iACT: A Software-Hardware Framework for Understanding the Scope of Approximate Computing

Asit K. Mishra Rajkishore Barik Somnath Paul
Intel Labs
{asit.k.mishra, rajkishore.barik, somnath.paul}@intel.com

Abstract

Approximate computing has recently emerged as a paradigm for enabling energy efficient software and hardware implementations by exploiting the inherent resiliency in applications to impreciseness in their underlying computations. Much of the previous work in this area has demonstrated the potential for significant energy and performance improvements, but these works largely consist of ad hoc techniques that are applied to a small number of similar applications. Mainstream adoption of approximate computing requires a deeper understanding of the inherent application resilience and the codesign hardware to go with the software. This dictates tools and methods that can help programmers reason about the scope and behavior of approximations in applications. To this end, this paper discusses an *open source* toolkit, called iACT (Intel's Approximate Computing Toolkit) to analyze and study the scope of approximations in applications. Our toolkit consists of a compiler, runtime and a simulated hardware test bed. We discuss the design of this toolkit and present examples of how to use this toolkit in this paper. As an example on how to use this toolkit, we include two different applications and analyze the scope of approximate computing in these.

1. Introduction

Recently, approximate computing has emerged as a new design approach that leverages inherent application resilience through hardware optimizations that trade off output quality for improved performance and energy efficiency. Inherent application resilience refers to the property of an application to produce acceptable outputs despite some of its underlying computations being executed incorrectly in the underlying hardware. Effectively, approximate computing optimizations relax the traditional requirement of exact (numerical or Boolean) equivalence between the specification and implementation. This effectively leads to trading off accuracy for better performance and energy efficiency and has turned out to be an attractive option for many important and resource-hungry applications, including machine learning, image and video processing, computer vision, probabilistic and statistical analytics, simulations, big data analytics, etc.

WACAS '14, March 2, 2014, Salt Lake City, Utah, USA.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

Why do applications exhibit inherent resiliency? The inherent resilience of the application classes mentioned above could be attributed to: (i) Significant redundancy is present in large, real-world data sets that these applications process and in many instances (such as in image and video processing applications) the inputs are drawn from the real world using sensors (such as camera), and anytime inputs are drawn from real world they always come with a share of noise which the applications know how to handle. (ii) Application classes mentioned above, produce outputs which are eventually *consumed* by humans and we, humans, have our own perceptual limitations. Further, in many of these applications, a range of outputs are equivalent (i.e., no unique golden output exists), or small deviations in the output cannot be perceived by users [9, 31].

Why study approximate computing now? The motivation for studying approximate computing and designing hardware to relax guarantees stems from two aspects: (i) Evolutionary: With transistor scaling becoming less and less effective at improving system energy efficiency and performance, we need avenues that provides a path forward for the computer industry. Along side, devices are becoming less and less reliable and designers are starting to operate the devices at the limits of their reliability. As such, designers are starting to seek alternative avenues in improving the die yield. In such a scenario, approximate computing could be a good candidate to improving the yield by hosting the resilient applications (or portions of applications that are resilient) on to the unreliable portions of the chip and the remaining portion of the application of the reliable portion of the chip. (ii) Many of the emerging and important applications can be approximate: As mentioned above, many of the emerging applications (targeted towards mobile and handheld computing, and data analytics) are amenable to this class of computation, but are very resource hungry. Approximate computing could help bring down the energy envelopes of these applications, make the hardware they run have longer battery life (even, smaller battery form factor) and improve the performance of these applications as well.

