PANDORA: An Architecture-Independent Parallelizing Approximation-Discovery Framework

GREG STITT and DAVID CAMPBELL, University of Florida, USA

In this paper, we introduce the PANDORA framework for automatically discovering application- and architecture-specialized approximations of provided code. PANDORA complements existing compilers and runtime optimizers by generating approximations with a range of Pareto-optimal tradeoffs between performance and error, which enables adaptation to different inputs, different user preferences, and different runtime conditions (e.g., battery life). We demonstrate that PANDORA can create parallel approximations of inherently sequential code by discovering alternative implementations that eliminate loop-carried dependencies. For a variety of functions with loop-carried dependencies, PANDORA generates approximations that achieve speedups ranging from 2.3x to 81x, with acceptable error for many usage scenarios. We also demonstrate PANDORA's architecture-specialized approximations via FPGA experiments, and highlight PANDORA's discovery capabilities by removing loop-carried dependencies from a recurrence relation with no known closed-form solution.

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

Additional Key Words and Phrases: symbolic regression, approximate computing, machine learning

ACM Reference Format:
Greg Stitt and David Campbell. 2018. PANDORA: An Architecture-Independent Parallelizing Approximation-Discovery Framework. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Since the introduction of computers, traditional design practices have focused on achieving exact semantic correctness of an application and/or system. However, the rapid adoption of machine learning over the past decade has resulted in the emergence of mainstream computing strategies where approximation is commonly accepted [24, 41, 49]. Even before machine learning, approximation was a widespread, but largely unrecognized, practice due to the impossibility of representing real numbers with finite precision. Such finite precision suggests that many applications—even those using double-precision arithmetic—already tolerate approximation, including signal processing, robotics, financial analysis, Internet searches, among others [28]. Even scientific-computing applications, which are known for their precision constraints, are inherently approximate due to the use of real numbers and the common discretization of continuous processes. For other applications, subjective quality often enables numerous approximations that trade off efficiency and quality. For example, many signal-processing applications can tolerate occasional incorrect pixels or frequent small inaccuracies where many pixels may be a slightly different color, where compression artifacts may be more apparent, etc.

Manuscript submitted to ACM

Approximate computing [17, 22, 28] is an emerging area of research that looks to exploit this inherent imprecision to explore more effective approximation strategies within acceptable error constraints. With the end of Dennard Scaling [4], and with Moore's Law decreasing [25], approximate computing provides a promising way of meeting the rapidly increasing demands of future applications without relying on additional transistors.

Current approximate computing research focuses largely on specialized compilers, optimization frameworks [3, 33, 36, 44], and/or specialized programming languages [2, 8, 9] that enable designers to specify the acceptable error of different regions of code [33, 36] and then apply approximations that meet the error constraint. There has also been a significant focus on algorithmic and data-type approximations [36, 45, 46, 55] and approximate architectures [10, 12, 14, 16, 21, 31, 39, 50, 54]. Although current work provides many technical benefits, these approaches suffer from several significant limitations.

The first limitation is that approximations are often too application-specific to be discovered or supported by compilers. For example, there are many unique application-specific approximations [36, 45, 46, 55] and optimizations [40, 43, 47] that a compiler can't achieve via a sequence of general code transformations. In addition, compilers are unlikely to provide a built-in set of such approximations because they have very limited applicability. Similarly, most application developers are unlikely to be aware of such niche approximations to apply them manually, and are even more unlikely to be able to create new approximations. Furthermore, many approximations are architecture-specific, which further decreases the likelihood of manual discovery and/or adoption by compilers.

Another limitation is that existing techniques only provide limited improvements up to 2x in performance and/or energy (e.g., [1, 11, 14, 16, 21, 37, 50, 51, 53, 54]). Although beneficial, these improvements are often modest compared to the 10x to 1000x improvements already provided by graphics-processing unit (GPU) and field-programmable gate array (FPGA) accelerators [19, 52].

We address these limitations by introducing a **p**arallelizing **a**pproximatio**n**-**d**isc**o**very f**ra**mework called PANDORA. Unlike many existing approximation approaches that derive approximations through a series of transformations to the original code, PANDORA uses machine learning to automatically discover application-specific approximations that are specialized for potentially any architecture. One key contribution of PANDORA is the use of approximation to increase parallelism and amenability to acceleration. Although numerous studies have introduced parallelizing transformations in compilers [5, 6, 15, 20, 26, 29], compilers require such transformations to be functionally equivalent. By dropping this requirement, PANDORA enables exploration of significantly more parallelization options, while also integrating and complementing existing techniques that reduce computation. Although several existing approaches also use approximation to increase parallelism [33, 36, 38], those approaches either explore a more restricted approximation space, typically by applying synchronization-relaxing approximations to an existing application [35, 36, 44], or use neural-net-based approximation [17, 38], which can require a huge computational overhead to approximate some functions. By contrast, our approach uses symbolic-regression-based machine learning to generate completely different algorithms, while using fitness functions that maximize parallelism or other optimization goals (e.g., performance, energy), and improve scalability by avoiding system-specific bottlenecks (e.g., data-movement, synchronization), while also meeting different constraints (e.g., error, power, performance). In other words, our approach evolves a custom parallelized approximation that explores a much larger parallelization space, of which previous approaches are a small subset.

Compared to exact parallel baselines, we present preliminary results showing speedups from 2x to 40x over a range of error constraints by eliminating loop-carried dependencies from well-known recurrence relations, which increases to 4,000x when not restricted by I/O bandwidth. We complement these experiments with 336 synthetic loops that

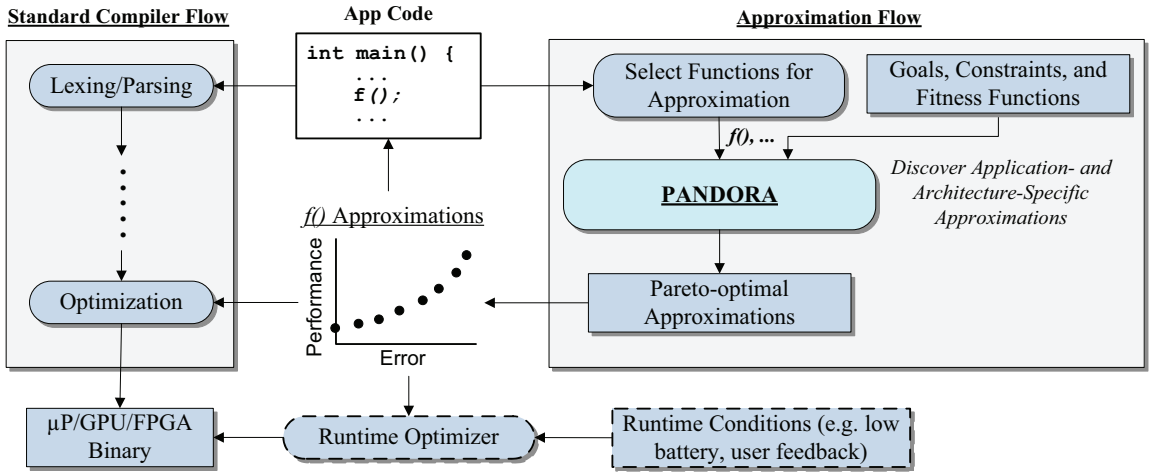

Fig. 1. Overview of how PANDORA integrates with existing compilers and runtime optimizers.

current compilers can only implement sequentially and show speedups ranging from 2.3x to 81x with an average speedup of 9.5x and an average error of 0.3%. We also demonstrate amenability to architecture-specific acceleration with FPGA-specialized approximations that achieve speedups from 2x to 10x for errors under 5%, and up to 757x for an error of 18%.

