ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing

Adrian Sampson André Baixo Benjamin Ransford Thierry Moreau
Joshua Yip Luis Ceze Mark Oskin
University of Washington

Abstract

Approximate computing trades off accuracy for better performance and energy efficiency. It offers promising optimization opportunities for a wide variety of modern applications, from mobile vision to data analytics. Recent approaches to approximate computing have relied on either manual program modification, based exclusively on programmer reasoning, or opaque automatic transformations, which sacrifice programmer control. We describe ACCEPT, a comprehensive framework for approximation that balances automation with programmer guidance. It includes C/C++ type qualifiers for constraining approximation, a compiler analysis library that identifies regions of approximable code, an autotuning system that automatically chooses the best approximation strategies, and a feedback mechanism that explains how annotations can be improved for better approximation opportunities. ACCEPT automatically applies a variety of approximation techniques, including hardware acceleration, while ensuring their safety. We apply ACCEPT to nine workloads on a standard desktop, an FPGA-augmented mobile SoC, and an energy-harvesting sensor device to evaluate the annotation process. We observe average speedups of 2.3×, 4.8×, and 1.5× on the three platforms, respectively.

1. Introduction

Recent work on *approximate computing* has exploited the fact that many applications, particularly those whose outputs are meant for human interpretation, can enable more efficient execution at the cost of slightly inaccurate outputs. 3-D rendering, search, and machine learning are examples of the many tasks that tolerate inaccuracy.

Research over the past few years has proposed a variety of software and hardware approaches to approximate computing, including: producing multiple versions of a program with varying accuracy and choosing between them at run time [Baek and Chilimbi 2010; Hoffmann et al. 2011]; periodically skipping loop iterations [Sidiroglou-Douskos et al. 2011; Misailovic et al. 2010b]; removing synchronization to reduce contention [Misailovic et al. 2012, 2010a; Renganarayanan et al. 2012]; adjusting floating-point precision [Rubio-González et al. 2013]; using special low-power hardware structures that produce wrong results probabilistically [Esmaeilzadeh et al. 2012b; Liu et al. 2011]; and training hardware neural networks to mimic the behavior of costly functions [Esmaeilzadeh et al. 2012a; St. Amant et al. 2014].

These proposals share an important distinction from traditional program optimizations: they have subtle and broad-ranging effects on safety, reliability, and output quality. Some work relies on programmers for manual reasoning to control these effects [Esmaeilzadeh et al. 2012a; Liu et al. 2011; Renganarayanan et al. 2012; Sidiroglou-Douskos et al. 2011], while other work proposes automated transformation based on code patterns or exhaustive search [Baek and Chilimbi 2010; Samadi et al. 2014, 2013]. Manual code editing can be tedious and error-prone, especially since important safety invariants are at stake. Conversely, full automation eliminates a crucial element of visibility and control. Programmers must trust the automated system; they have no recourse when opportunities are missed or invariants are broken.

We propose ACCEPT (an Approximate C Compiler for Energy and Performance Trade-offs), a framework for approximation that balances automation with programmer guidance. ACCEPT is *controlled* because it preserves programmer intention expressed via code annotations. A static analysis rules out unintended side effects. The programmer participates in a feedback loop with the analysis to enable more approximation opportunities. ACCEPT is *practical* because it facilitates a range of approximation techniques that work on currently available hardware. Just as a traditional compiler framework provides common tools to support optimizations, ACCEPT's building blocks help implement automatic approximate transformations based on programmer guidance and dynamic feedback.

ACCEPT's architecture combines static and dynamic components. The frontend, built atop LLVM [Lattner and Adve 2004], extends the syntax of C and C++ to incorporate an `APPROX` keyword that programmers use to annotate types as in other work [Sampson et al. 2011]. ACCEPT's central analysis, *precise-purity*, identifies coarse-grained regions of code that can affect only approximable values. Coarse region

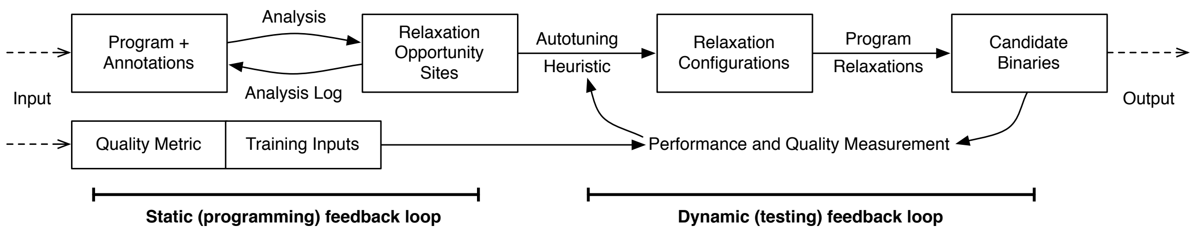
Analysis
Program + Annotations
Relaxation Opportunity Sites
Analysis Log
Autotuning
Heuristic
Relaxation Configurations
Program
Relaxations
Candidate Binaries
Input
Output
Quality Metric
Training Inputs
Performance and Quality Measurement
Static (programming) feedback loop
Dynamic (testing) feedback loop

Figure 1: Overview of the ACCEPT compiler workflow.

selection is crucial for safe approximation strategies: client optimizations use the results to transform code and offload to accelerators while preserving static safety properties. After compilation, an autotuning component measures program executions and uses heuristics to identify program variants that maximize performance and output quality. To incorporate application insight, ACCEPT furnishes programmers with feedback to guide them toward better annotations.

Contributions. ACCEPT is an end-to-end framework that makes past proposals for approximate program transformations practical and disciplined. Its contributions are:

- A programming model for program relaxation that combines lightweight annotations with compiler analysis feedback to guide programmers toward effective relaxations;
- An autotuning system that efficiently searches for a program's best approximation parameters;
- A core analysis library that identifies code that can be safely relaxed or offloaded to an approximate accelerator;
- A prototype implementation demonstrating both pure-software optimizations and hardware acceleration using an off-the-shelf FPGA part.

We evaluate ACCEPT across three platforms: a standard Intel-based server; a mobile SoC with an on-chip FPGA, which we use as an approximate accelerator; and an ultra-low-power, energy-harvesting embedded microcontroller where performance is critical to applications' viability. The experiments demonstrate average speedups of $2.3\times$, $4.8\times$, and $1.5\times$ on the three platforms, respectively, with quality loss under 10%.

We also report qualitatively on the programming experience. Novice C++ programmers were able to apply ACCEPT to legacy software to obtain new speedups. We plan to open-source the ACCEPT toolchain after publication both as a research tool for implementing new approximation techniques and as an end-to-end system for practitioners to experiment with approximate computing.