Several recent efforts have explored approximate computing both in software as well as hardware [1, 7, 19, 23, 29, 31, 33] with encouraging results. Software schemes proposed in these works improve performance by skipping computations or reducing the use of expensive operations such as inter-thread synchronization, whereas hardware techniques simplify or modify the design at various levels of abstraction to introduce tradeoffs between output quality and efficiency, such as operating modules at lower voltage or reducing the precision of computations. All these efforts have brought forth the significant potential of approximate computing, and there is a strong interest in its use with the growth in emerging applications that are amenable to this class of computations. However, several critical challenges need to be addressed before approximate computing can move from its infancy and research efforts confined in academic labs to broader adoption by the industry. First, and

foremost, the amenability of various applications and their inherent resilient properties need to be understood and quantified. Second, both programmers and hardware designers require tools that quantitatively evaluate the approximation amenability of a given application. These two aspects necessitate the need of tools and methods that can help programmers and designers reason about the scope and behavior of approximations in applications and effectively evaluate various approximate computing techniques for a given application, or a given technique across a wide range of applications.

To this end, this paper starts off with a taxonomy of related terms in this field and then presents an *open source* toolkit to analyze and study the scope of approximations in applications. Our toolkit, called iACT (Intel's Approximate Computing Toolkit), consists of a compiler, a runtime and a simulated hardware test bed. The compiler portion consists of a LLVM [20] based pragma annotation framework that embeds in the binary of the compiled application programmer provided hints for the hardware. Based on LLVM and Clang, we present an example of a runtime framework that handles approximate memoization during the application's execution. A Pin [26] based framework handles approximation in hardware. We discuss the design of this toolkit and present examples of how to use this toolkit in this paper. As an example on how to use this toolkit, we include two different applications and analyze the scope of approximate computing in these. We believe that this open source toolkit could provide a common collaborative framework for the ongoing research in academia where researches could work on these toolkit and contribute back to the development of this toolkit. We believe this will greatly aid the larger community in adopting approximate computing as an avenue for design optimization.

The rest of this paper is organized as follows - we present a summary of related work and draw a taxonomy for this field in Section 2. Then we present our toolkit design and provide a description of it in Section 3 followed by how to use it and application analysis using the toolkit in Section 4. We present our next steps and discuss some challenges in this field in Section 5 and finally conclude in Section 6.

2. Prior work and taxonomy

There are various proposals from academia as well as industry advocating trading of application output quality, QoS and performance for energy efficiency. These approaches have been varied and have come from several different computer science and electrical engineering research communities. Below is a gist of the prior works.

Architectural Approximate Computation Schemes: Tong et al. [36] examine adapting the floating-point (FP) mantissa width to suit the application requirements. Because FP computations implicitly incorporate imprecision, coarsening the imprecision has a mild effect on some applications. Similarly, work in [3] exploits FP operation memoization (a scheme where instances of an already executed operation are reused). Works in [15, 19, 32–34] argue for using logic circuits that are amenable to voltage-overscaling (lowering operating voltage) alongside units with strict guarantees.

Circuit Techniques Facilitating Approximate Computing: Probabilistic CMOS [2, 7, 8, 14, 17, 18] is a similar concept from the VLSI community that advocates codesign of the technology, architecture, and application to produce approximate ASICs for particular *soft* applications. Research work in [10, 29] has looked at building circuits and ASICs to handle approximate computing.

Application-Level Error Tolerance: Works in [11, 21–23, 37] focus on exploring the tolerance of selected applications to transient faults. All these papers advocate that some parts of the application (some memory regions, some instructions) are much more tolerant to errors in hardware than others and then relax the hardware guarantees to error protection for these code segments.

Compiler Techniques: Language and compiler researchers have explored software-only optimization that trade away strict correctness guarantees. Rinard et al. [1, 28, 30] propose program/code transformations such as "loop perforation". Green [4] is a different approach to imprecise computing that allows the programmer to write several implementations of a single function: a *precise* one and several of varying imperfect precision. A runtime system then continuously monitors application QoS and dynamically adapts to provide the target QoS value.

Language-Exposed Relaxations in Hardware: Some approaches combine architecture-level accuracy loss with programming constructs for exploiting it. Relax [12] allows the programmer to annotate regions of code for which hardware error recovery mechanisms could be turned off. The hardware only performs error detection and the programmer can choose how to handle hardware faults if they occur. Flikker [25] focus on memory rather than logic: it lets the programmer allocate some data in a failure-prone region of main memory. The refresh rates of the main memory cells in these regions are then reduced, saving power at the cost of occasional bit-flips. Work in [35] proposes a parallel architecture that uses language-level error bound expressions to map messages to higher- or lower-reliability communication channels. EnerJ [31] is an extension to Java that exposes hardware faults in a safe, principled manner. Simulation of a selectively reliable hardware, Truffle, suggests that EnerJ programs can save large amounts of energy with only slight sacrifices to quality of service [13].