The paper is organized as follows. Section 2 gives an overview of approximation flows and how PANDORA complements existing compilers. Section 3 describes our approach to approximation discovery. Section 4 presents experimental results. Section 5 discusses related work. Section 6 describes limitations and ongoing work to enable more widespread usage of the presented techniques. Section 7 presents conclusions.

2 PANDORA OVERVIEW

Figure 1 illustrates the envisioned usage of PANDORA with compilers and runtime optimizers. PANDORA complements existing compilers by automatically creating new approximations for provided code. In the envisioned approximation flow, the application code would initially be passed to a tool to determine which functions are the best approximation candidates, how much approximation each function can tolerate, etc. This step is the subject of numerous existing studies (e.g., [33, 36]) and is not the focus of this paper. Instead, we can leverage any of the ongoing work in that area to provide PANDORA with a set of functions to approximate. In addition, PANDORA also takes as input a set of optimization goals, application constraints, and fitness functions for the targeted architecture. These fitness functions, discussed in more detail in Section 3, guide the discovery process by ensuring constraints are met, while estimating metrics (performance, energy) of an approximation on a given architecture.

In our envisioned usage, optimization goals would be generally decided by the application designer. Constraints would be provided by either the designer, or by existing tools that determine how much approximation can be tolerated by a given application. Fitness functions would generally be provided in a library, but may require custom specification for performance and/or energy estimation on specialized architectures (e.g., FPGAs).

Given these inputs, PANDORA automatically discovers a set of Pareto-optimal tradeoffs between performance (or any optimization goal) and error. PANDORA provides those approximations back to the compiler to apply to the original

code, or could potentially just generate a modified version of the original code that can be compiled. This latter approach has the benefit of the compiler applying additional architecture-specific optimizations to the generated approximation. Due to the increased compilation times for discovering approximations, we envision PANDORA being used as a final optimization step (e.g., -O3), although preliminary results often only took on the order of minutes. We also envision an offline approach where PANDORA concurrently searches for approximations in the background during application development, and provides them to the compiler as the approximations are discovered.

Alternatively, PANDORA can also be used to provide approximations with a range of tradeoffs to a runtime optimizer. Whereas static approaches are restricted to a single approximation, the use of PANDORA with a runtime optimizer enables adaptive optimizations that could increase approximation based on appropriate runtime conditions (e.g., low battery life). Static approximations also restrict designers to a single definition of error, which will generally be pessimistic to support the most-demanding users, leaving less-demanding users with untapped efficiency improvements. Furthermore, even the same user could have different opinions of acceptable error in different situations (e.g., noisy vs. quiet environments). PANDORA avoids this one-size-fits-all restriction by generating a range of Pareto-optimal approximations that enables different approximations to be used in different usage scenarios.

3 PARALLELIZING APPROXIMATION DISCOVERY

In this section, we discuss PANDORA's automatic discovery of parallelizing approximations for potentially any architecture. Although we use the term *function* for simplicity, the approach applies to any level of granularity (e.g., loops, basic blocks, statements).

Whereas existing approximating compilers implement many computation-reducing approximations by replacing portions of a dataflow graph with known approximations (e.g., [8, 17, 30, 38, 42, 44]), automatically creating parallelizing approximations that aren't derived from a series of transformations to the original code is a far more difficult problem. This challenge is highlighted by the limited existence of manually introduced parallelizing approximation strategies. In many cases, a parallelizing approximation may not even be known, especially for a particular architecture or combination of resources.

In addition to approximation discovery, another key contribution is the use of approximation to significantly increase the exploration space for automatic parallelization, which in turn makes more applications amenable to FPGA and GPU acceleration. In fact, our preliminary experiments show that in some cases increasing the amount of computation—a strategy that to our knowledge is not considered by any current approximation approach—can enable significant amounts of untapped parallelism that greatly outweighs any extra operations. In addition to exploiting parallelism to maximize performance, PANDORA can alternatively optimize for energy while meeting a power constraint and/or performance constraint. Furthermore, while increasing parallelism alone might result in communication or memory bottlenecks, PANDORA can use system-specific fitness functions to generate specialized approximations that avoid such bottlenecks, providing improved scalability.

PANDORA complements conceptually similar approximate-computing research that increases parallelism via synchronization relaxing [7, 35, 36, 44], and approaches that use FPGA- and GPU-amenable approximations (e.g., neural nets [17, 38]). By using symbolic regression to evolve parallel approximations based on completely new algorithms, our approach both includes and significantly expands the parallelization space of previous strategies. Such expanded exploration is critical for identifying approximations where existing neural-net approaches have high overhead. PANDORA is also conceptually similar to Paragen [13], which was a compiler technology that used genetic programming

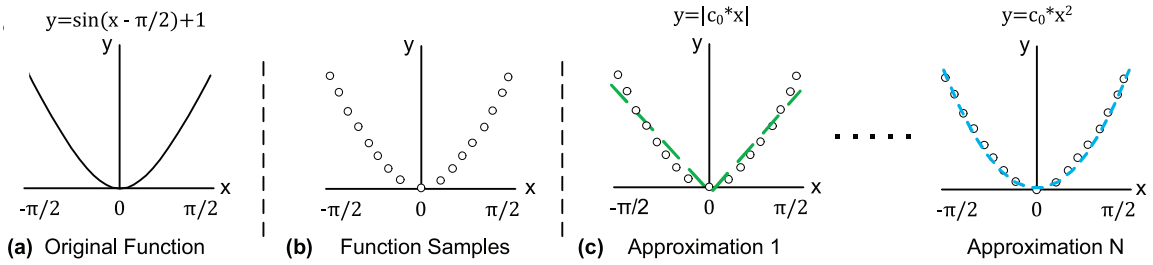

Fig. 2. PANDORA discovers approximations of (a) an existing function by (b) sampling outputs from within a relevant input range, and then (c) performing a specialized symbolic regression to find equations with Pareto-optimal tradeoffs between error and performance (or any goal).

to parallelize sequential software for multi-core processors. However, like other compilers, Paragen was restricted to generating functionally correct alternatives, and did not support arbitrary architectures such as FPGAs and GPUs.