2. Overview

To safely and efficiently harness the potential of approximate programs, ACCEPT combines three main techniques: (1) a programmer–compiler feedback loop consisting of source code annotations and an analysis log; (2) a compiler analysis library that enables a range of automatic program relaxations; and (3) an autotuning system that uses dynamic measurements of candidate program relaxations to find the best balances between efficiency and quality. The final output is a set of Pareto-optimal versions of the input program that reflect its efficiency–quality trade-off space.

Figure 1 illustrates how these components make up ACCEPT's workflow. Two feedback loops control the impact of potentially destructive program relaxations: a *static* feedback loop providing conservative guarantees and a complementary *dynamic* feedback loop that measures real program behavior to choose the best optimizations. A key hypothesis of this paper is that neither static nor dynamic constraints are sufficient, since dynamic measurements cannot offer guarantees and static constraints do not capture the full complexity of relationships among relaxations, performance, and output quality. Together, however, the two feedback loops make ACCEPT's optimizations both controlled and practical.

Safety constraints and feedback. Because program relaxations can have outsize effects on program behavior, programmers need *visibility* into—and *control* over—the transformations the compiler applies. To give the programmer fine-grained control over relaxations, ACCEPT extends an existing lightweight annotation system for approximate computing based on type qualifiers [Sampson et al. 2011]. ACCEPT gives programmers visibility into the relaxation process via feedback that identifies which transformations can be applied and which annotations are constraining it. Through annotation and feedback, the programmer iterates toward an annotation set that unlocks new performance benefits while relying on an assurance that critical computations are unaffected.

Automatic program transformations. Based on programmer annotations, ACCEPT's compiler passes apply transformations that involve only approximate data. To this end, ACCEPT provides a common analysis library that identifies code regions that can be safely transformed. We bring ACCEPT's safety analysis, programmer feedback, and automatic site identification to existing work on approximate program transformations [Sidiroglou-Douskos et al. 2011; Renganarayanan et al. 2012; Rinard 2013; Misailovic et al. 2010a, 2012; Esmaeilzadeh et al. 2012a; St. Amant et al. 2014].

Autotuning. While a set of annotations may permit many different safe program relaxations, not all of them are beneficial. A practical system must help programmers choose from among many candidate relaxations for a given program to strike an optimal balance between performance and quality. ACCEPT's autotuner heuristically explores the space of possible relaxed programs to identify Pareto-optimal variants.

3. Annotation and Programmer Feedback

This section describes ACCEPT's annotations and feedback, which helps programmers balance safety with approximation. Rather than proving theoretical accuracy guarantees for restricted programming models as in other work [Misailovic et al. 2011; Zhu et al. 2012; Sampson et al. 2014], ACCEPT's workflow extends mainstream development practices: it combines lightweight safety guarantees, programmer insight, and testing to apply approximation to general code.

3.1 Annotation Language

The programmer uses annotations to communicate to the compiler which parts of a program are safe targets for program relaxation. ACCEPT adapts the type system of EnerJ, a language for approximate computing that was originally developed for bounding the effects of unreliable hardware components [Sampson et al. 2011]. As in EnerJ, all types in ACCEPT are qualified as either *approximate* or *precise*, with precise being the default.

Information flow and endorsement. ACCEPT's type system uses a standard information-flow approach to prevent approximate data from affecting precise data. Precise types are strict subtypes of their approximate counterparts, so variables annotated as approximate cannot be assigned to precise ones without explicit programmer intervention. (The opposite direction, wherein precise data affects approximate data, is permitted.) ACCEPT's type system is directly derived from EnerJ's [Sampson et al. 2011], for which its authors prove a noninterference property ensuring that precise data stays precise. The same noninterference guarantee applies to ACCEPT's type-qualifier extension for type-safe subsets of C and C++. Undefined behavior in C and C++ remains undefined in ACCEPT: programs that violate type safety can also violate ACCEPT's guarantees.

Strict information flow can be too constraining. For example, a program may need to compute an approximate value, where relaxations can apply, but then check the resulting output for integrity while treating it as precise:

```
APPROX int a = expensiveCall();
cheapChecksumPrecise(a); // illegal
```

To permit this pattern, ACCEPT provides an *endorsement* expression that acts as a cast from an approximate type to its precise equivalent. The above program fails to typecheck, but using `ENDORSE(a)` as the argument is legal:

```
APPROX int a = expensiveCall();
cheapChecksumPrecise(ENDORSE(a));
```

Endorsements give programmers explicit control over information flow when dealing with approximate values.

Pointer types. For basic, non-reference types, ACCEPT's dialect of C allows unidirectional information flow: precise values can be assigned into approximate variables but not vice-versa. For pointers and references, however, even precise-to-approximate flow is unsound since it creates aliases for the same data that disagree on its type. Pointer types are therefore invariant in the referent type. The language does not permit approximate pointers—i.e., addresses must be precise.

Implicit flow. Control flow provides an avenue for approximate data to affect precise data without a direct assignment. For example, `if (a) p = 5;` allows the variable `a` to affect the value of `p`. Like EnerJ, ACCEPT prohibits approximate values from being used in conditions—specifically, in `if`, `for`, `do`, `while`, and `switch` statements and in the ternary conditional-expression operator. Programmers can use endorsements to explicitly circumvent this restriction.

Escape hatches. ACCEPT decides whether program relaxations are safe based on the *effects* of the statements involved. Section 4 goes into more detail, but at a high level, code can be relaxed if its externally visible effects are approximate. For example, if `a` is a pointer to an `APPROX int`, then the statement `*a = 5;` has an approximate effect on the heap. Escape hatches from this sound reasoning are critical in a practical system that must handle legacy code. To enable or disable specific optimizations, the programmer can override the compiler's decision about a statement's effects using two annotations. The `ACCEPT_PERMIT` annotation forces a statement to be considered approximate and `ACCEPT_FORBID` forces it to be precise, forbidding any relaxations involving it.

These two annotations represent escape hatches from ACCEPT's normal reasoning and thus violate the safety guarantees it normally provides. Qualitatively, when annotating programs, we use these annotations much less frequently than the primary annotations `APPROX` and `ENDORSE`. We find `ACCEPT_PERMIT` to be useful when experimentally exploring program behavior before annotating and in system programming involving memory-mapped registers. Conversely, `ACCEPT_FORBID` is useful for marking parts of the program involved in introspection. Section 7.4 gives more detail on these experiences.