Industry Effort on Approximate Computing: Intel recently published a variable precision ALU component, called Minerva [16] which is a FMA unit (O = A*B+C). Higher floating-point precision offer improved accuracy but come at the expense of performance and energy efficiency. On the other hand variable-precision floating-point circuits in Minerva provide run-time precision selection leading to accuracy trade-off at the benefits of energy savings.

2.1 Taxonomy

Since this field is relatively new and is starting to draw attention, we present a taxonomy to the readers for a better understanding of topics and terms related to this area.

Computing in hardware (and software) can be precise or imprecise (see Figure 1). With *precise computing* the hardware (and at times the software) always provisions for unexpected errors; if errors are introduced in the hardware, then the hardware corrects the errors without letting the application programmer know about it. For example, if a data bit gets corrupted during transmission over the memory bus, then the hardware's ECC mechanisms identify and correct this error and the software or the user never knows that such an error occurred in the first place.

Imprecise computing on the other hand refers to the fact that the system (typically the hardware) does not produce the *exact* output at all times. This technique of computing originated in the context of the real-time computing [6, 24, 27]. The idea there was to design tasks that always produce a valid result when the timing deadline is reached - if the task completes before the deadline, then the result is the precise value, else it is a valid but imprecise result. This reduces timing errors in real time systems when certain tasks are unable to meet the deadlines because of errors or resource constraints.

We classify imprecise computing into several classes - *variable-precision, soft, probabilistic, stochastic, approximate, quantum* and *non-Boolean* computing.

Variable-precision computing refers to the class of computing where the working precision of the arithmetic supported by the hardware can be dynamically and arbitrarily varied leading to pre-

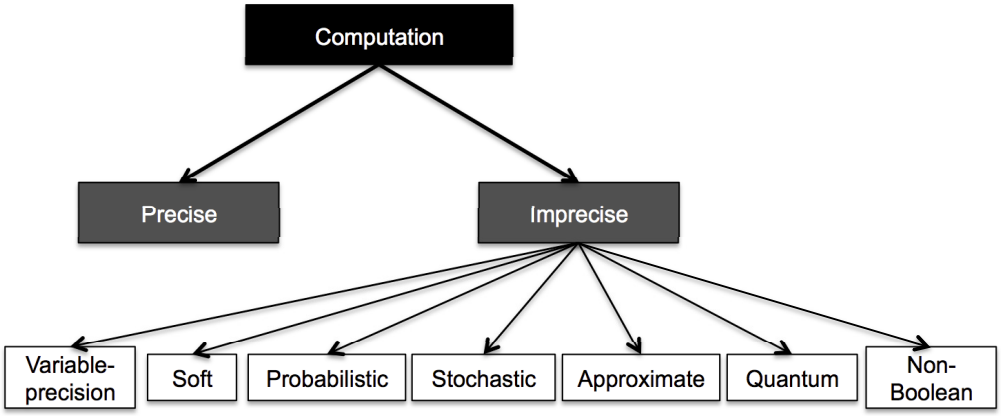

Figure 1. Taxonomy.

cision loss. This is mostly done to save energy while ensuring that the error accumulated is within reasonable limits.

Soft computing is a term applied to the field in computer science that is characterized by the use of inexact solutions to computationally hard tasks (such as the solution of NP-complete problems). Neural networks, machine learning algorithms, fuzzy logic, genetic algorithms, meta-heuristic intelligence (ant colony optimization, swarm optimization, etc.), Bayesian networks, chaos theory, etc. fall under this class of computation. Applications that are amenable to soft computing are tolerant to imprecision, uncertainty, partial truth, and approximations in results at the end (possibly at intermediate steps as well). In effect, the role model for soft computing is the human mind. The guiding principle of soft computing is - exploit the tolerance for imprecision, uncertainty, partial truth, and approximation to achieve tractability, robustness and low solution cost.

