PANDORA: A Parallelizing Approximation-Discovery Framework (WIP Paper)

Greg Stitt
University of Florida
Gainesville, FL, USA
gstitt@ufl.edu

David Campbell
University of Florida
Gainesville, FL, USA
dcampbell82@ufl.edu

Abstract

In this paper, we introduce PANDORA—a framework that complements existing parallelizing compilers by automatically discovering application- and architecture-specialized approximations. We demonstrate that PANDORA creates approximations that extract massive amounts of parallelism from inherently sequential code by eliminating loop-carried dependencies—a long-time goal of the compiler research community. Compared to exact parallel baselines, preliminary results show speedups ranging from 2.3x to 81x with acceptable error for many usage scenarios.

CCS Concepts • **Software and its engineering → Compilers**; • **Mathematics of computing → Approximation**; • **Computing methodologies → Machine learning approaches**.

Keywords symbolic regression, approximate computing, machine learning

ACM Reference Format:
Greg Stitt and David Campbell. 2019. PANDORA: A Parallelizing Approximation-Discovery Framework (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES '19), June 23, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 5 pages.
https://doi.org/10.1145/3316482.3326345

1 Introduction

Over the last decade, the increasingly common usage of machine learning has resulted in a historic, yet non-obvious, change to computing: whereas traditional computing has focused on semantic correctness, machine learning has made approximation a mainstream design strategy. Similarly, yet counter-intuitively, the common usage of real numbers shows that many applications already tolerate some approximation, including robotics, financial analysis, Internet searches, and even some scientific-computing applications, among others [11]. For other applications (e.g., signal processing), subjective quality often enables numerous approximations that trade off efficiency and quality.

Approximate computing is a technology that exploits acceptable error in these applications to improve performance and/or energy [5, 8]. Existing research automates approximation with specialized frameworks, languages, and/or compilers that apply known approximations while meeting user-specified error constraints [1–4, 13–16]. However, resulting improvements are often modest because effective approximations cannot always be generated via transformations to the original code. In addition, approximations are often too application specific to have widespread compiler support, and the acceptable level of approximation is often subjective, or may even change dynamically (e.g., low battery), which makes "one-size-fits-all" approximations less effective.

In this paper, we introduce PANDORA—a **p**arallelizing **a**pproximati**o**n-**d**isc**o**very f**ra**mework—which uses machine learning to automatically discover application-specific approximations that are specialized for potentially any architecture, and are not derived from transformations to the original code. Unlike most existing approximate-computing approaches that reduce computation, PANDORA also discovers approximations that increase parallelism, even for inherently sequential applications from which current compilers cannot extract parallelism. Although parallelizing compiler transformations have been studied for decades [7], those transformations are restricted by exact correctness. By contrast, PANDORA discovers new approximations via symbolic-regression-based machine learning that eliminates dependencies to exploit parallelism that improves performance or energy given an error constraint. In addition, PANDORA significantly improves scalability by automatically generating system-specialized approximations that reduce communication, data-movement, and synchronization bottlenecks. Furthermore, PANDORA automatically generates a range of Pareto-optimal tradeoffs between error, performance, and/or energy, which enables adaptation to different use cases.

LCTES '19, June 23, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6724-0/19/06…$15.00
https://doi.org/10.1145/3316482.3326345

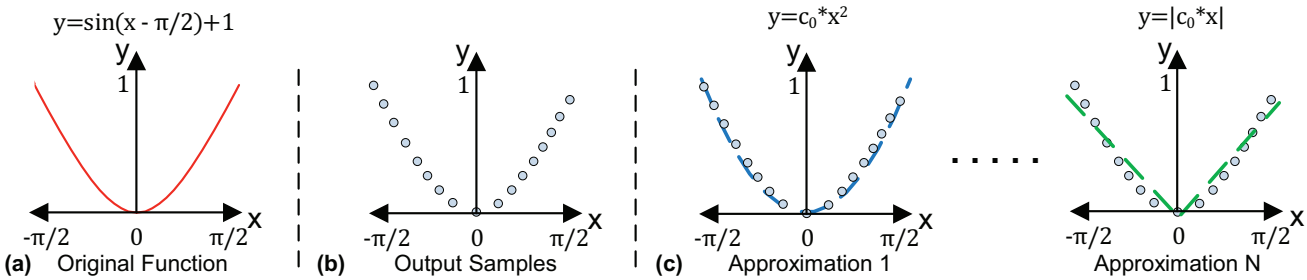

Figure 1. PANDORA (a) approximates an existing function by (b) sampling a number of outputs across the desired input range, and then (c) performing a specialized symbolic regression that optimizes a fitness function that uses error and performance (or any goal) to create a number of Pareto-optimal approximations.