3.1 Approach

To discover approximations, PANDORA samples the original function's output across an input range of interest. After replacing the original function with samples, PANDORA exploits the fact that there are infinite functions that coincide, or nearly coincide, with the samples. PANDORA searches these alternative functions looking for ones that are cheaper computationally, more parallel, lower energy, etc. than the original function. Although counter-intuitive, replacing the original function with samples enables the possibility of discovering numerous approximations that cannot be derived via transformations to the original code, which we show is critical for both increasing parallelism and specializing an approximation for a given architecture.

To find such an approximation, PANDORA performs a specialized form of symbolic regression, which is the problem of searching the space of all mathematical equations to automatically discover the model of a given dataset. However, whereas symbolic regression is solely 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.

Figure 2(a) demonstrates a simple example of approximating a sine wave within the input range of $-\pi/2$ to $\pi/2$. Figure 2(b) shows the sampled output of the function. Figure 2(c) demonstrates two example regressions that approximate the original function within the restricted range: a a piece-wise linear regression and a parabola. For this simple example, the parabola approximation requires two multiplications, but has higher accuracy than the piece-wise linear approximation that only requires one multiplication.

For these simple approximations, larger ranges can be achieved via piece-wise decomposition of the input space, where if-statements first check the range and then apply the corresponding approximation. However, in most cases, the approximation will automatically adapt to the entire range. Additionally, in many cases, a user may want the range to be restricted to values used by the application. Although this intentionally simple example does not demonstrate increased parallelism, we present experiments in Section 4 that significantly increase parallelism and in some cases remove loop-carried dependencies. 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, likely prohibitive error).

With this formulation of the problem, approximation discovery requires effective solutions to symbolic regression. Most existing techniques for symbolic regression rely on genetic programming [23, 27]. To evaluate PANDORA, we developed a custom symbolic-regression framework in Python that extends the DEAP [18] evolutionary computation library with additional genetic-programming capabilities. The experiments in this paper approximate functions written in Python, but PANDORA supports any language by discovering approximations based on output samples, as opposed to language-specific constructs.

To guide genetic programming, the framework takes a configuration file as input that specifies a number of options. First, the configuration file specifies the sampling strategy for training and testing, which currently allows for uniform sampling and random sampling, while also specifying the number of samples to use in each input dimension, and the range of values for those samples.

Next, the configuration file specifies the fitness function, which includes an optimization goal and any constraints. PANDORA supports a variety of existing fitness functions, and is easily extendable to support other functions. For example, in the simplest case, a user could select a fitness function to minimize root mean square error. In this case, PANDORA essentially performs traditional symbolic regression without any consideration of performance or energy of the resulting approximation. Typical fitness functions include minimizing error given a performance/energy constraint or maximizing performance/energy given an error constraint. PANDORA allows for specification of any error metric, but currently supports root mean square error, mean square error, and mean absolute percentage error. In general, an ideal fitness function would provide an exact performance estimate for a given architecture. However, since determining highly accurent performance estimates may require lengthy computation or even simulations that would result in prohibitive training times, we expect most use cases to perform coarser estimations. Ultimately, the accuracy requirement of the performance estimate depends on the use case. For approximations providing small performance improvements (e.g., 5% to 10%), a more accurate estimate is needed. However, for approximations that achieve 2x to 10x improvements, more error can be tolerated in the performance estimate.

To create a fitness function, we provide an architecture-specific performance-estimation heuristic that is applied to each approximation. Although any performance estimation technique can be used, most of our experiments use an estimate that is a function of the depth of the resulting approximation tree structure. By optimizing for tree depth, genetic programming tends to find solutions that do more operations in parallel, since such parallelism tends to make the tree wider while reducing the depth. For FPGA experiments, the fitness function uses the resource requirements of the approximation to determine how many operations can fit on the FPGA, which also determines how many operations can be done in parallel to improve performance. As a result, genetic programming tends to reduce the resource requirements of the original code so that more operations can occur in parallel, so that the function can be replicated more times, etc. In general, 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).

After specifying the fitness function, the configuration file allows specification of primitives from which to build the approximation during genetic programming. PANDORA includes basic mathematical primitives (addition, multiplication, sine, log, etc.), in addition to combined with existing coarse-grained approximations (neural nets, perceptrons, hidden Markov models). New primitives can easily be added simply by defining a function for the primitive, and adding that function, along with definitions of its parameters and return values, to the code. The configuration settings also include a large number of genetic-programming configuration options such as population size, number of generations,

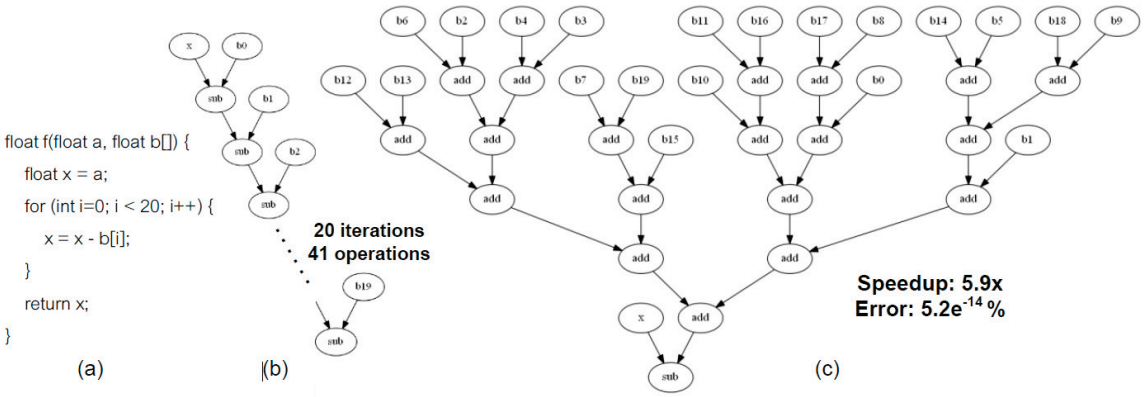

Fig. 3. (a) Example of a loop whose (b) unrolled DFG has no parallelism due to loop-carried dependencies and non-associative operations. PANDORA discovers (c) an application-specific approximation with significant amounts of parallelism and acceptable error for most use cases.

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).