Probabilistic computing was motivated by the increasing potential for errors in hardware as we move to smaller technology nodes. The idea here is that certain applications (like Bayesian inference, cellular automata, neural networks, hyper encryption [2]) are resilient to hardware errors and hence, not all the hardware errors need to be caught/detected and corrected. The CMOS devices that become susceptible to perturbations due to noise, soft-errors, variations or reliability are referred to as probabilistic CMOS (or PCMOS) and the computations done using circuits built out of these devices are called probabilistic computations.

Stochastic computing is a some what general formulation of the probabilistic computing paradigm. The idea here is to match up the hardware error distribution with the statistical properties of the application's output to reason about how valid results can be produced even in the presence of hardware errors. Stochastic computing relies on exploiting the somewhat relaxed definition of "correctness" afforded by certain applications (RMS, graphics, video, etc.). Stochastic computing views nanoscale circuit fabrics as noisy communication channels and incorporates statistical behavioral models of the circuit fabric. In doing so, the aim is to develop communications-inspired resilient design techniques based on the well-established foundations of statistical estimation and detection [33]. Specifically, stochastic computation advocates an explicit characterization and exploitation of error statistics due to nanoscale artifacts, as seen at the architectural/algorithmic/system levels. The benefits of such a design philosophy are the gains in robustness and energy-efficiency in presence of a extremely high-degree of unreliability at the circuit fabric.

Approximate computing is a very generic formulation that is inclusive of all the above techniques in a cohesive manner. The idea here is to provide programming language constructs that allow application to express its tolerance to hardware errors or *reduced resource computation* or accuracy in computations, as well as how the application algorithm can adapt to execute within variable amounts of resources. This information can be used by the compiler (or runtime) to perform probabilistic code transformations that might introduce some impreciseness in the final output data. Also, this information is exposed to the hardware (possibly via native machine instructions).

Overall, all the above classes of imprecise computing fall under the over-arching category of reduced-resource computing - where the actual computation or algorithm might vary depending on the amount of resource available to perform the task. The resource can be defined either in terms of physical units required to perform a task, or time or energy. Either the application, or the hardware or the compiler and the runtime is typically is made aware of reduction is resources and the application eventually expects some level of impreciseness in result.

The final two classes under imprecise computing are somewhat different from the rest.

Quantum computing shares theoretical similarities with non-deterministic and probabilistic computers, like the ability to be in more than one state simultaneously.

Non-Boolean computing refers to the class of computing where a single storage bit can correspond to more than 2 values at any instant of time as opposed to binary computing where a bit can only correspond to truth-values (0 or 1).

3. Intel's Approximate Computing Toolkit (iACT)

The iACT consists of (1) a compiler based static framework that embeds in the binary the programmer annotated approximate computing knobs, (2) a runtime framework for approximate memoization and (3) a HW substrate (done through simulations) that exercises the approximations. The key idea is when the programmer writes a program, he/she annotates the approximation amenable functions (code segments) with high-level pragmas and also provides a quality or checker function (the compiler or the adaptive runtime layer can auto-generate the quality or checker function). The high-level pragmas provide hints to the compiler and adaptive runtime system to perform approximation on the desired functions until quality is met. The compiler performs static analysis and then transforms the functions statically for approximation and finally, embeds the programmer specified transformation information (through annotations) in the application binary. When the hard-

