bits of mantissa, but SPS cannot unlock this opportunity. Under SPS, FA renders unacceptable accuracy loss if we omit 50 percent of the

bits (12 bits). In the next section, we show that FA can tolerate omission of up to 22 bits under DPS+ (Figure 3(e) and 3(f)). Similarly, under SPS, BS cannot tolerate the omission of 50 percent of its mantissa bits (12 bits), where according to Figure 1(a), many of its regions may tolerate corruption at higher order bits (bit 18, for example). For the rest of the applications, SPS performs arguably well. Still, DPS+ can unlock more opportunities for power efficiency; as a specific example, PR under SPS with 75 percent of mantissa bits (18 bits) omitted results in acceptable accuracy loss. According to Figure 1(e), PR can temporally tolerate the omission of even higher order bits. In the next section, we show how DPS+ can unlock this opportunity by omitting 90 percent of the mantissa bits on average to render an accuracy loss of 0.02.

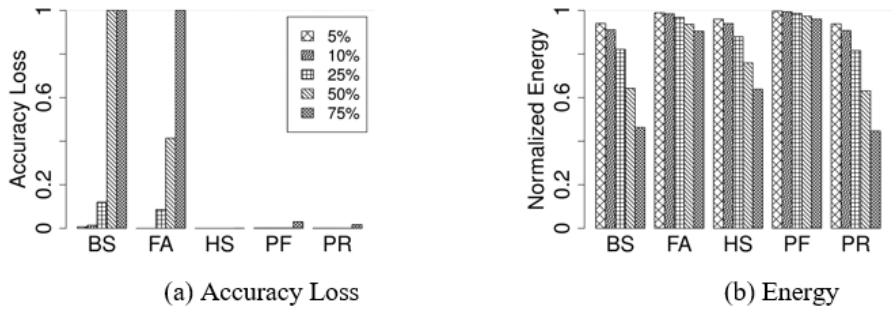

Figure 2. Impact of SPS on (a) accuracy loss and (b) energy.

Dynamic Precision Scaling

We next evaluate the effectiveness of DPS+. We invoke the two DPS algorithms with different values of targetAccLoss, the maximum accuracy loss in the end result the application can tolerate and ignore the resulting #omittedBits. Recall that #omittedBits gives the total number of (consecutive) mantissa bits (starting from the least significant) we can omit on a per-dynamic-region-instance basis, while the corresponding accuracy loss in the end result remains lower than targetAccLoss. Here, we report the outcome for targetAccLoss values between 0.05 and 0.2 at increments of 0.05. Figure 3 depicts the number of omitted bits for each dynamic execution of a region as a function of targetAccLoss, for FA, PR, and PF, respectively (under DPS in (a) and DPS+ in (b)).

The pattern under DPS closely tracks our findings in Figure 1, as DPS considers each region in isolation. This observation also holds for BS and HS, not shown as the corresponding figures were barely readable due to very fine-grain temporal fluctuations. Recall that the x-axis captures each dynamic region instance in the order of execution and, hence, represents a proxy for time. Figure 3 shows how the number of omitted bits changes over time to track the temporal changes in the noise tolerance of the applications (more noise-tolerant phases being able to accommodate a higher number of omitted mantissa bits).

At this point, let us introduce a new baseline for comparison: SPS+. SPS+ works as follows. During profiling, we use the DPS heuristic to find the number of bits to be omitted for each dynamic region instance. We then find the minimum, of the number of omitted bits, for each static region over all of its dynamic instances. SPS+ imposes this minimum number of bits on all dynamic instances of the respective static region throughout execution.

We next examine the corresponding accuracy loss in the end result of the applications in Table 1 for DPS, DPS+, and SPS+. As expected, a higher targetAccLoss renders monotonically higher accuracy loss for all applications. The accuracy loss under SPS+ is the minimum because it is the most conservative policy. Dependency tracking introduced by DPS+ yields a more accurate result compared to DPS.