3.2 Example Approximations

To illustrate usage of PANDORA, we provide several simple motivating examples to explain the functionality. Figure 3(a) shows example code of a simple loop with loop-carried dependencies. Optimizing this type of loop is a well-known compiler challenge because the iterations are dependent, and unrolling the loop creates a long sequence of dependent operations with no parallelism (Figure 3(b)). Although compilers can sometimes parallelize similar examples via tree-height reduction, those optimizations only work for associative operations. This example uses non-associative subtractions, which prevents traditional parallelization. Although an integer version of this example could potentially be parallelized by adding all of *b[]* with an adder tree and then subtracting from *x*, a compiler would be unlikely to implement such a rarely applicable optimization. PANDORA has the key advantage of being able to automatically generate such application-specific parallelizations.

For this example, we assume all operations take the same time, which makes sequential performance equal to the total operations (41 for this example) and parallel performance equal to the maximum depth of the tree. We also assume sufficient resources for maximum parallelism, which can easily be obtained on CPUs, FPGAs and GPUs for these examples. To approximate this function with PANDORA, we use a fitness function that minimizes depth of the tree without consideration of error, which generates a range of tradeoffs.

Figure 3(c) illustrates an approximation generated by PANDORA. The more balanced tree structure shows significantly increased parallelism compared to the original code, achieving a 5.9x speedup, with a mean absolute percentage error of only $5.2e^{-14}$%, which we determined using 10,000 uniformly distributed random inputs between -32k and 32k.

For this approximation, the only error was due to the non-associativity of floating-point operations. Existing compilers either ignore the error introduced by the order of floating-point operations, prevent any optimization that introduce error, or allow the designer to specify different optimization goals (e.g., precision, fast, strict [54]). PANDORA provides another alternative that identifies Pareto-optimal tradeoffs between error and performance. Although previous work has also identified tradeoffs for floating-point applications [135], those approaches are a subset of the potential

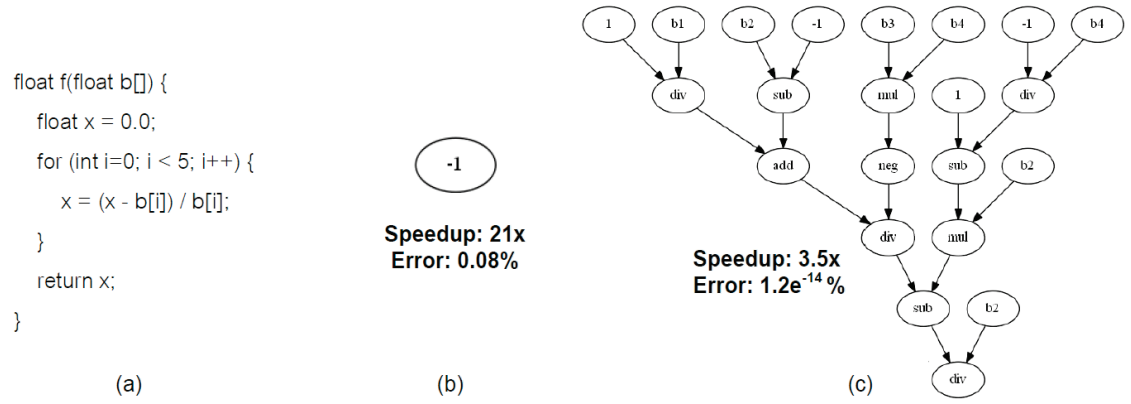

Fig. 4. (a) Example of a loop where PANDORA creates different specialized approximations for (b) random inputs ranging from -32k to 32k, and (c) random inputs between 0 and 1.

approximations applied by our approach. In addition, PANDORA discovers these tradeoffs automatically as part of the many implicitly explored approximation strategies enabled by genetic programming, without specifically trying to optimize floating-point operations.

Figure 4 illustrates another example showing how different inputs can benefit from different approximations. The loop in Figure 4(a) similarly has loop-carried dependencies and non-associative operations that are not parallelized by compilers. Figure 4(b) illustrates an unexpected approximation generated using 10,000 uniformly distributed random inputs ranging from -32k to 32k. In this case, PANDORA identifies that the loop is statistically very likely to converge towards an output of -1, which eliminates all operations for a speedup of 21x, with an error of 0.08%. However, using random inputs between 0 and 1 provides a significantly different approximation (Figure 4(c)) with speedup of 3.5x and an error of $1.2e^{-14}$%. These results suggest that designers could use PANDORA to create multiple approximations for different input values, similar to how function specialization optimizes a function for common inputs.

4 EXPERIMENTS

In this section, we present preliminary experiments demonstrating the capabilities of PANDORA. All experiments use the Python framework described in the previous section.

Figure 5 demonstrates PANDORA's ability to replace loop-carried dependencies—a long-time goal of compilers—with approximations that have independent (i.e., parallel) iterations. The figure shows tradeoffs between error and performance of multiple approximations, which we generated in PANDORA using different error constraints for each function. The evaluated examples are recurrence relations with dependencies between iterations, which shows that PANDORA is able to automatically find closed-form solutions or solutions without these dependencies. For example, PANDORA automatically discovers a finite impulse response (FIR) filter when using an infinite impulse response (IIR) as input. The included examples demonstrate three separate trends that we have also observed across other examples. Although some of these examples have known approximations or closed-form solutions, the key advantage of PANDORA is the automatic discovery of a closed-form solution or parallel approximation. This advantage is significant due to the lack of approximations for the vast majority of real functions, and due to the common use of application-specific approximations that are too unique for compiler support.

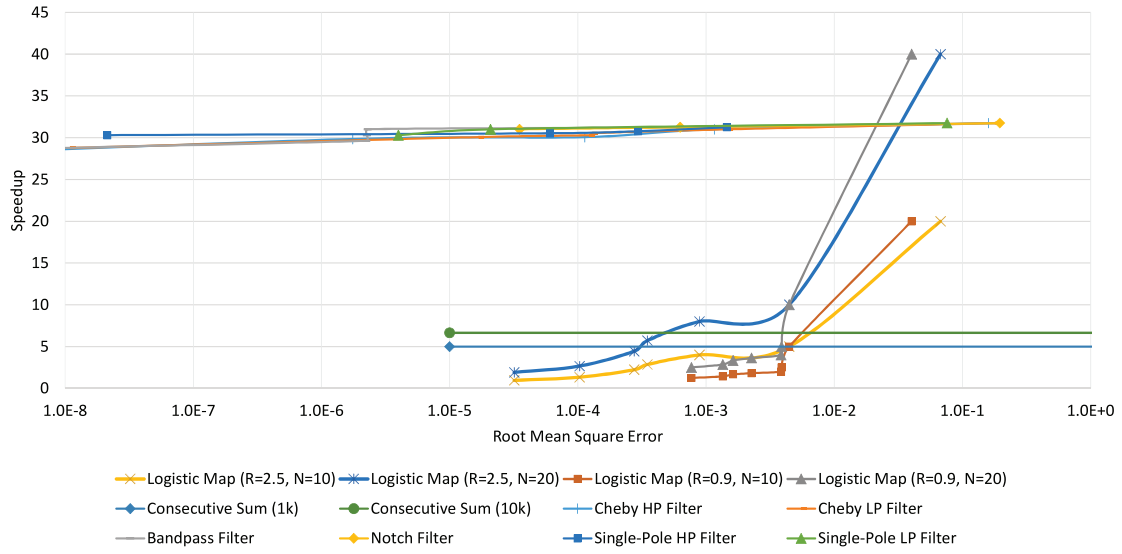

Fig. 5. Parallel approximation speedup (with constrained input bandwidth) and root mean square error of various recurrence relations. PANDORA discovered these approximations by eliminating loop-carried dependencies.