3.2 Programmer Feedback

ACCEPT takes inspiration from parallelizing compilers that use a development feedback loop to help guide the programmer toward parallelization opportunities [Ringenburg and Choi 2009; Hiranandani et al. 1994]. It provides feedback through an *analysis log* that describes the relaxations that it attempted to apply. For example, for ACCEPT's synchronization-elision relaxation, the log lists every lexically scoped lock acquire/release pair in the program. For each relaxation opportunity, it reports whether the relaxation is safe—whether it involves only approximate data—and, if it is not, identifies the statements that prevent the relaxation from applying. We call these statements with externally visible precise effects *blockers*.

ACCEPT reports blockers for each failed relaxation-opportunity site. For example, during the annotation of one program in our evaluation, ACCEPT examined this loop:

```
650  double myhiz = 0;
651  for (long kk=k1; kk<k2; kk++) {
652    myhiz += dist(points->p[kk], points->p[0],
653      ptDimension) * points->p[kk].weight;
654  }
```

The store to the precise (by default) variable `myhiz` prevents the loop from being approximable. The analysis log reports:

```
loop at streamcluster.cpp:651
blockers: 1
 * streamcluster.cpp:652: store to myhiz
```

Examining that loop in context, we found that `myhiz` was a weight accumulator that had little impact on the algorithm, so we changed its type from `double` to `APPROX double`. On its next execution, ACCEPT logged the following message about the same loop, highlighting a new relaxation opportunity:

```
loop at streamcluster.cpp:651
can perforate loop
```

The feedback loop between the programmer's annotations and the compiler's analysis log strikes a balance with respect to programmer involvement: it helps identify new relaxation opportunities while leaving the programmer in control. Consider the alternatives on either end of the programmer-effort spectrum: On one extreme, suppose that a programmer wishes to speed up a loop by manually skipping iterations. The programmer can easily misunderstand the loop's side effects if it indirectly makes system calls or touches shared data. On the other extreme, unconstrained automatic transformations are even more error prone: a tool that removes locks can easily create subtle concurrency bugs. Combining programmer feedback with compiler assistance balances the advantages of these approaches.

4. Analysis and Relaxations

ACCEPT takes an annotated program and applies a set of program transformations to code that affects only data marked approximate. We call these transformations *relaxations* because they trade correctness for performance. To determine relaxation opportunities from type annotations, ACCEPT uses a an analysis called *precise-purity*. This section describes ACCEPT's implementations of several program relaxations drawn from the literature and how precise-purity analysis makes them safe. As a framework for approximation, ACCEPT is extensible to relaxations beyond those we describe here.

4.1 Precise-Purity Analysis

ACCEPT provides a core program analysis that client optimizations use to verify that program relaxations are appropriately constrained. This analysis must reconcile a fundamental difference between the language's safety guarantees and the transformation mechanisms: the programmer specifies safety in terms of fine-grained annotations on individual data elements, but program relaxations affect coarse-grained regions of code such as loop bodies or entire functions. Rather than resort to opaque and error-prone code-centric annotation, ACCEPT bridges this gap by analyzing the side effects of coarse-grained code regions.

ACCEPT's analysis library determines, for a region of interest (e.g., a loop body), whether its side effects are exclusively approximate or may include precise data—in our terminology, whether or not the region is *precise pure*. Precise-purity is the key criterion for whether a relaxation can apply. In ACCEPT, every relaxation strategy consults the precise-purity analysis results and may only apply if and only if it affects exclusively precise-pure code. A region is precise pure if it:

- contains no stores to precise variables that may be read outside the region;
- does not call any functions that are not precise pure; and
- does not include an unbalanced synchronization statement (locking without unlocking or vice versa).

The analysis begins with the conservative assumption that the region is not precise pure and asserts otherwise only if it can prove precise-purity. For example, this code:

```
int p = ...;
APPROX int a = p * 2;
```

is precise pure if and only if the variable `p` is never read outside this code region. External code may, however, read the variable `a` since it is marked as approximate. Together with the information-flow type system, the precise-purity restriction ensures that code transformations only influence approximate data. Since only the approximate value `a` escapes the precise-pure block above, dependent code must also be marked as `APPROX` to obey the typing rules: any code that treats `a` as precise is a type error. The block can affect only other approximate computations, so it is safe for relaxations to transform it.

We implement the core precise-purity analysis conservatively using SSA definition–use chains and a simple pointer-escape analysis. Section 6 gives more implementation details.

4.2 Approximate Region Selection

To support accelerator-style program transformations that provide replacements for coarse-grained regions of approximate code, ACCEPT can enumerate a function's replaceable approximate code regions. A *candidate region* is a set of instructions that is precise pure, forms control flow with a single entry and a single exit, and has identifiable live-ins and live-outs. Client optimizations, such as the neural acceleration described in Section 4.3.3, can enumerate the candidate regions in a program to attempt optimization. Precise-purity

Algorithm 1: Approximate region selection.

```
Input: function $f$
Output: set of precise-pure regions $R$ in $f$
1  foreach basic block $B$ in $f$ do
2      foreach block $B'$ strictly post-dominated by $B$ do
3          if $B'$ dominates $B$ then
4              $region \leftarrow$ formRegionBetween$(B', B)$
5              if $region$ is precise-pure then
6                  $R \leftarrow R \cup \{region\}$
7              end
8          end
9      end
10 end
```

analysis enables region selection by proving that chunks of code are cleanly separable from the rest of the program.

Region selection meets the needs of accelerators that do not access memory directly and therefore require statically identifiable inputs and outputs; patterns such as dynamic array updates cannot be offloaded. The same analysis can be adapted to superoptimizers and synthesizers that need to operate on delimited subcomputations. For example, an accuracy-aware superoptimizer such as STOKE [Schkufza et al. 2014] could use ACCEPT's region selection to search for tractable optimization targets in a large program. Each fragment could be optimized independently and spliced back into the program.

Algorithm 1 shows how ACCEPT enumerates candidate regions. The algorithm uses dominance and post-dominance sets to identify pairs of basic blocks B_1 and B_2 where B_1 dominates B_2 and B_2 post-dominates B_1. The portion of the control-flow graph between these pairs represent all the single-entry, single-exit portions of a function. For a function with n blocks, the enumeration needs n^2 precise-purity checks in the worst case—but typically fewer because the LLVM compiler infrastructure pre-computes the dominator and post-dominator trees.

4.3 Safe Approximate Relaxations

To demonstrate ACCEPT's flexibility as a framework, we implement three approximation strategies from the literature using precise-purity analysis.