One drawback of DPS+ is that if the consecutively executed (dynamic) region instances are not data-dependent, DPS+ may limit precision reduction unnecessarily. As a result, the end result

may become more accurate, leaving potential energy savings on the table. BS from Table 1 represents such an example, where each dynamic region instance processes independent inputs and generates independent outputs. As Table 1 reveals, SPS+ renders a lower accuracy loss. However, as SPS+ cannot exploit temporal changes in algorithmic noise tolerance, it leaves potential savings in energy untapped. For example, BS features a single static region (function), some dynamic instances of which cannot tolerate approximation (where #omittedBits becomes zero). In this case, SPS+ imposes this #omittedBits as the minimum over all dynamic calls and, hence, excludes any approximation. As a result, SPS+ cannot deliver any energy savings for BS as opposed to DPS+, as shown in Figure 4.

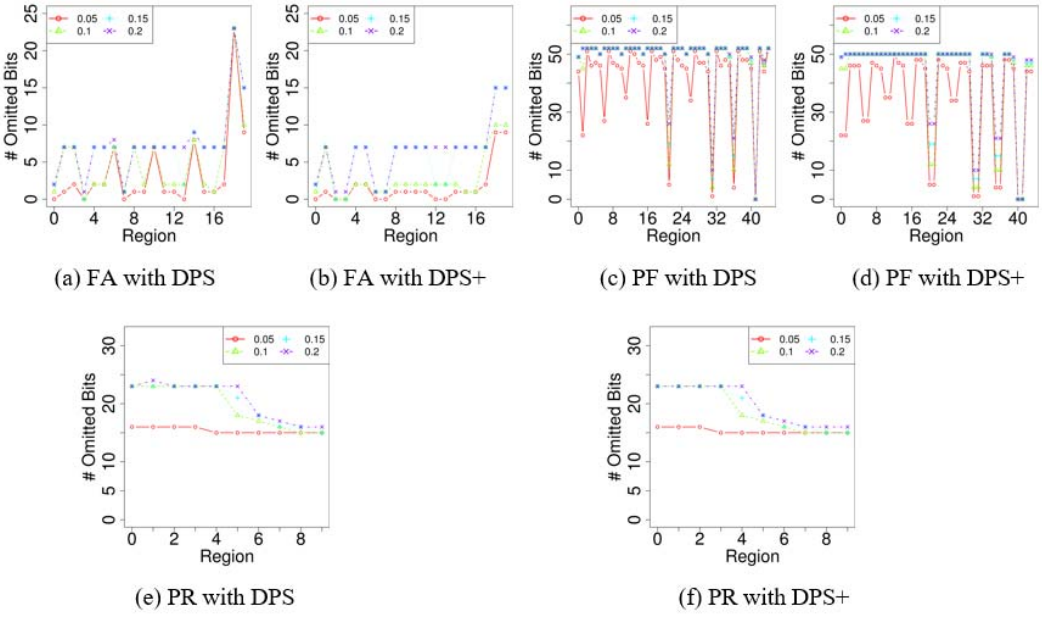

Figure 3. #omittedBits (Algorithm 1) under DPS+ for different values of targetAccLoss.

Figure 4(a) captures energy savings under DPS for targetAccLoss values between 0.05 and 0.2 at increments of 0.05. Savings span 5.4 to 57.9 percent for targetAccLoss = 0.2. Notable energy savings apply to BS, HS, and PR. Savings for FA and PF are modest. This is because the most energy-hungry dynamic region (function) instances feature the lowest number of omitted mantissa bits (see Figure 3(c) and 3(d)).

DPS+ renders fewer (or at most equal) numbers of omitted bits when compared to DPS and, hence, may leave potential energy savings untapped in trying to limit the accuracy loss. However, we find that the maximum difference between energy savings of DPS and DPS+ barely reaches 3 percent. On average, the difference remains around 0.93 percent.

Figure 4(b) captures the energy profile under SPS+. Overall, SPS+ renders a higher energy consumption than DPS+. BS is not the only application that loses the energy benefits of DPS under SPS+. Energy consumption of PR also increases, by 10 percent when compared to DPS. Overall, the increase in energy consumption varies between 0 and 40 percent across all applications.