We created each example by writing corresponding Python code, which PANDORA then sampled to discover approximations. The experiments evaluate 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. The depth of the DFG represents the length of the longest dependence chain, which bounds execution time even with unlimited parallel resources. In general, a more parallel approximation tends to have a smaller depth than a sequential approximation, where loop-carried dependencies result in a long sequence of dependent iterations in the unrolled DFG. Although not all architectures will provide enough resources to achieve a performance equal to this bound, the comparison is applicable to many existing architectures for the selected examples. The figure specifies application-specific parameters (e.g., input sizes, constants) used in each experiments. To avoid arbitrarily large speedups from large inputs, the experiments evaluate a range of input sizes that illustrate the basic trends. Although approximation discovery times generally ranged from seconds to hours, we ran these experiments for days and then collected results of the best approximations found at that time.

To avoid the unrealistic possibility of infinite input bandwidth, the speedup in Figure 5 is based on an input-bandwidth constraint that matches existing PCIe bandwidth. The results demonstrate several trends. The first trend is that all of the filter examples achieved a speedup of around 30x with low error, which increases slightly as more error is allowed. The consecutive-sum examples demonstrate the second trend, where PANDORA achieved speedups of 5x and 7x for input sizes of 1k and 10k. Unlike the other trends, this speedup was independent of the error constraint because PANDORA found an exact closed-form solution that was simple enough to not benefit from approximation. The remaining examples demonstrate the third trend, where PANDORA found closed-form solutions that exhibited a rapid speedup increase from 1x to 40x for different error constraints. The examples in this third trend were more complex than the closed-form solutions in the second trend, which enabled a larger set of Pareto-optimal tradeoffs.

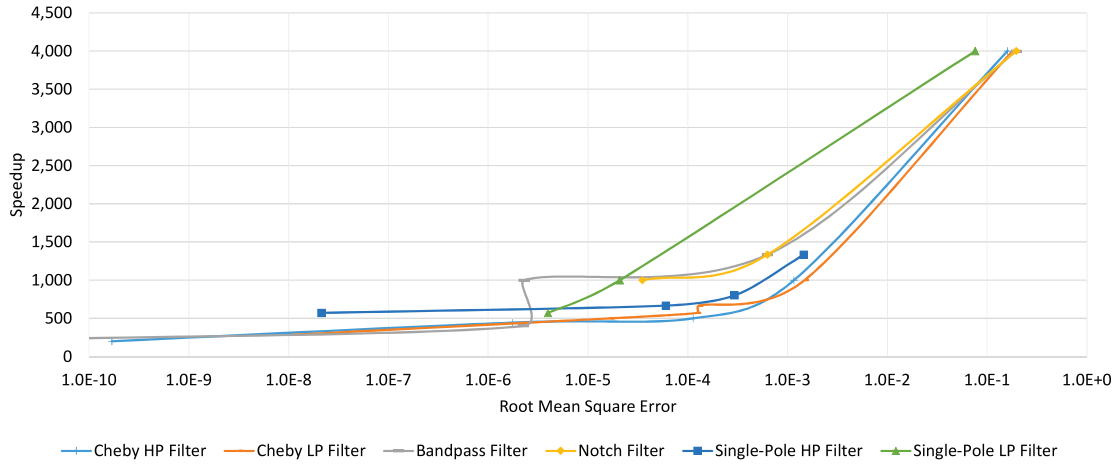

Fig. 6. Speedup from approximations of the filter examples in Figure 5 when not constrained by PCIe bandwidth.

The benefits of PANDORA are best highlighted by the logistic-map approximation in Figure 5. Whereas the other examples have known approximations or closed-form solutions, logistic map does not have a known closed-form solution. Despite the lack of a known solution, PANDORA found approximations with a root-mean-square error (RMSE) of less than $1e^{-4}$. Although not an exact solution, such error is likely to be acceptable for some cases. One interesting finding that highlights PANDORA's discovery capabilities is the counter-intuitive characteristics of the generated approximation. Despite the logistic map only using multiplication and subtraction, the discovered approximation uses a square root, cosine, and hyperbolic tangent, which is unlikely to be discovered by any programmer.

Figure 6 re-evaluates the filter examples for use cases that may not be as limited by input bandwidth (e.g., FPGA internal memory, large distributed systems with replicated data). The other examples are omitted because their results do not change with additional bandwidth. Two important differences can be seen in these experiments: (1) significantly higher speedups, ranging from 200x to 4000x; and (2) significant improvements from increasing error. Although not all architectures will be able to realize this amount of parallelism, these results highlight potential performance improvements.

Figure 7 complements these results with 336 randomly generated synthetic DFGs that represent unrolled loops with loop-carried dependencies. For these experiments, PANDORA generated an approximation based on a mean absolute percentage error constraint of 1%. To generate the synthetic loops, we created a script that produced corresponding DFGs that would be difficult to parallelize with existing compilers. Each DFG included random types of floating-point operations, 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. Across all examples, speedup and error averaged 9.5x and 0.3%, respectively. PANDORA was able to meet the error constraint for 93% of the examples, with many examples using orders-of-magnitude less error. For the examples where PANDORA couldn't meet the error constraint, the error ranged from 1% and 5.9%. The speedup of many examples was limited by the small DFG size. The ratio of speedup to total operations tended to increase for larger examples, which suggests that larger functions will likely experience significantly larger speedups.

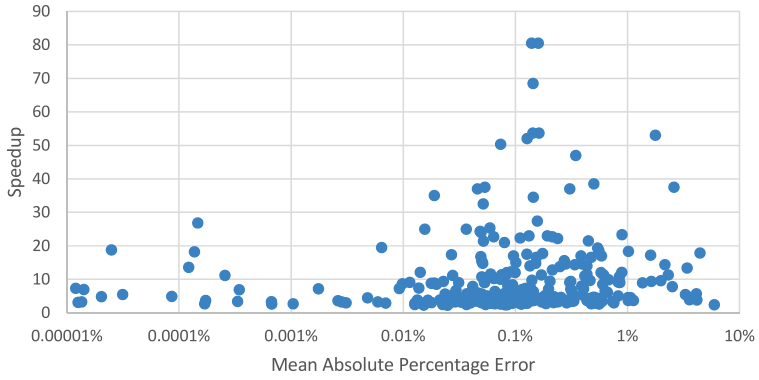

Fig. 7. Approximation tradeoffs between speedup and mean absolute percentage error (log scale) for 336 synthetic loops.