4.3.1 Loop Perforation

Sidiroglou et al. propose *loop perforation*, which exploits the fact that many programs tolerate some skipping of loop iterations without significant quality degradation [Sidiroglou-Douskos et al. 2011]. A perforated loop includes a parameter, the *perforation factor*, that governs how often an iteration can be skipped at run time.

ACCEPT considers a loop safe to perforate if its body is precise pure and free of early exits (i.e., `break` statements), which can cause nontermination if skipped. To perforate a loop, ACCEPT inserts a counter and code to increment and check it in each loop iteration. To minimize the overhead of loop perforation, ACCEPT requires the perforation factor p to be a power of two to enable bitwise tests against the counter. The loop body executes once every p iterations.

4.3.2 Synchronization Elision

In parallel programs, inter-thread synchronization constructs—locks, barriers, semaphores, etc.—are necessary for program predictability but threaten scalability. Recent research has proposed to strategically reduce synchronization in approximate programs [Rinard 2013; Misailovic et al. 2010a, 2012; Renganarayanan et al. 2012]. Even though removing synchronization can add data races and other nondeterminism to previously race-free or deterministic programs, this recent work has observed that the "incorrectness" is often benign: the resulting lost updates and atomicity violations can sometimes only slightly change the program's output.

ACCEPT can elide calls to locks (mutexes) and barriers from the pthreads library. To permit the elision of a lock acquire–release pair, ACCEPT requires that the critical section—the code between the acquire and release—be precise pure. To elide `pthread_barrier_wait ()` synchronization, ACCEPT looks for pairs of calls whose intervening code is precise pure, in such cases removing the *first* call (the second call remains to delimit the end of the region).

4.3.3 Neural Acceleration

Recent work has shown how to accelerate approximate programs with hardware neural networks [Chen et al. 2012; Temam 2012; Belhadj et al. 2013]. *Neural acceleration* uses profiled inputs and outputs from a region of code to train a neural network that mimics the code. The original code is then replaced with an invocation of an efficient hardware accelerator implementation, the Neural Processing Unit (NPU) [Esmaeilzadeh et al. 2012a; St. Amant et al. 2014; Moreau et al. 2015]. But the technique has thus far required manual identification of candidate code regions and insertion of offloading instructions. ACCEPT automates the process.

ACCEPT implements an automatic neural acceleration transform that uses an existing configurable neural-network implementation for an on-chip field-programmable gate array (FPGA) [Moreau et al. 2015]. ACCEPT uses approximate region selection (§4.2) to identify acceleration targets, then trains a neural network on execution logs for each region. It then generates code to offload executions of the identified region to the accelerator. The offload code hides invocation latency by constructing batched invocations that exploit the high-bandwidth interface between the CPU and FPGA. We target a commercially available FPGA-augmented system on a chip (SoC) and do not require specialized neural hardware.

4.3.4 Other Client Relaxations

The three optimizations above demonstrate ACCEPT's breadth as a framework for realizing ideas from approximate-computing research. Though we omit details for space, we have also prototyped two other optimizations using ACCEPT: an approximate alias analysis that unlocks secondary compiler optimizations such as loop-invariant code motion and vectorization for approximate data, and approximate strength reduction that aggressively replaces expensive arithmetic operations with cheaper shifts and masks that are not exactly equivalent. Other optimizations from the literature are also amenable to ACCEPT's architecture, including approximate parallelization [Misailovic et al. 2010a], float-to-fixed conversion [Aamodt and Chow 2008], bit-width reduction [Tong et al. 2000; Rubio-González et al. 2013], GPU pattern replacement [Samadi et al. 2014], and alternate-algorithm selection [Baek and Chilimbi 2010; Ansel et al. 2009].

5. Autotuning Search

The autotuner is a test harness in which ACCEPT explores the space of possible program relaxations through empirical feedback. We call a particular selection of relaxations and associated parameters (e.g., loop perforation with factor p) a *relaxation configuration*. The autotuner heuristically generates relaxation configurations and identifies the ones that best balance performance and output quality. The programmer also provides multiple inputs to the program. ACCEPT validates relaxation configurations by running them on fresh inputs to avoid overfitting.

Because the definition of quality is application dependent, ACCEPT relies on programmer-provided *quality metrics* that measure output accuracy, as in previous work [Sampson et al. 2011; Esmaeilzadeh et al. 2012b; Misailovic et al. 2010b; Carbin et al. 2012; Baek and Chilimbi 2010; Esmaeilzadeh et al. 2012a]. The quality metric is another program that (1) reads the outputs from two different executions of the program being transformed and (2) produces an error score between 0.0 (outputs are identical) and 1.0 (outputs are completely different), where the definitions of "identical" and "different" are application dependent.

A naïve method of exploring the space of relaxation configurations is to enumerate all possible configurations. But the space of possible relaxation configurations is exponential in the number of relaxation opportunities and therefore infeasible to even enumerate, let alone evaluate empirically. We instead use a heuristic that prioritizes a limited number of executions that are likely to meet a minimum output quality.

ACCEPT's heuristic configuration search consists of two steps: it vets each relaxation opportunity individually and then composes relaxations to create composites.

Vetting individual relaxations. In the first step, the autotuner separately evaluates each relaxation opportunity ACCEPT's analysis identified. Even with ACCEPT's static constraints, it is possible for some relaxations to lead to unacceptably degraded output or zero performance benefit. When the programmer uses escape hatches such as `ENDORSE` incorrectly, approximation can affect control flow or even pointers and hence lead to crashes. ACCEPT vets each relaxation opportunity to disqualify unviable or unprofitable ones.

For each relaxation opportunity, the autotuner executes the program with only that relaxation enabled. If the output error is above a threshold, the running time averaged over several executions is slower than the baseline, or the program crashes, the relaxation is discarded. Then, among the surviving relaxations, the autotuner increases the aggressiveness of any optimizations that have parameters. (In our prototype, only loop perforation has a variable parameter: the perforation factor p.) The autotuner records the range of parameters for which each opportunity site is "good"—when its error is below a threshold and it offers speedup over the original program—along with the running time and quality score. These parameters are used in the next step to create composite configurations.

Composite configurations. After evaluating each relaxation opportunity site individually, ACCEPT's autotuner composes multiple relaxations to produce the best overall program configurations. For a program of even moderate size, it is infeasible to try every possible combination of component relaxations. ACCEPT heuristically predicts which combinations will yield the best performance for a given quality constraint and validates only the best predictions experimentally.

To formulate a heuristic, ACCEPT hypothesizes that relaxations compose linearly. That is, we assume that two program relaxations that yield output error rates e_1 and e_2, when applied simultaneously, result in an error of $e_1 + e_2$ (and that performance will compose similarly). Different relaxations can in practice compose unpredictably, but this simplifying assumption is a tractable approximation that ACCEPT later validates with real executions.