Input Sensitivity

To quantify the input dependence due to profiling, we experiment with PR, which features a rich set of inputs. Similar to other profiling-based approaches, DPS+ is input-dependent. However, the degree of this dependence changes from application to application. At the same time, when the properties (such as size and value distribution) of two input datasets are close to each other, #omittedBits per dynamic region instance (or function call, if applicable), as identified by profiling under one dataset, may result in reasonable output when applied to another dataset. Table 2

quantifies this effect for PR. This table is the equivalent of Table 1, except that only the gnu04 input dataset is used for profiling. In other words, #omittedBits as determined by a profiling pass for gnu04 is imposed when running the same application with different input datasets (as tabulated in the first column of Table 2). The number of vertices in gnuXX graphs vary between 8,846 and 22,687, and the number of edges vary between 31,839 and 54,705. For these inputs, PR features a relatively weak input dependence. We experiment with one more graph: web-Google (wg) from the same database. This graph has 875,713 vertices and 5.1 million edges. As Table 2 captures, the discrepancy in this case becomes notable: 7x more accuracy loss under DPS, when compared to the gnu04 dataset.

Table 1. Accuracy loss under DPS+ and SPS+ for different targetAccLoss values.

Policy	App	targetAccLoss			
		0.05	**0.1**	**0.15**	**0.2**
DPS	BS	0.0101	0.0194	0.0278	0.0362
	FA	8.63E-05	0.0019	0.1053	0.1053
	HS	4.54E-05	5.84E-05	7.13E-05	7.23E-05
	PF	0.0003	0.0024	0.0025	0.0037
	PR	0.0033	0.0102	0.0135	0.0222
DPS+	BS	0.0083	0.016	0.0228	0.0304
	FA	4.69E-05	0.0001	0.1054	0.1054
	HS	4.54E-05	5.84E-05	7.11E-05	7.23E-05
	PF	0.0003	0.0003	0.0021	0.0209
	PR	0.0033	0.0063	0.0077	0.014
SPS+	BS	0	0	0	0
	FA	3.21E-08	3.21E-08	3.21E-08	1.62E-06
	HS	4.54E-05	5.84E-05	5.84E-05	7.23E-05
	PF	0.0009	0.0009	0.0009	0.0009
	PR	0.0033	0.0033	0.0033	0.0079

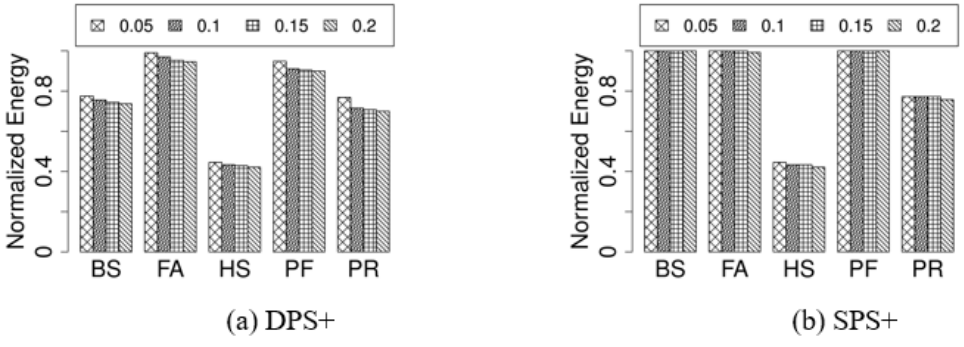

Figure 4. Energy consumption under DPS+ and SPS+ for different targetAccLoss values.

Table 2. Input sensitivity analysis for PageRank (PR) with different datasets from the Snap database. Only gnu04 is deployed for profiling.