2 PANDORA

As input, PANDORA takes a function to approximate and an optional set of constraints, and outputs a set of Pareto-optimal approximations that are specialized for the targeted architecture. The basic strategy of PANDORA is to first replace the original function with samples of the function's output. PANDORA exploits the fact that there is an infinite number of functions that coincide, or nearly coincide, with the samples to find an alternative function that is cheaper or more parallel than the original function. To find such a function, PANDORA performs a specialized form of symbolic regression, which searches the space of all mathematical equations to automatically discover the model of a dataset. However, whereas symbolic regression is generally focused on finding a function that minimizes error, PANDORA is also concerned with finding functions (i.e., approximations) that have desirable computation or communication characteristics for a given architecture.

For illustrative purposes, Figure 1(a) demonstrates a simple example of a sine wave restricted to the range of $-\pi/2$ to $\pi/2$. Figure 1(b) shows the sampled output of the function. Figure 1(c) demonstrates two example regressions that approximate the original function within the restricted range: a parabola and a piecewise linear regression. Approximating larger ranges can be achieved via piecewise decomposition, where if-statements first check the range and then apply the corresponding approximation. However, in many cases, a user may want the range to be restricted to values used by the application. Although this simple example does not demonstrate increased parallelism, it is intended to show the basic strategy of PANDORA. In general, PANDORA can trade off error for increased performance to support different use cases, where at one extreme is the original function (low performance, no error) and at the opposite extreme is a constant (high performance, prohibitive error).

To evaluate PANDORA, we developed a symbolic-regression framework in Python that extends the DEAP [6] evolutionary computation library with additional genetic-programming capabilities. The presented experiments approximate Python functions, but the framework can support any language by simply sampling a function's outputs. Given a function, the framework assembles approximations using genetic programming [9, 10] built from basic mathematical primitives (addition, multiplication, sine, log, etc.) combined with existing coarse-grained approximations (neural nets, perceptrons, hidden Markov models), while using standard crossover strategies (e.g., one-point crossover, one-point leaf-biased crossover) and mutation strategies (e.g., uniform, node replacement, random subtree insertion, random subtree removal). To guide genetic programming, we provide PANDORA with a fitness function that considers the parallelism of the approximation, in addition to goal-specific metrics (performance, energy estimations) and application constraints (power, error). Performance and energy estimations include system-specific characteristics (e.g., communication bandwidth limits), which enable genetic programming to modify the approximation to avoid bottlenecks. For example, if data movement becomes a bottleneck, then genetic programming would prioritize approximations with reduced communication (e.g., by eliminating inputs and/or synchronization). For each generated approximation, PANDORA computes several error metrics (e.g., root-mean-square, mean absolute-percentage, etc.) by comparing the approximation and the original code with random inputs. Although approximation discovery times generally range from seconds to hours, we ran these experiments for days and then collected results of the best approximations found at that time.

Figure 2 demonstrates the benefits of the main envisioned use of PANDORA: replacing loop-carried dependencies with approximations that have independent (i.e., parallel) iterations. To show tradeoffs between error and performance, we generated multiple approximations using different error constraints. We evaluate this use case using recurrence relations and show that PANDORA is able to automatically find closed-form solutions or solutions with parallel iterations. For example, PANDORA automatically converts the infinite impulse response (IIR) filters into finite impulse response (FIR) filters, with no pre-defined knowledge of such concepts.