The configuration-search problem is equivalent to the 0/1 Knapsack Problem. In the Knapsack formulation, each configuration's output error is its *weight* and its performance benefit $1 - \frac{1}{\text{speedup}}$ is its *value*. The goal is to find the configuration that provides the most total value subject to a maximum weight capacity.

The Knapsack Problem is NP-complete and intractable even for programs with only a few dozen potential relaxations. Instead, ACCEPT uses a well-known approximation algorithm [Dantzig 1957] to sort the configurations by their value-to-weight ratio and greedily selects configurations in rank order up to an error budget. To account for our simplifying assumptions, we use a range of error budgets to produce multiple candidate composites. The algorithm is dominated by the sorting step, so its running time is $O(n \log n)$ in the number of vetted relaxation-opportunity sites (and negligible in practice). Like other candidate configurations, the composites are executed repeatedly to measure their true output quality and speedup.

6. Implementation

ACCEPT extends the LLVM compiler infrastructure [Lattner and Adve 2004] and has three main components: (1) a modified compiler frontend based on Clang [Clang] that augments C and C++ with an approximation-aware type system; (2) a program analysis and set of LLVM optimization passes that implement program relaxations; and (3) a feedback and autotuning system that automatically explores quality–efficiency trade-offs.

6.1 Type System

We implemented our approximation-aware type system, along with the syntactic constructs `APPROX` and `ENDORSE`, as an extension to the Clang C/C++ compiler.

Pluggable types layer. We modified Clang to support *pluggable types* in the style of Cqual [Foster 2002] and Java's JSR-308 with its accompanying Checker Framework [Ernst; Papi et al. 2008]. Pluggable types allow a compiler's built-in type system to be overlaid with arbitrary qualifiers and typing rules. Syntactically, we provide a GNU C `__attribute__(())` construct that specifies the type qualifiers for any variable, field, parameter, function, or method definition. Our pluggable type library implements a bottom-up AST traversal with an interface for defining typing rules. Finally, the compiler emits LLVM IR bitcode augmented with per-instruction metadata indicating the qualifiers on the value of each SSA operation. For example, when the result of the expression `a + b` has the type `APPROX float`, it emits an `add` instruction reflecting the qualifier. This representation allows LLVM's compiler passes, which have access only to the IR and not to the AST, to use the programmer-provided qualifier information. (We plan to release the source code for the generic pluggable types layer along with the rest of the system.)

Approximation-aware type system. The primary constructs in our EnerJ-inspired, approximation-aware type system are the `APPROX` type qualifier and the `ENDORSE` explicit type conversion. Both are provided as macros in a C header file. The `APPROX` macro expands to an `__attribute__(())` construct, and `ENDORSE(e)` expands to an opaque C comma expression with a magic number that the checker recognizes and interprets as a cast. The type checker itself follows a standard information-flow implementation: most expressions are approximate if any of their subexpressions is approximate; ACCEPT checks types and emits errors in assignments, function calls, function returns, and conditionals.

The escape hatches `ACCEPT_PERMIT` and `ACCEPT_FORBID` are parsed from C-style comments.

6.2 Analysis and Relaxations

Precise-purity (§4.1) and region selection (§4.2) are implemented as LLVM analysis passes. The ACCEPT prototype includes three relaxations, also LLVM passes, that consume the analysis results. The precise-purity analysis offers methods that check whether an individual LLVM IR instruction is approximate, whether an instruction points to approximate memory, and whether a code region (function or set of basic blocks) is precise pure. The region-selection analysis offers methods to enumerate precise-pure regions of a function that can be treated specially, e.g., offloaded to an accelerator.

We special-case the C memory-management intrinsics `memcpy` and `memset` to assign them appropriate effects. For example, `memset(p,v,n)` where `p` has type `APPROX float *` is considered precise pure because it behaves as a store to `p`.

The loop-perforation and synchronization-elision relaxations (§4) use precise-purity analysis to determine whether a loop body or critical section can be considered approximate. Loop perforation generates a counter and mask to skip iterations; and synchronization elision deletes lock and barrier call instructions. Neural acceleration uses region selection to identify target code and subsequently generates inline ARM assembly to buffer data and communicate with the FPGA over a coherent bus.

6.3 Autotuning

ACCEPT's autotuning system is implemented separately from the compiler component. It communicates with the compiler via command-line flags and a pass-generated configuration file that enumerates the program's relaxation opportunities.

The programmer provides a quality metric to the autotuner in the form of a Python script that defines a `score` function, which takes as input two execution outputs and produces an error value between 0.0 and 1.0.

The autotuner's heuristic search consists of many independent program executions, so it is embarrassingly parallel. ACCEPT optionally distributes the work across a cluster of machines to accelerate the process. Workers on each cluster node receive a configuration, compile the program, execute it, and return the output and timing statistics. The master node coordinates the search and reports results.

6.4 Neural Acceleration

We evaluate ACCEPT's approximate region selection using a Neural Processing Unit (NPU) accelerator implemented on an on-chip FPGA (§4.3.3). The design is based on recent work that implements an NPU based on systolic arrays [Esmaeilzadeh et al. 2012a; Moreau et al. 2015].

7. Evaluation

We evaluated ACCEPT's effectiveness at helping programmers to tune programs. We collected applications from domains known to be resilient to approximation, annotated each program using ACCEPT's feedback mechanisms, and applied the autotuner to produce relaxed executables. We examined applications targeting three platforms: a standard x86 server system, a mobile SoC augmented with an FPGA for neural acceleration, and a low-power, embedded sensing device.

Application	Description	Quality Metric	LOC	`APPROX`	`ENDORSE`
canneal	VLSI routing	Routing cost	3144	91	8
fluidanimate	Fluid dynamics	Particle distance	2138	30	47
streamcluster	Online clustering	Cluster center distance	1122	51	24
x264	Video encoding	Structural similarity	22018	300	69
sobel	Sobel filter	Mean pixel difference	154	7	5
zynq-blackscholes	Investment pricing	Mean relative error	318	50	10
zynq-inversek2j	Inverse kinematics	Euclidean distance	67	6	6
zynq-sobel	Sobel filter	Mean pixel difference	356	16	7
msp430-activity	Activity recognition	Classification rate	587	19	5

Table 1: The approximate applications used in our evaluation. The final two columns show source code annotation counts.