When repeating the tests with integer data, PANDORA discovered some approximations with no error. To our knowledge, no existing approach can automatically convert a series of non-associative operations into an exact parallel alternative. As a result, PANDORA is not solely limited to approximations, and can also be used to discover new error-free transformations.

Another interesting result was that PANDORA was able to generate a wide range of different approximations for the examples in Figure 7. Although many examples had increased parallelism, PANDORA occasionally removed iterations of a loop that had little effect on error, which is a known approximation strategy referred to as loop perforation [8, 46] that PANDORA discovered automatically. In most cases, PANDORA both reduced the total operations and parallelized those operations. For five examples, PANDORA enabled a parallel approximation by increasing the total number of operations. PANDORA also often automatically eliminated some of the inputs from the original application, which is a known machine-learning technique referred to as dimensionality reduction. We envision this capability being useful for eliminating communication bottlenecks and improving scalability in parallel systems.

Figure 8 demonstrates PANDORA's ability to create parallelizing approximations for a specific architecture, which for this experiment was an Arria 10 FPGA. In FPGAs, parallelism of an application is often limited by available resources. By discovering approximations that require fewer FPGA resources, PANDORA increases realizable parallelism. In these experiments, we compare achievable parallelism in terms of the number of IP instances that would fit in the FPGA between Intel-provided IP cores and automatically generated approximations. For these experiments, PANDORA generated approximations with speedups between 1x and 10x for mean-absolute percentage errors below 5%, with rapid increases in speedup to over 100x for larger errors. For the *ln* example, PANDORA generated an approximation that experienced a speedup of 757x at 18% error. Although the larger errors are unlikely to be acceptable, we included 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.

5 RELATED WORK

Existing approximate computing work focuses largely on languages, compilers, and/or optimization frameworks to improve performance/energy within designer-specified constraints. Rely [8] is a specialized language that enables

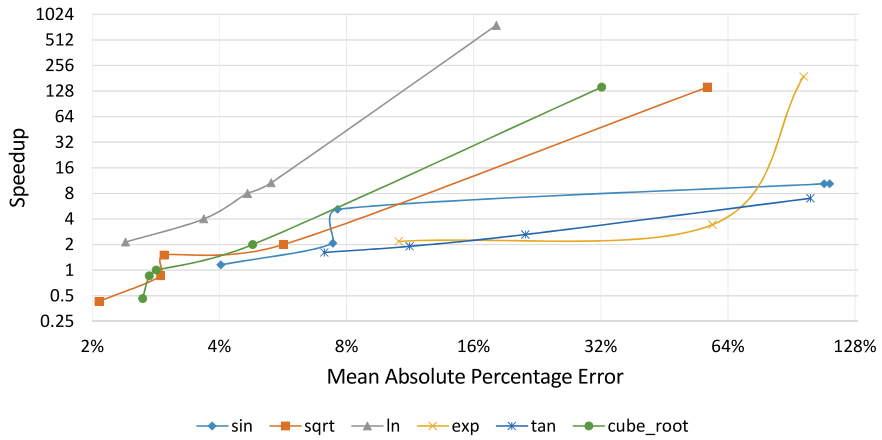

Fig. 8. FPGA speedup ($\log_2$ scale) and mean absolute percentage error of PANDORA-generated approximations that reduce resource utilization compared to Intel IP cores.

designers to specify accuracy and reliability (i.e., probability that a computation is correct) for different functions. When combined with a specification for approximate hardware resources, Rely enables designers to evaluate if the architecture can provide sufficient application reliability. Chisel [34] is a compiler-like framework that approximates a Rely program to maximize performance or energy while ensuring reliability and accuracy constraints. Green [3] is a similar approach that applies approximations to improve performance and energy while providing a specified quality of service. Our approach complements these works by significantly expanding the exploration for parallel approximations via genetic programming, while also removing the requirement for specification of approximation candidates and acceptability.

Quickstep [35] is a compiler-like framework with a similar goal as our proposed work: exploit approximation to create parallelism within error constraints. The key difference is that Quickstep introduces parallelism by transforming an existing program to allow relaxed synchronization and data races. By contrast, our approach generates a completely new algorithm via genetic programming, which explores a much larger space of parallelization options. Dubstep [36] is an extension of Quickstep that further increases parallelism via opportunistic synchronization and barriers. Renganarayana et al. [44] present a similar synchronization-relaxing methodology for reducing synchronization overhead in parallel programs. All of these prior studies are complementary to our proposed work, and could be used to increase parallelism by relaxing synchronization within our generated approximations. We plan to integrate the synchronization-relaxing approximation strategy into genetic programming as a mutation that would be considered with all other approximation options.

SNNAP [38], also conceptually similar to our proposed work, approximates an application with neural nets, which are executed on FPGAs. As opposed to using an approximation that is known to be efficient for a specific architecture, our proposed approach has the more general goal of generating a parallel approximation for potentially any architecture, including heterogeneous systems with different types of resources. Also, by performing genetic programming to create a completely new algorithm that may contain neural nets as an approximation strategy, we explore a larger parallelization space than SNNAP.

We previously published a preliminary version of PANDORA in [48], which we expand in this paper with significantly more detailed explanation of the approach and framework, example approximations that demonstrate different

capabilities and tradeoffs, an expanded evaluation in the experiments, a discussion of the envisioned usage of the framework with compilers and runtime optimizers, and a greatly expanded discussion of related work.

6 LIMITATIONS AND FUTURE WORK

Despite demonstrating considerable potential for approximation, there are several limitations and challenges that must be addressed to make PANDORA more widely usable. The most significant challenge is that PANDORA is based on symbolic regression, which is known to be a challenging problem where existing strategies generally only work for toy problems [32]. For future work, we plan to address this challenge in two ways. The first solution is hardware-accelerated symbolic regression that performs over a million times faster than existing software implementations, according to our preliminary results. Whereas existing software can generally only evaluate 100s to 1000s of solutions a second, a custom hardware pipeline can evaluate 100s of millions of solutions per second. Such performance enables fundamentally new exploration algorithms that we expect to overcome existing limitations. Our second planned solution is to consider providing PANDORA with information about the original code. Although this solution may restrict some of the counter-intuitive approximations that are currently discovered, it could also enable better solutions in situations where symbolic regression techniques cannot find an attractive approximation.

Another limitation is that even when existing symbolic-regression techniques achieve an effective approximation, the time required to discover that approximation can be a bottleneck. As a solution to this problem, PANDORA can be configured to return the best approximation seen so far after a specified amount of time. Alternatively, PANDORA can be used as a final optimization step, or to search for approximations in the background during application development.