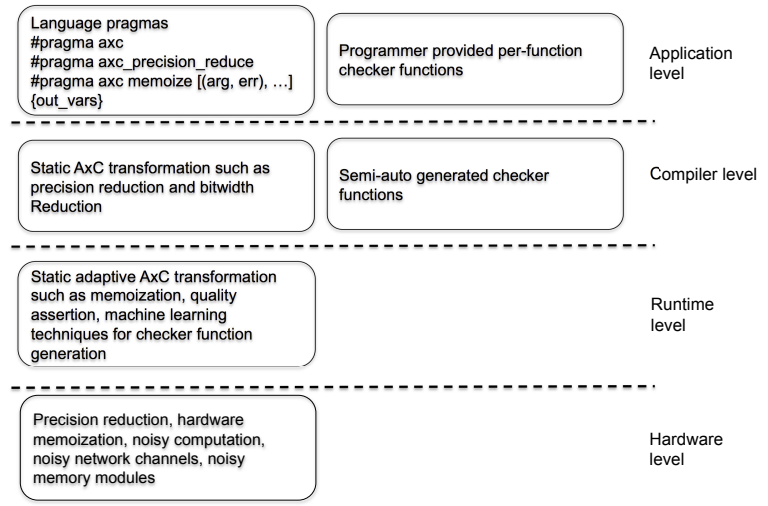

Figure 2. Summary of the capabilities of iACT.

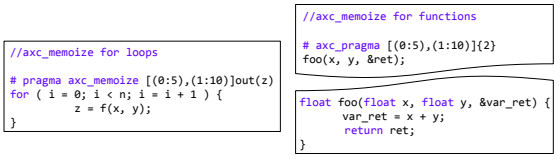

Figure 3. Language extensions via memoize pragma for functions and loops. In this case, a runtime maintains a global table with "in" and "out" parameters; during execution of the application, the runtime either skips the function computation by reading the "out" values directly from the table if the current function inputs are within the specified range or computes the function and populates the table with the "in" and "out" values if the table does not have such an entry.

ware that is running the binary comes across these annotations, the hardware executes the portion of the code with the right set of approximation knobs while the runtime ensures that the eventual function output are within the programmer defined boundaries (specified through the checker functions).

The checker functions could be programmer specified or could also be semi-auto generated by the adaptive runtime layer. In the later case, the program is run on a test-bed and for each of the approximations carried out in the test-bed, the programmer (or the end-user) specifies if the output is acceptable or not. The adaptive runtime learns these approximation-output relationships using a machine learning based framework and auto-generates the checker function, which is then used from there on to guarantee that the function output is within the programmer expected output range. We call this eventual program a user and machine tuned program. This aspect of our toolkit is currently work in progress.

Currently, our toolkit supports three kinds of transformations: an automated variable precision reduction framework, a framework for noisy ALU computation and an approximate memoization framework. The compilation framework is implemented for C/C++ programs and is based on Clang and LLVM framework, and the precision reduction and noisy computation frameworks are based on a Pin tool. Overall, our toolkit enables an application programmer to perform and understand application-level error tolerance and also understand how the underlying hardware could provide native support for approximate computation.

Figure 2 shows a summary of the capabilities of iACT at each layer starting from application level to hardware level. iACT is hosted on https://github.com/IntelLabs/iACT and currently, the compiler framework for pragma support, runtime framework for approximate memoization support, and Pin based tools to simulate *noisy* hardware is released and hosted on the github repository. With the *pragma axc* annotation to a C function declaration, the Pin tools simulate a noisy hardware - noisy arithmetic instructions that operate on floating point values and noisy memory loads and stores. The tools support several different parameterized noise models - probability based, operand bit-position based, bit-width based, etc. with knobs to extend the noise models. The *pragma axc_precision_reduce* annotation for a function declaration down-converts all the floating point values in the function to 16-bit width precision. The idea here being that this pragma could later on support any arbitrary precision arithmetic and aid a programmer figure out what precision levels are appropriate for his/her application.