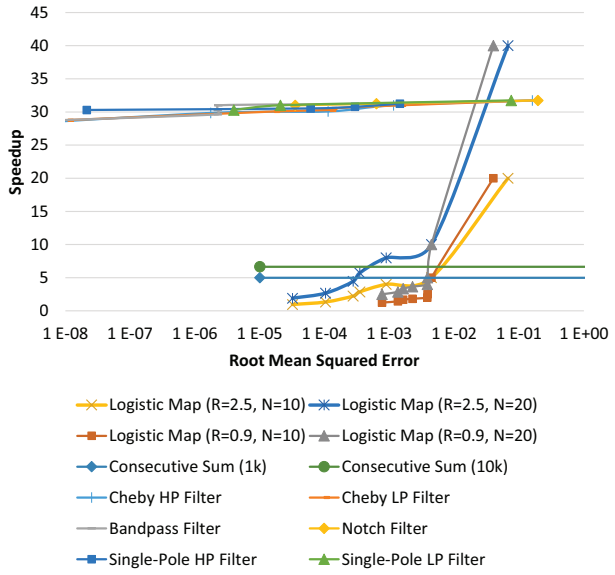

Figure 2. Speedup from elimination of loop-carried dependencies via automatically discovered parallel approximations with constrained input bandwidth.

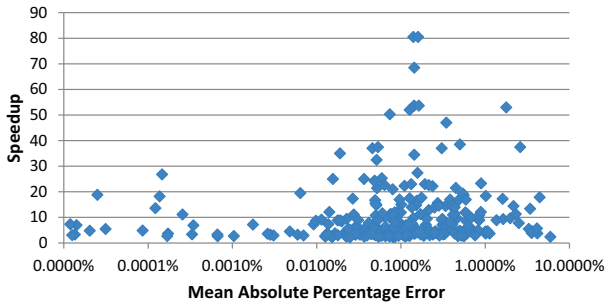

Figure 3. Approximation tradeoffs between speedup and error (log scale) for 336 tested applications.

Although there are known closed-form solutions and approximations for some examples, the key point is that PANDORA automatically finds a closed-form solution or a parallel approximation. Such functionality is critical due to the vast majority of real functions not having existing approximations, and due to the common use of application-specific approximations that are too unique for compiler support.

We created each example by writing corresponding Python code, but we analyzed performance independently from language and architecture by comparing the depth of the approximation's dataflow graph (DFG) with the depth of the original DFG after applying unrolling and tree-height reduction. We use the depth of the DFG because the depth represents the length of the longest dependence chain, which puts a bound on execution time. In general, a more parallel solution tends to have a smaller depth than a sequential solution, where loop-carried dependencies will create a sequence of iterations in the unrolled DFG. Although some architectures may not provide enough resources to achieve such parallelism, for these examples, the comparison is applicable to most architectures. Application-specific parameters (e.g., input sizes, constants) used in the experiments are specified in the legend. Because recurrence relations with closed-form solutions could have arbitrarily large speedup by using large inputs, we chose a range of input sizes that illustrate the basic trends.

To evaluate realistic use cases, the speedup in Figure 2 constrains input bandwidth to existing PCIe bandwidth. The results demonstrate several trends. First, all of the filter examples tend to achieve a significant speedup of around 30x with low error, which increases slightly as more error is allowed. The second trend is shown by the consecutive-sum examples, which demonstrated flat speedups of 5x and 7x for input sizes of 1k and 10k. This speedup does not increase for additional error due to PANDORA finding an exact closed-form solution that is simple enough to not benefit from additional approximation. The third trend is shown by the remaining examples, for which PANDORA found closed-form solutions that demonstrated a rapid increase in speedup from 1x to 40x for different error constraints due to more complex solutions.

The most interesting example in Figure 2 is the logistic-map approximation. Despite having no known closed-form solution, PANDORA found approximations with a root-mean-square error (RMSE) of less than 1E-4. Although not an exact solution, that level of error can be treated as a solution in many use cases. Furthermore, the discovery capabilities are demonstrated by the counter-intuitive characteristics of the generated approximation. Despite the logistic map only using multiplication and subtraction, the approximation uses a square root, cosine, and hyperbolic tangent.

When not limited by input bandwidth (e.g., internal memory), the results showed two trends (not in figure): (1) significantly higher speedups ranging from ~200x to ~4000x, and (2) significant improvements from increasing error.