7.1 Applications

Table 1 lists the applications we use in this evaluation. Since there is no standard suite of benchmarks for evaluating approximate-computing systems, we collect approximable applications from multiple sources, following the lead of other work in the area [Sampson et al. 2011; Esmaeilzadeh et al. 2012a; Chen et al. 2012; Misailovic et al. 2010b; Temam 2012]. Five programs—canneal, fluidanimate, streamcluster, x264, and zynq-blackscholes—are from the PARSEC parallel benchmark suite [Bienia 2011]. They implement physical simulation, machine learning, video, and financial algorithms. Another program, sobel along with its ARM port zynq-sobel, is an image convolution kernel implementing the Sobel filter, a common component of image processing pipelines. The final program, msp430-activity, is an activity-recognition workload that uses a naïve Bayesian classifier to infer a physical activity from a sequence of accelerometer values on an MSP430 microcontroller [Texas Instruments, Inc.].

To evaluate the applications' output accuracy, we develop application-specific quality metrics as in prior work on approximate computing [Sampson et al. 2011; Baek and Chilimbi 2010; Misailovic et al. 2010b; Esmaeilzadeh et al. 2012a,b]. Table 1 lists the metric for each program. In one case, fluidanimate, the benchmark shipped with an output-comparison tool.

We annotated each benchmark by inserting type annotations and interacting with the compiler's feedback mechanisms to identify fruitful optimizations. Table 1 shows the source code annotation density. Section 7.4 reports qualitatively on our experiences with the annotation process.

To validate the generality of ACCEPT's program relaxations, we used one set of inputs (the *training set*) during autotuning and a distinct input set (the *testing set*) to evaluate the final speedup and quality loss.

7.2 Experimental Setup

Each application targets one of three evaluation platforms: an x86 server, an ARM SoC with an integrated FPGA, and an embedded sensing system. The server platform is a dual-socket, 64-bit, 2.8 GHz Intel Xeon machine with two-way simultaneous multithreading and 4 GB memory. During autotuning, we distributed work across a cluster of 20 of these Xeon machines running Red Hat Enterprise Linux 6.5 with kernel version 2.6.32. The FPGA-augmented SoC is included to demonstrate the NPU relaxation, which requires programmable logic. We implemented the neural-network accelerator (§6.4) on a Xilinx Zynq-7020 part, which includes a dual-core ARM Cortex-A9 and an FPGA fabric on a single TSMC 28 nm die. Full details on the accelerator implementation can be found in [Moreau et al. 2015]. Finally, for the embedded msp430-activity workload, we used the WISP [Sample et al. 2008] device depicted in Figure 2. The WISP incorporates a prototype MSP430FR5969 "Wolverine" microcontroller with 2 KB of SRAM and 64 KB of nonvolatile ferroelectric RAM (FRAM) along with an onboard accelerometer. The WISP can harvest energy from radio waves, but we powered it via its JTAG interface to ensure reliable, repeatable runs connected to our test harness.

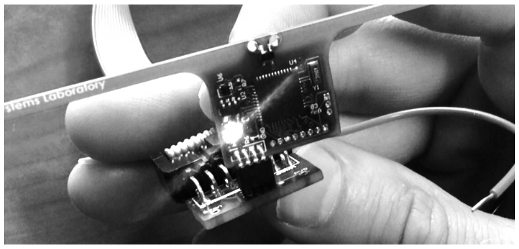
Figure 2: WISP sensing platform [Sample et al. 2008].

We compiled all applications with LLVM's aggressive `-O2` optimizations in addition to ACCEPT's program relaxations. We measured performance by reading the system clock before and after a region of interest that excluded the loading of data files from disk and dumping of results. (This region of interest was already defined for the PARSEC benchmarks.) To obtain accurate time measurements, we ran each configuration five times and averaged the running times.

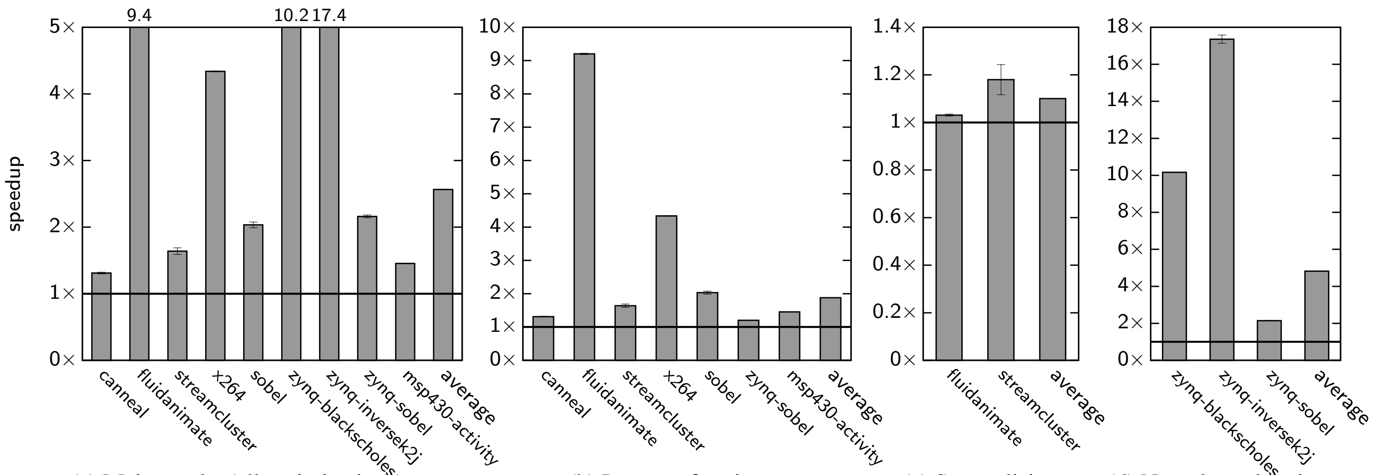

Figure 3: Speedup for each application, including all optimizations (a) and each optimization in isolation (b–d).

Application	Sites	Composites	Total	Optimal	Error	Speedup
canneal	5	7	32	11	1.5–15.3%	1.1–1.7×
fluidanimate	20	13	82	11	<0.1%	1.0–9.4×
streamcluster	23	14	66	7	<0.1–12.8%	1.0–1.9×
x264	23	10	94	3	<0.1–0.8%	1.0–4.3×
sobel	6	5	21	7	<0.1–26.7%	1.1–2.0×
zynq-blackscholes	2	1	5	1	4.3%	10.2×
zynq-inversek2j	3	2	10	1	8.9%	17.4×
zynq-sobel	6	2	27	4	2.2–6.2%	1.1–2.2×
msp430-activity	4	3	15	5	<0.1%	1.5×

Table 2: Tuning statistics and resulting optimal configurations for each benchmark.