In addition to speeding up symbolic regression, there is a need for better understanding effects of configuration parameters. For example, use of course-grained primitives significantly speeds up the search, but also restricts the size of the solution space, potentially resulting in less attractive approximation tradeoffs. Using finer-grained primitives enables a larger solution space, but that larger space also suffers from numerous local optima in which the search heuristic might get stuck. Similarly, there is a need to understand the tradeoffs between training time, approximation quality, population size, and number of generations.

For any use case where the training data has noise, symbolic regression is often limited by the tendency to overfit, which results in an overly complex equation that tries to account for the noise. When used for approximation, the training data has no noise because function outputs are deterministic, which helps to eliminate this problem. PANDORA can still potentially experience overfitting, but since most uses of PANDORA will specify a fitness function that minimizes the size of the discovered approximations, overfit solutions will tend to be eliminated during exploration.

Another area of future work will be automatically decomposing the input space into piece-wise approximations. In our experiments, we have noticed that in many cases PANDORA will provide good approximations within a restricted range, while providing less attractive tradeoffs when used to approximation a larger input space. We have currently dealt with this issue by manually creating multiple approximations, but that is only feasible in situations where the designer understands the original function. In many cases, especially with machine learning, the original function is unknown, which therefore requires an automated approach. As future work, we plan to investigate search heuristics that hierarchically decompose the input space into smaller subspaces as long as approximation quality continues to increase.

One fundamental limitation of PANDORA is that by relying on machine learning, PANDORA can only make probabilistic, as opposed to absolute, guarantees about approximation error. However, all machine-learning techniques share this same limitation, which has not prevented its widespread acceptance in various application domains.

One interesting limitation of PANDORA is the tendency to discover unintuitive approximations. If a designer needs to understand or debug the approximation when integrated with a larger application, the unintuitive nature of approximations could be prohibitive. However, similar to the probabilistic error issue, this limitation is shared by all machine-learning approaches, and has not limited the success of those approaches. Interestingly, the uninuitive nature of the approximations highlights one of the biggest advantages of PANDORA: non-obvious approximations are unlikely to be supported by a compiler or discovered by a designer.

7 CONCLUSIONS

We introduced the PANDORA framework for automatically discovering parallelizing approximations for potentially any targeted architecture. Whereas existing approximation approaches focus on languages and compilers that transform provided code into an approximation, PANDORA uses a symbolic-regression-based machine-learning strategy that discovers a new approximation based on sampled outputs of the original function, as opposed to specific coding constructs. By sampling function outputs, PANDORA explores the infinite number of alternative functions that coincide, or nearly coincide, with the samples of the original function in order to find an approximation that can be computed more efficiently. Envisioned usage of PANDORA includes generation of a range of Pareto-optimal approximations that can be used by a compiler or runtime optimizer to adapt the level of approximation to the current input, the current user's preferences, or to runtime conditions such as battery life.

In this paper, we demonstrated that PANDORA can remove loop-carried dependencies from recurrence relations, while also increasing parallelism in the presence of non-associative operations. We also showed how PANDORA can generate FPGA-specific approximations that reduce resources requirements for a number of FPGA functions, which achieved attractive tradeoffs between error and performance.

Although there are technical challenges preventing PANDORA from being widely usable in its current state, this paper presents a proof-of-concept that demonstrates attractive Pareto-optimal tradeoffs to the decades-long compiler challenge of parallelizing sequential code. This work also motivates the need for improvements in symbolic regression, which would in turn provide even more attractive approximations.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-1149285, CNS-1718033, and CCF-1909244.

REFERENCES

[1] C. Alvarez, J. Corbal, and M. Valero. 2005. Fuzzy memoization for floating-point multimedia applications. *Computers, IEEE Transactions on* 54, 7 (July 2005), 922–927. https://doi.org/10.1109/TC.2005.119

[2] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. 2011. Language and Compiler Support for Auto-tuning Variable-accuracy Algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, Washington, DC, USA, 85–96. http://dl.acm.org/citation.cfm?id=2190025.2190056

[3] 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

[4] M. Bohr. 2007. A 30 Year Retrospective on Dennard's MOSFET Scaling Paper. *Solid-State Circuits Society Newsletter, IEEE* 12, 1 (Winter 2007), 11–13. https://doi.org/10.1109/N-SSC.2007.4785534

[5] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. 1998. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput.* 24, 3-4 (1998), 421–444. https://doi.org/10.1016/S0167-8191(98)00020-9

[6] E. Bugnion, Shih-Wei Liao, B.R. Murphy, S.P. Amarasinghe, J.M. Anderson, M.W. Hall, and M.S Lam. 1996. Maximizing multiprocessor performance with the SUIF compiler. *Computer* 29, 12 (1996), 84,85,86,87,88,89. https://doi.org/10.1109/2.546613

[7] Simone Campanoni, Kevin Brownell, Svilen Kanev, Timothy M. Jones, Gu-Yeon Wei, and David Brooks. 2014. HELIX-RC: An Architecture-compiler Co-design for Automatic Parallelization of Irregular Programs. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 217–228. http://dl.acm.org/citation.cfm?id=2665671.2665705

[8] 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

[9] 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

[10] Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. 2006. 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)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 1110–1115. http://dl.acm.org/citation.cfm?id=1131481.1131790

[11] Lakshmi N.B. Chakrapani, Kirthi Krishna Muntimadugu, Avinash Lingamneni, Jason George, and Krishna V. Palem. 2008. Highly Energy and Performance Efficient Embedded Computing Through Approximately Correct Arithmetic: A Mathematical Foundation and Preliminary Experimental Validation. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '08)*. ACM, New York, NY, USA, 187–196. https://doi.org/10.1145/1450095.1450124

[12] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Analysis and Characterization of Inherent Application Resilience for Approximate Computing. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 113, 9 pages. https://doi.org/10.1145/2463209.2488873

[13] Noel Cressie. 1990. The origins of kriging. *Mathematical Geology* 22, 3 (1990), 239–252. https://doi.org/10.1007/BF00889887

[14] M. de la Guia Solaz and Richard Conway. 2010. Comparative study on Wordlength Reduction and Truncation for low power multipliers. In *MIPRO, 2010 Proceedings of the 33rd International Convention*. 84–88.

[15] A.E. Eichenberger, K. O'Brien, Peng Wu, Tong Chen, P.H. Oden, D.A. Prener, J.C. Shepherd, Byoungro So, Z. Sura, A. Wang, Tao Zhang, Peng Zhao, and M. Gschwind. 2005. Optimizing Compiler for the CELL Processor. In *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*. 161 – 172. https://doi.org/10.1109/PACT.2005.33

[16] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Architecture Support for Disciplined Approximate Programming. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII)*. ACM, New York, NY, USA, 301–312. https://doi.org/10.1145/2150976.2151008

[17] 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

[18] 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.