The *pragma axc_memoize [(arg_num, error_percentage),...]{output_var_list}* annotation for a function at it's call site invokes the runtime approximate memoization framework. The *[(arg_num, error_percentage),...]* annotation specifies error tolerance percentages for function arguments. The *output_var_list* specifies the list of output arguments through which the function returns values to the calling context. Using the above annotation values, the runtime memoization framework creates a global table for each call-site and populates this table during the function execution with new values of the input arguments and their corresponding outputs. For subsequent invocations of the same function, the runtime first tries to find if the incoming values of the arguments are already in the table with their specified error bounds in which case it does not execute the function but returns the results from the table (thus, approximating the output values). Otherwise, the function is executed precisely and new values are stored in the table. The global table size is kept relatively small to reduce the cost of expensive search. Note that, the runtime also checks the quality function before applying memoization approximation. Figure 3 shows how memoization pragmas are specified for loops and functions in our toolkit.

4. Applications

To help researchers in using iACT, we include two applications as part of the first release - bodytracking application from the PARSEC [5] benchmark suite and a Sobel filtering kernel. For this paper, we also describe an in-house machine learning based classification algorithm and describe our experience in analysis of the scope of approximate computing for this application. Since, bodytrack and Sobel filter are released with the iACT, we do not discuss them in this paper in detail and could answer any queries related to these two applications on the repository discussion board. We do describe in detail the classification algorithm and its amenability to approximations.

Bodytracking application: Bodytrack is a computer vision application which tracks a 3D pose of a marker-less human body with multiple cameras through an image sequence. The underlying kernel in this application is an annealed particle filter to track the pose using edges and the foreground silhouette as image features, based on a 10 segment 3D kinematic tree body model. We found bodytrack application to be amenable to several different forms of noisy computation and precision reduction schemes making it a good candidate for approximate computing. Overall, with 16-bit preci-

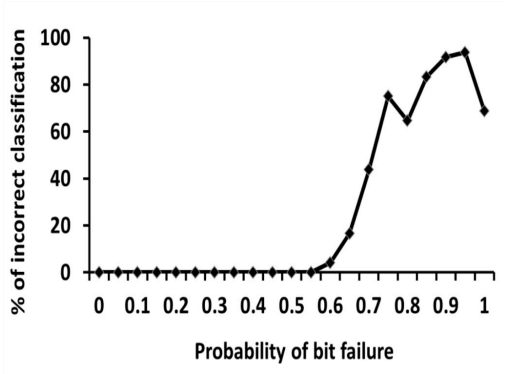

Figure 4. Behavior of the classification workload as the probability of bit flips vary.

sion, bodytrack application provides 22% dynamic energy reduction with less than 4% quality degradation compared to the 64-bit (precise) execution.

Sobel filter: Sobel filter performs a 2-D spatial gradient measurement on an image and identifies regions of high spatial frequency that correspond to edges in the image. Sobel filtering is widely used as a pre-processing step in image processing applications. Sobel filter uses a 3x3 operator and convolves pixels in the image with this operator in a sliding window manner. For the iACT release, we include a very simple implementation of this filtering application in C and show how programmers should annotate the code for invoking the approximate memoization runtime framework. With the approximate memoization scheme, we see dynamic energy savings up to 22% with 10% of the image pixels deviating in value from the execution without using any memoization framework averaged across all the sample images in the repository.

Classification algorithm: Classification problems in machine learning have the general characteristic of first acquiring labeled training samples, which are provided as an input to the algorithm which trains a learning model. Once the model is adequately trained, it is expected to correctly classify any new sample which is an input to this model. An in-house classification algorithm for location identification was considered as a candidate for approximate computing. During the training phase, each location was uniquely represented through a set of samples for a given attribute. This attribute can be image or any property which can be used to differentiate one location from another. A variant of the k-nearest-neighbor based classification algorithm was then used to classify a new set of samples for an unknown location as one of the locations for which the model was trained. The distance metric used in this algorithm is a combination of the Euclidean and Hamming distances. The algorithm was implemented in C++ and a total of 3 functions were considered for approximation. These include: (i) Function to calculate the Euclidean and Hamming distances between two samples and then normalize the Euclidean with the Hamming distance. This involves floating point division and multiplication operations which are approximated through the Pin tool. Note in our application, the distances themselves are calculated using integer operations. (ii) Function which aggregates the effect of all samples and returns the distance from a given neighbor. This involves floating point subtraction and compare operations. (iii) Function which aggregates the distances from all neighbors and classifies the unknown location based on a preset threshold and distances from the k-best neighbors. It also involves floating point comparison and multiplication operations.