7.3 Results

Figure 3a plots the speedup (versus precise execution) of the best-performing relaxed versions that ACCEPT found for each application with output error under 10%. Speedups in the figure range from 1.3× (canneal) to 17.4× (zynq-inversek2j) with a harmonic mean of 2.3× across all three platforms.

Figure 3 shows the speedup for relaxed versions with only one type of optimization enabled. Not every optimization applies to every benchmark: notably, neural acceleration applies only to the Zynq benchmarks, and synchronization elision applies only to the two benchmarks that use fine-grained lock- and barrier-based synchronization. Loop perforation is the most general relaxation strategy and achieves a 1.9× average speedup across 7 of the benchmarks. Synchronization elision applies to fluidanimate and streamcluster, for which it offers speedups of 3% and 1.2× respectively. The optimization reduces lock contention, which does not dominate the running time of these benchmarks. Neural acceleration offers the largest speedups, ranging from 2.1× for zynq-sobel to 17.4× for zynq-inversek2j.

ACCEPT's feedback system explores a two-dimensional trade-off space between output quality and performance. For each benchmark, ACCEPT reports Pareto-optimal configurations rather than a single "best" relaxed executable; the programmer can select the configuration that strikes the best quality–performance balance for a particular deployment. Table 2 shows the range of output error rates and speedups in the frontiers for our benchmarks. We highlight canneal as an example. For this program, ACCEPT identifies 11 configurations with output error ranging from 1.5% to 15.3% and speedup ranging from 1.1× to 1.7×. Using this Pareto frontier output, the developer can choose a configuration with a lower speedup in error-sensitive situations or a more aggressive 1.7× speedup if higher error is considered acceptable for a deployment.

One benchmark, fluidanimate, exhibits especially low error even under aggressive optimization; the configuration with the best speedup, which removed two locks and perforated nine loops, had overall error (change in final particle positions) under 0.00001%. For msp430-activity, error remained at 0% in all acceptable configurations.

Autotuner characterization. Table 2 shows the number of relaxation opportunities (labeled *sites*), the number of composite configurations considered, the total number of configurations explored (including parameter-tuning configurations), and the number of optimal configurations on the output Pareto frontier for each benchmark. For streamcluster, a moderately sized benchmark by code size, exhaustive exploration of the 23 optimizations would have required more than 8 million executions; instead, ACCEPT's search heuristic considered only 14 composites to produce 7 optimal configurations.

ACCEPT's heuristics help make its profiling step palatable. On our 20-node evaluation cluster for the server applications, the total end-to-end optimization time was typically within a few minutes: times ranged from 14 seconds (sobel) to 11 minutes (x264) with an average of 4 minutes. Tuning for the Zynq and MSP430 platforms was not parallelized and took 19 minutes on average and 5 minutes, respectively.

Accelerator power and energy. We measured power consumption on the Zynq SoC, including its FPGA and DRAM, using a Texas Instruments UCD9240 power supply controller while executing each benchmark in a loop to reach a steady state. Compared to baseline ARM-core–only execution in which the FPGA is not programmed and inactive, power overheads range from from 8.6% (zynq-sobel) to 22.6% (zynq-blackscholes). The zynq-sobel benchmark exhibits lower power overhead because a larger percentage of the program executes on the CPU, putting less load on the FPGA. When we account for the performance gains, energy savings range from 2× (zynq-sobel) to 15.7× (zynq-inversek2j).

7.4 Experiences

This section reports qualitatively on our experiences using ACCEPT to optimize the benchmarks. The programmers included three undergraduate researchers, all of whom were beginners with C and C++ and new to approximate computing, as well as graduate students familiar with the field.

Quality metrics. The first step in tuning a program with ACCEPT is to write a quality metric. In some cases, the program included code to assess output quality. For each remaining case, the programmer wrote a simple Python program (54 lines at most) to parse the program's output and compute the difference between two outputs.

Like any specification, a quality metric can be subtle to write correctly. Although it was not an intended use case, programmers found ACCEPT's dynamic feedback to be helpful in debugging quality metrics. In one instance, ACCEPT reported suspiciously low error for some configurations; these results revealed a quality metric that was ignoring certain missing values in the output and was therefore too permissive.

Initial and iterated annotations. One option when annotating a program for ACCEPT is to first analyze an unannotated program to enumerate all potential optimization sites. However, the programmers preferred to provide an initial annotation set by finding the "core" approximable data in the program—e.g., the vector coordinates in streamcluster or the pixels in sobel. With this data marked as approximate, the type checker reports errors when this data flows into variables that are not yet marked; for each such error, programmers decided whether to add another `APPROX` annotation or to stop the flow of approximation with an `ENDORSE` annotation.

Next, programmers expanded the annotation set to enable more optimizations. Using ACCEPT's analysis log (Section 3.2), they looked for optimizations that could *almost* apply—those that indicated only a small number of blockers.

A persistent consideration was the need to balance effort with potential reward. The programmers focused their attention on parts of the code most likely to provide good quality–efficiency trade-offs. In some cases, it was helpful to take "shortcuts" to program relaxations to test their viability before making them safe. If the programmer was unsure whether a particular lock in a program was contended, for example, it was useful to try eliding that lock to see whether it offered any speedup. Programmers used the `ACCEPT_PERMIT` annotation *temporarily* for an experiment and then, if the optimization proved beneficial, removed the escape-hatch annotation and added the safer `APPROX` and `ENDORSE` annotations.

These experiences highlighted the dual importance of both static and dynamic feedback in ACCEPT. Especially when the programmer is unfamiliar with the application's architecture, the static type errors and conservative precise-purity analysis helped highlight unexpected interactions between components. However, test runs were critical in discovering whether a given subcomputation is important to an algorithm, either in terms of performance or output accuracy.

Code navigation and heuristics. For large programs, programmers reported a need to balance their time between learning the application's architecture and trying new optimizations. (We anticipate that a different strategy would be appropriate when the programmer is already familiar with the code before annotation.) One programmer used a call-graph visualizer to find code closely related to the main computation. In general, more modular code was easier to annotate: when effects are encapsulated, the volume of code related to an optimization is smaller and annotations are more local.

Programmers relied on ACCEPT's analysis feedback for hints about where time would be best spent. They learned to scan for and ignore reports involving memory allocation or system calls, which are rarely fruitful approximation opportunities. Relaxation sites primarily involved with large data arrays were typically good targets.