Input	Policy	targetAccLoss			
		0.05	0.1	0.15	0.2
gnu04	DPS	0.0033	0.0102	0.0135	0.0222
	DPS+	0.0033	0.0063	0.0077	0.014
	SPS+	0.0033	0.0033	0.0033	0.0079
gnu05	DPS	0.0034	0.0073	0.0101	0.0158
	DPS+	0.0033	0.005	0.0061	0.0108
	SPS+	0.0033	0.0033	0.0033	0.0074
gnu25	DPS	0.0026	0.0043	0.0056	0.0102
	DPS+	0.0026	0.0032	0.0035	0.0072
	SPS+	0.0026	0.0026	0.0026	0.0058
wg	DPS	0.0121	0.137	0.1479	0.1695
	DPS+	0.0114	0.114	0.1226	0.1408
	SPS+	0.0097	0.0097	0.0097	0.0222

RELATED WORK

Adaptive precision reduction has been explored in the context of physics simulation to minimize the area cost of floating-point units[1] and for digital signal processing.[7] We consider a broader class of RMS applications. On the other hand, automated program analysis/tuning tools to help developers optimize floating-point precision[8-10] fit well into the offline profiling stage of DPS, but the existing body of work in this domain usually does not explore adaptive tuning. Identification of approximation targets at different granularities, as explored in Chisel[11] or Approxilyzer,[12] can be useful within the offline profiler or runtime monitor modules of a DPS system, as well. For precision tuning, a practical DPS implementation can also benefit from an instruction set that features explicit accuracy specification for each instruction's outcome.[3]

ACKNOWLEDGMENTS

This work was supported in part by NSF grant no. CCF-1438286.

REFERENCES

1. T. Y. Yeh et al., "The Art of Deception: Adaptive Precision Reduction for Area Efficient Physics Acceleration," *International Symposium on Microarchitecture*, 2007.
2. A. Sampson et al., "EnerJ: Approximate Data Types for Safe and General Low-power Computation," *Conference on Programming Language Design and Implementation* (PLDI), 2011.
3. S. Venkataramani et al., "Quality programmable vector processors for approximate computing," *International Symposium on Microarchitecture* (MICRO), 2013.

4. A. K. Mishra, R. Barik, and S. Paul, "iact: A software-hardware framework for understanding the scope of approximate computing," 2014; https://sampa.cs.washington.edu/wacas14/papers/mishra.pdf.
5. Y. Shao and D. Brooks, "Energy Characterization and Instruction-Level Energy Model of Intel's Xeon Phi Processor," *International Symposium on Low Power Electronics and Design* (ISLPED), 2013.
6. 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 Very Large-Scale Integration Systems*, vol. 8, no. 3, June 2000.
7. S. Lee and A. Gerstlauer, "Fine grain word length optimization for dynamic precision scaling in dsp systems," *International Conference on Very Large-Scale Integration*, 2013.
8. C. Rubio-Gonzalez et al., "Precimonious: Tuning assistant for floating-point precision," *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.
9. M. D. Linderman et al., "Towards Program Optimization Through Automated Analysis of Numerical Precision," *International Symposium on Code Generation and Optimization*, 2010.
10. T. Moreau et al., "Exploiting quality-energy tradeoffs with arbitrary quantization," *International Conference on Hardware/Software Codesign and System Synthesis*, 2017.
11. S. Misailovic et al., "Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels," *SIGPLAN Not.*, vol. 49, no. 10, October 2014.
12. R. Venkatagiri et al., "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," *International Symposium on Microarchitecture*, 2016.

ABOUT THE AUTHORS

Serif Yesil is a PhD student in the computer science department at the University of Illinois at Urbana-Champaign. He has a master's degree in computer engineering from Bilkent University. His research interests include computer architecture, heterogeneous architectures, and parallel computing. Contact him at syesil2@illinois.edu.

Ismail Akturk is an assistant professor in the Electrical Engineering and Computer Science Department at the University of Missouri-Columbia. He has a PhD from the Department of Electrical and Computer Engineering at the University of Minnesota, Twin Cities. His main research area is computer architecture, including energy efficiency, communication reduction in many-core systems, scalability, and soft computing. Contact him at akturki@missouri.edu.

Ulya Karpuzcu is an associate professor in the Department of Electrical and Computer Engineering at the University of Minnesota, Twin Cities. She has a PhD in computer engineering from the University of Illinois at Urbana-Champaign. Her research interests span the impact of technology on computing, energy-efficient computing, application domain specialized architectures, approximate computing, and computing at ultra-low voltages. Contact her at ukarpuzc@umn.edu.