Effect of random bit failures: Figure 4 shows the effect of random bit failures on the location identification workload. Random bit failures modeled in the Pin tool are representative of the timing failures which can happen at low-voltage or high-frequency operation modes. As observed from the figure, even at high probability of these failures, the percentage output mismatch or percentage of incorrect classification remains within tolerable limits (<5%). This is primarily due to two reasons: (i) distances between samples go through a max pooling operation before the k-best neighbors are selected. This pooling operation helps to mask the relative variability among the distances corrupted as a result of random bit failures. (ii) The final matching criterion is based on a preset threshold. In scenarios where the distance values are much greater or lesser than the threshold limit, the final match/no-match decision remains unaffected irrespective of the bit failures. However, beyond a certain failure probability (0.6), the output accuracy suffers greatly, leading to an avalanche effect. This behavior has significant implication on the energy-efficiency of the hardware platform running the location identification workload. It shows that if the hardware platform is originally designed to run at a maximum frequency (Fmax) at a given voltage (V), then while running this application, it can potentially run at F $>$ Fmax, thereby improving performance or at V' $<$ V, thus improving the power consumption.

Effect of lower precision: In our workload, the final distance (D) between two samples is first calculated by accumulating the distance (d) of individual features of the two samples and then normalizing it with the number of overlapping features (N) which matched between those two samples. For the case of image samples, features can be specific objects identified from the image samples. The formula used in the original workload is $D = \frac{\sqrt{\sum_{i=1}^{N} d}}{N^{\alpha}}$, where α is a normalization factor. However, the summation of "d" quickly leads to overflow of the half-precision limits. In a modified version of the workload, we therefore implemented the formula to $D = \frac{\sum_{i=1}^{N} \sqrt{d}}{N^{\alpha}}$ to keep the summation of the distances within half-precision limits. This also required changing the initial thresholds for match/no-match. After these two adjustments, the classification accuracy exactly matched the accuracy for single-precision floating point representation. This establishes that alternate approaches to workload implementation can lead to improvement in error-tolerance for the workload under consideration. Lower-precision floating point hardware [16] can take advantage of the above optimizations to further reduce the power consumption.

5. Future Work

We have released the first version of iACT with support for three approximation knobs for now. We plan to continue adding features to this repository based upon community feedback and requests. Features like auto-generating a checker function based on machine learning is currently our top priority. We also plan to actively monitor the discussion boards in the repository and answer queries related to our software release.

6. Conclusions

For approximate computing to become mainstream, architects and designers need to solve many challenges, the first of which is the scope and limits of savings using this scheme. In this quest, this paper describes our open source toolkit, called iACT, which consists of a compiler, a runtime and a simulated hardware platform for testing out various approximation schemes in an application. We believe, this is the first open source simulation test bed to experiment approximate computing schemes (both in software and hardware). Our goal behind releasing these tools is to foster a collaborative research effort across people in academia and industry and help adoption of this topic of approximate computing mainstream.

References

[1] A. Agarwal, M. Rinard, S. Sidiroglou, S. Misailovic, and H. Hoffmann. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical report, MIT, 2009.

[2] B. E. S. Akgul, L. N. Chakrapani, P. Korkmaz, and K. V. Palem. Probabilistic cmos technology: A survey and future directions. In *VLSI-SoC*, 2006.

[3] C. Alvarez, J. Corbal, and M. Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7), 2005.

[4] W. Baek and T. M. Chilimbi. Green: a framework for supporting energy-conscious programming usingcontrolled approximation. In *PLDI*, 2010.

[5] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 72–81. ACM, 2008. ISBN 978-1-60558-282-5.

[6] L. Budin, D. Jakobovic, and M. Golub. Genetic algorithms in real-time imprecise computing. In *Industrial Electronics, 1999. ISIE '99. Proceedings of the IEEE International Symposium on*, pages 84 –89 vol.1, 1999.