Figure 3 presents 336 randomly generated DFGs representing unrolled loops with loop-carried dependencies that were approximated with a mean-absolute percentage error constraint of 1%. The DFGs contained random types of floating-point operations with multiple non-associative operations, random numbers of inputs, and random numbers of operations ranging from 20 to 160. The results show a wide range of speedups from 2.3x to 81x, with a trend towards larger speedups for larger error. The overall averages for speedup and error were 9.5x and 0.3%, respectively. 93% of the examples were able to meet the error constraint, with many examples using orders-of-magnitude less error. The remaining 7% achieved errors between 1% and 5.9%.

Interestingly, when repeating the tests with integers, some approximations had 0% error. To our knowledge, no previous

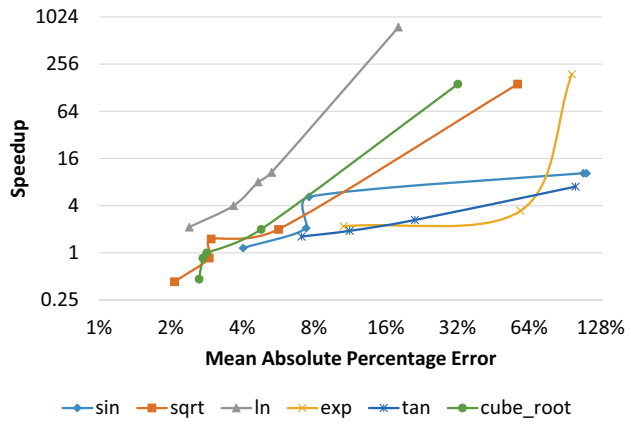

Figure 4. FPGA speedup ($\log_2$ scale) of reduced-area approximations compared to Intel IP cores.

compilation approach can automatically convert a series of non-associative operations into an exact parallel alternative.

Another interesting result was the wide range of different approximations. In addition to increasing parallelism, PANDORA occasionally removed iterations of a loop that had little effect on error. Such an approximation is an existing technique known as loop perforation [3, 17], but the significance here is that this approximation was discovered automatically. In most cases, PANDORA both reduced the total operations and parallelized those operations. For five examples, PANDORA increased the number of operations to enable a parallel approximation. In addition, PANDORA often eliminated some of the original inputs, which we envision being useful for eliminating communication bottlenecks and improving scalability, which could be further improved by integrating dimensionality-reduction techniques.

Figure 4 demonstrates parallelizing approximations specifically for FPGAs, where parallelism is often limited by available resources. In these experiments, we compare achievable parallelism (i.e., number of IP cores that would fit in the FPGA) between Intel-provided IP cores (synthesized for Arria 10) and automatically generated approximations. By reducing the resources needed to implement each core, the approximations enable FPGA designs to achieve significant increases in spatial parallelism compared to Intel's optimized IP cores. The results show speedups between 1x and 10x for mean-absolute percentage errors below 5%, with rapid increases in speedup to over 100x for larger errors. *ln* achieved a speedup of 757x at 18% error. Although the larger errors are unlikely to be acceptable, we include the tradeoffs to demonstrate upper bounds on performance. The results assume equal clock frequencies, which likely makes the speedup pessimistic due to most of the approximations using finer-grained operations that support higher frequencies.

3 Limitations and Future Work

Although the results from the previous section demonstrate considerable potential, PANDORA has several technical limitations that need to be addressed to make the framework more widely usable. One primary challenge is that symbolic regression is known to be a considerably challenging problem, with most existing strategies only working well for toy problems [12]. We plan to address this challenge with hardware-accelerated symbolic regression that performs over a million times faster than existing software implementations, according to our preliminary results.

Another fundamental limitation is that by using machine learning, PANDORA can only make probabilistic guarantees about the error of the approximation. However, this same limitation applies to all machine-learning techniques, which are widespread in many application domains.

Another limitation is the tendency for unintuitive approximations. As a result, if a designer needed to understand or debug the approximation, it would be difficult to do so. Interestingly, this limitation is also one of the biggest advantages because these non-obvious solutions suggest that an existing compiler is unlikely to include, or a designer is unlikely to manually create, approximations of similar quality.