[19] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. 2012. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *FPGA '12: Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays (FPGA '12)*. ACM, New York, NY, USA, 47–56. https://doi.org/10.1145/2145694.2145704

[20] 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

[21] V. Gupta, D. Mohapatra, Sang Phill Park, A. Raghunathan, and K. Roy. 2011. IMPACT: IMPrecise adders for low-power approximate computing. In *Low Power Electronics and Design (ISLPED) 2011 International Symposium on*. 409–414. https://doi.org/10.1109/ISLPED.2011.5993675

[22] 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

[23] 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

[24] K. Hornik, M. Stinchcombe, and H. White. 1989. Multilayer Feedforward Networks Are Universal Approximators. *Neural Netw.* 2, 5 (July 1989), 359–366. https://doi.org/10.1016/0893-6080(89)90020-8

[25] A. Huang. 2015. Moore's Law is Dying (and that could be good). *Spectrum, IEEE* 52, 4 (April 2015), 43–47. https://doi.org/10.1109/MSPEC.2015.7065418

[26] P. M. W. Knijnenburg, T. Kisuki, and M. F. P. O'Boyle. 2002. *Iterative compilation.* Springer-Verlag New York, Inc., New York, NY, USA, 171–187.

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

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

[29] Sameer Kulkarni and John Cavazos. 2012. Mitigating the Compiler Optimization Phase-ordering Problem Using Machine Learning. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '12)*. ACM, New York, NY, USA,

147–162. https://doi.org/10.1145/2384616.2384628

[30] J. M. Pierre Langlois and Dhamin Al-Khalili. 2006. Carry-free approximate squaring functions with O(n) complexity and O(1) delay. *IEEE Trans. on Circuits and Systems* 53-II, 5 (2006), 374–378. http://dblp.uni-trier.de/db/journals/tcas/tcasII53.html#LangloisA06

[31] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. 2011. 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 XVI)*. ACM, New York, NY, USA, 213–224. https://doi.org/10.1145/1950365.1950391

[32] 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

[33] 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

[34] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. 2014. Chisel: Reliability- and Accuracy-aware Optimization of Approximate Computational Kernels. *SIGPLAN Not.* 49, 10 (Oct. 2014), 309–328. https://doi.org/10.1145/2714064.2660231

[35] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. 2013. Parallelizing Sequential Programs with Statistical Accuracy Tests. *ACM Trans. Embed. Comput. Syst.* 12, 2s, Article 88 (May 2013), 26 pages. https://doi.org/10.1145/2465787.2465790

[36] 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

[37] D. Mohapatra, V.K. Chippa, A. Raghunathan, and K. Roy. 2011. Design of voltage-scalable meta-functions for approximate computing. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. 1–6. https://doi.org/10.1109/DATE.2011.5763154

[38] 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

[39] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. 2010. Scalable Stochastic Processors. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '10)*. European Design and Automation Association, 3001 Leuven, Belgium, Belgium, 335–338. http://dl.acm.org/citation.cfm?id=1870926.1871008

[40] Preeti Ranjan Panda, Nikil D. Dutt, Alexandru Nicolau, Francky Catthoor, Arnout Vandecappelle, Erik Brockmeyer, Chidamber Kulkarni, and Eddy De Greef. 2001. Data Memory Organization and Optimizations in Application-Specific Systems. *IEEE Des. Test* 18, 3 (May 2001), 56–68. https://doi.org/10.1109/54.922803

[41] J. Park and I. W. Sandberg. 1991. Universal Approximation Using Radial-basis-function Networks. *Neural Comput.* 3, 2 (June 1991), 246–257. https://doi.org/10.1162/neco.1991.3.2.246

[42] Suganth Paul, Nikhil Jayakumar, and Sunil P. Khatri. 2009. A Fast Hardware Approach for Approximate, Efficient Logarithm and Antilogarithm Computations. *IEEE Trans. VLSI Syst.* 17, 2 (2009), 269–277. https://doi.org/10.1109/TVLSI.2008.2003481

[43] Efecan Poyraz, Heming Xu, and Yifeng Cui. 2014. Application-specific I/O Optimizations on Petascale Supercomputers. *Procedia Computer Science* 29 (2014), 910 – 923. https://doi.org/10.1016/j.procs.2014.05.082

[44] 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

[45] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. 2011. EnerJ: Approximate Data Types for Safe and General Low-power Computation. *SIGPLAN Not.* 46, 6 (June 2011), 164–174. https://doi.org/10.1145/1993316.1993518

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

[47] Huaiming Song, Yanlong Yin, Yong Chen, and Xian-He Sun. 2013. Cost-intelligent Application-specific Data Layout Optimization for Parallel File Systems. *Cluster Computing* 16, 2 (June 2013), 285–298. https://doi.org/10.1007/s10586-012-0200-4

[48] Greg Stitt and David Campbell. 2019. PANDORA: A Parallelizing Approximation-discovery Framework (WIP Paper). In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES 2019)*. ACM, New York, NY, USA, 198–202. https://doi.org/10.1145/3316482.3326345

[49] Vladimir Vapnik, Steven E. Golowich, and Alex Smola. 1996. Support Vector Method for Function Approximation, Regression Estimation, and Signal Processing. In *Advances in Neural Information Processing Systems 9*. MIT Press, 281–287.

[50] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. 2013. Quality Programmable Vector Processors for Approximate Computing. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2540708.2540710

[51] S. Venkataramani, A. Sabne, V. Kozhikkottu, K. Roy, and A. Raghunathan. 2012. SALSA: Systematic logic synthesis of approximate circuits. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*. 796–801.

[52] John Wernsing, Jeremy Fowers, and Greg Stitt. 2012. RACECAR: A Heuristic for Automatic Function Specialization on Multi-core Heterogeneous Systems. In *CASES'12: IEEE/ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*.

[53] K.E. Wires, M.J. Schulte, and J.E. Stine. 2000. Variable-correction truncated floating point multipliers. In *Signals, Systems and Computers, 2000. Conference Record of the Thirty-Fourth Asilomar Conference on*, Vol. 2. 1344–1348 vol.2. https://doi.org/10.1109/ACSSC.2000.911211

[54] Ning Zhu, Wang Ling Goh, Weija Zhang, Kiat Seng Yeo, and Zhi Hui Kong. 2010. Design of Low-power High-speed Truncation-error-tolerant Adder and Its Application in Digital Signal Processing. *IEEE Trans. Very Large Scale Integr. Syst.* 18, 8 (Aug. 2010), 1225–1229. https://doi.org/10.1109/TVLSI.2009.2020591

[55] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. 2012. Randomized Accuracy-aware Program Transformations for Efficient Approximate Computations. *SIGPLAN Not.* 47, 1 (Jan. 2012), 441–454. https://doi.org/10.1145/2103621.2103710