[7] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee. Ultra-efficient (embedded) soc architectures based on probabilistic cmos (pcmos) technology. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, DATE '06, 2006.

[8] L. N. Chakrapani, P. Korkmaz, B. E. S. Akgul, and K. V. Palem. Probabilistic system-on-a-chip architectures. *ACM Trans. Des. Autom. Electron. Syst.*, 12(3), 2008.

[9] V. Chippa, S. Chakradhar, K. Roy, and A. Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–9, 2013.

[10] V. K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S. T. Chakradhar. Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. In *DAC*, 2010.

[11] M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *SELSE*, 2009.

[12] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

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

[14] J. George, B. Marr, B. E. S. Akgul, and K. V. Palem. Probabilistic arithmetic and energy efficient embedded signal processing. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '06, 2006.

[15] A. Kahng, S. Kang, R. Kumar, and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *HPCA*, 2010.

[16] H. Kaul, M. Anders, S. Mathew, S. Hsu, A. Agarwal, F. Sheikh, R. Krishnamurthy, and S. Borkar. A 1.45ghz 52-to-162gflops/w variable-precision floating-point fused multiply-add unit with certainty tracking in 32nm cmos. In *ISSCC*, 2012.

[17] P. Korkmaz, B. E. S. Akgul, and K. V. Palem. Energy, performance, and probability tradeoffs for energy-efficient probabilistic cmos circuits. *IEEE Trans. on Circuits and Systems*, 55-I(8), 2008.

[18] P. Korkrnaz, B. E. S. Akgul, and K. V. Palem. Ultra-low energy computing with noise: Energy-performance-probability trade-offs. In *Proceedings of the IEEE Computer Society Annual Symposium on Emerging VLSI Technologies and Architectures*, ISVLSI '06, 2006.

[19] R. Kumar. Computing with stochastic processors: revisiting the correctness contract between software and hardware. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*, ISLPED '10, 2010.

[20] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[21] X. Li and D. Yeung. Exploiting soft computing for increased fault tolerance. In *ASGI*, 2006.

[22] X. Li and D. Yeung. Application-level correctness and its impact on fault tolerance. In *HPCA*, 2007.

[23] X. Li and D. Yeung. Exploiting application-level correctness for low-cost fault tolerance. *Journal of Instruction-Level Parallelism*, 2008.

[24] J. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83 –94, jan 1994.

[25] S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: saving dram refresh-power through critical data partitioning. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ASPLOS '11, 2011.

[26] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. ACM, 2005. ISBN 1-59593-056-6.

[27] H. R. Mahdiani, A. Ahmadi, S. M. Fakhraie, and C. Lucas. Bio-inspired imprecise computational blocks for efficient vlsi implementation of soft-computing applications. *Trans. Cir. Sys. Part I*, 57(4): 850–862, 2010. ISSN 1549-8328.

[28] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard. Quality of service profiling. In *ICSE*, 2010.

[29] D. Mohapatra, V. K. Chippa, A. Raghunathan, and K. Roy. Design of voltage-scalable meta-functions for approximate computing. In *DATE*, 2011.

[30] M. Rinard, H. Hoffmann, S. Misailovic, and S. Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Onward!*, 2010.

[31] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[32] J. Sartori, J. Sloan, and R. Kumar. Stochastic computing: embracing errors in architecture and design of processors and applications. In *CASES*, 2011.

[33] N. R. Shanbhag, R. A. Abdallah, R. Kumar, and D. L. Jones. Stochastic computation. In *DAC*, 2010.

[34] J. Sloan, J. Sartori, and R. Kumar. On software design for stochastic processors. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, 2012.

[35] P. Stanley-Marbell and D. Marculescu. A programming model and language implementation for concurrent failure-prone hardware. In *PMUP*, 2006.

[36] J. Y. F. Tong, D. Nagle, and R. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Trans. VLSI Syst.*, 8(3), 2000.

[37] V. Wong and M. Horowitz. Soft error resilience of probabilistic inference applications. In *SELSE*, 2006.