4 Conclusions

In this paper, we introduced the PANDORA framework for automatically discovering parallelizing approximations that are specialized for a targeted architecture. Unlike existing approximate-computing work that focuses on languages and compilers that transform the original code into an approximation, PANDORA leverages a specialized symbolic-regression strategy that discovers a new approximation that is not based directly on the original code. Instead, PANDORA samples the outputs of the original function and then leverages the fact that there are an infinite number of functions that coincide, or nearly coincide, with those samples to search for an alternative function that can be computed more efficiently.

We demonstrated that PANDORA can remove loop-carried dependencies from recurrence relations, while also increasing parallelism in the presence of non-associative operations. Finally, we generated FPGA-specific approximations to reduce the resources needed to implement a number of FPGA functions, which demonstrated attractive tradeoffs between error and performance. Although there are still technical challenges preventing PANDORA from being widely usable, this work-in-progress demonstrates Pareto-optimal tradeoffs to the decades-long compiler challenge of parallelizing sequential code.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1149285 and CNS-1718033.

References

[1] J. Ansel, Y. L. Wong, C. Chan, M. Olszewski, A. Edelman, and S. Amarasinghe. 2011. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO 2011)*. 85–96. https://doi.org/10.1109/CGO.2011.5764677

[2] Woongki Baek and Trishul M. Chilimbi. 2010. Green: A Framework for Supporting Energy-conscious Programming Using Controlled Approximation. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, New York, NY, USA, 198–209. https://doi.org/10.1145/1806596.1806620

[3] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. 2013. Verified Integrity Properties for Safe Approximate Program Transformations. In *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation (PEPM '13)*. ACM, New York, NY, USA, 63–66. https://doi.org/10.1145/2426890.2426901

[4] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. 2013. Verifying Quantitative Reliability for Programs That Execute on Unreliable Hardware. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 33–52. https://doi.org/10.1145/2509136.2509546

[5] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 449–460. https://doi.org/10.1109/MICRO.2012.48

[6] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. 2012. DEAP: Evolutionary Algorithms Made Easy. *Journal of Machine Learning Research* 13 (July 2012), 2171–2175.

[7] Milind Girkar and Constantine D. Polychronopoulos. 1995. Extracting Task-level Parallelism. *ACM Trans. Program. Lang. Syst.* 17, 4 (July 1995), 600–634. https://doi.org/10.1145/210184.210189

[8] J. Han and M. Orshansky. 2013. Approximate computing: An emerging paradigm for energy-efficient design. In *2013 18th IEEE European Test Symposium (ETS)*. 1–6. https://doi.org/10.1109/ETS.2013.6569370

[9] Gregory S. Hornby. 2006. ALPS: The Age-layered Population Structure for Reducing the Problem of Premature Convergence. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation (GECCO '06)*. ACM, New York, NY, USA, 815–822. https://doi.org/10.1145/1143997.1144142

[10] John R. Koza. 1994. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge, MA, USA.

[11] Logan Kugler. 2015. Is "Good Enough" Computing Good Enough? *Commun. ACM* 58, 5 (April 2015), 12–14. https://doi.org/10.1145/2742482

[12] James McDermott, David R. White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reilly. 2012. Genetic Programming Needs Better Benchmarks. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation (GECCO '12)*. ACM, New York, NY, USA, 791–798. https://doi.org/10.1145/2330163.2330273

[13] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. 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 (OOPSLA '14)*. ACM, New York, NY, USA, 309–328. https://doi.org/10.1145/2660193.2660231

[14] Sasa Misailovic, Stelios Sidiroglou, and Martin C. Rinard. 2012. Dancing with Uncertainty. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2414729.2414738

[15] T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. 2015. SNNAP: Approximate computing on programmable SoCs via neural acceleration. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 603–614. https://doi.org/10.1109/HPCA.2015.7056066

[16] Lakshminarayanan Renganarayana, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. 2012. Programming with Relaxed Synchronization. In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES '12)*. ACM, New York, NY, USA, 41–50. https://doi.org/10.1145/2414729.2414737

[17] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. 2011. Managing Performance vs. Accuracy Tradeoffs with Loop Perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, New York, NY, USA, 124–134. https://doi.org/10.1145/2025113.2025133