Encapsulating precise systems programming. The "escape hatches" from ACCEPT's safety analysis were crucial for low-level systems code. In msp430-activity, a routine manipulates memory-mapped registers to read from an accelerometer. The pointers involved in communicating with the memory-mapped peripheral are necessarily precise, but the reading itself is approximate and safe to relax. The

`ACCEPT_PERMIT` escape hatch enabled its optimization. This annotation suggests a pattern in systems programming: the language's last-resort annotations can communicate approximation information about opaque low-level code to ACCEPT.

Self-checking code. The complementary escape hatch, `ACCEPT_FORBID`, was useful for one specific pattern: when benchmarks include code to evaluate their own quality. For example, x264 computes a standard image quality metric and canneal evaluates the total design fitness at every iteration. Programmers used `ACCEPT_FORBID` to ensure that this code, despite involving approximate data, was never corrupted.

8. Related Work

ACCEPT builds on a body of prior work on approximate computing. Three main research directions are most relevant to ACCEPT: static safety analyses for approximate programs, program relaxations, and quality-aware autotuners.

One group of proposals seeks to statically analyze approximate programs to prove properties even in the face of unreliable hardware or program transformations. Carbin et al. propose a general proof system for relating baseline executions to relaxed executions [Carbin et al. 2012] and an integrity property checker for loop perforation [Carbin et al. 2013a]. Rely [Carbin et al. 2013b] and Chisel [Misailovic et al. 2014] analyze and tune the chance that a nondeterministic computation goes wrong. EnerJ [Sampson et al. 2011] provides a simple noninterference guarantee that we adapt in this work. Other recent work [Misailovic et al. 2011; Zhu et al. 2012] uses probabilistic reasoning to prove conservative accuracy guarantees for relaxations of restricted programming patterns. We instead focus on bringing approximation to general programs and rely on programmer involvement and dynamic testing to build confidence in an approximation's suitability. ACCEPT's safety guarantees need to be both general and lightweight, requiring minimal programmer overhead, to be practical. The information-flow types adapted from EnerJ permit straightforward automated compiler reasoning and are not constrained to specific code patterns. EnerJ itself, however, is limited to a specific style of approximation, where individual variables and instructions introduce error on hypothetical future hardware. ACCEPT expands the scope to today's computers using compiler analysis and programmer feedback to enable coarser-grained optimizations.

ACCEPT complements specific software approximation strategies, such as loop perforation [Sidiroglou-Douskos et al. 2011], alternate-implementation selection [Ansel et al. 2009; Baek and Chilimbi 2010; Fang et al. 2014], parameter selection [Hoffmann et al. 2011], synchronization relaxation [Misailovic et al. 2010a, 2012; Renganarayanan et al. 2012; Rinard 2013], and pattern substitution [Samadi et al. 2014]. This work contributes a unifying framework to make relaxations controlled and automatic.

ACCEPT's autotuning component resembles other prior work on dynamic measurement of approximation strategies, including off-line tools such as PetaBricks [Ansel et al. 2011, 2009] and on-line methods such as Green [Baek and Chilimbi 2010], ApproxIt [Zhang et al. 2014], and SAGE [Samadi et al. 2013]. Misailovic et al.'s quality-of-service profiler [Misailovic et al. 2010b] evaluates the results of candidate loop perforations by assessing their performance and quality impacts for reporting to the programmer. Precimonious [Rubio-González et al. 2013] tunes variables' floating-point widths to adjust overall precision. ACCEPT generalizes these autotuning techniques: it applies to a variety of safety-constrained relaxations and guides the process using a practical search heuristic.

The type-qualifier overlay system we develop for Clang and LLVM is modeled after work on practical pluggable types for Java [Ernst; Papi et al. 2008]. Another notable predecessor is Cqual [Foster 2002], which provided a similar mechanism in GCC for finding bugs (e.g., format-string vulnerabilities [Shankar et al. 2001]). ACCEPT's implementation of type qualifiers adds integration with the rest of the compiler toolchain via IR metadata, which enables program analysis beyond type checking and type-based optimization.

Our neural accelerator design builds on recent work on using hardware neural networks as accelerators [Esmaeilzadeh et al. 2012a; St. Amant et al. 2014; Belhadj et al. 2013]. To our knowledge, this is the first work to demonstrate safe, automated, compiler-based offloading to the neural network.

9. Conclusion

Many important classes of applications can tolerate some imprecision. Programs from diverse domains such as imaging, machine learning, vision, physical simulation, and embedded sensing exhibit trade-offs between accuracy and performance. Approximate program relaxations hold the potential to dramatically improve performance and energy usage while only minimally impacting output quality. But these transformations are potentially destructive and can have subtle, far-reaching effects. Programmers need better tools for program relaxation that are safer than manual reasoning but more practical than opaque automatic transformation.

ACCEPT is a compiler framework for approximate programming that balances automation with programmer insight. It supports a wide range of approximate code transformations, including both pure-software relaxations and offloading to hardware accelerators. This paper demonstrates that ACCEPT can yield substantial end-to-end performance benefits with little quality degradation.

Approximate computing research is in its infancy and needs more tools for prototyping and evaluating ideas. The ACCEPT framework can simplify the practical implementation of new optimization strategies and accelerators. We hope to make ACCEPT a common, open-source platform for approximate computing research.

Acknowledgments

This work was supported in part by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, by NSF grant #1216611, by a Qualcomm Innovation Fellowship, and by gifts from Microsoft. We thank Dan Grossman for reading a draft of this paper.

References

T. M. Aamodt and P. Chow. Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy. *ACM TECS*, 7(3):26:1–26:27, May 2008.

J. Ansel, C. P. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. P. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *PLDI*, 2009.

J. Ansel, Y. L. Wong, C. P. Chan, M. Olszewski, A. Edelman, and S. P. Amarasinghe. Language and compiler support for autotuning variable-accuracy algorithms. In *CGO*, 2011.

W. Baek and T. M. Chilimbi. Green: A framework for supporting energy-conscious programming using controlled approximation. In *PLDI*, 2010.

B. Belhadj, A. Joubert, Z. Li, R. Heliot, and O. Temam. Continuous real-world inputs can open up alternative accelerator designs. In *International Symposium on Computer Architecture (ISCA)*, pages 1–12, 2013.

C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, Jan. 2011.

M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *PLDI*, 2012.

M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Verified integrity properties for safe approximate program transformations. In *PEPM*, 2013a.

M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability of programs that execute on unreliable hardware. Technical Report MIT-CSAIL-TR-2013-014, MIT, June 2013b.

T. Chen, Y. Chen, M. Duranton, Q. Guo, A. Hashmi, M. H. Lipasti, A. Nere, S. Qiu, M. Sebag, and O. Temam. BenchNN: On the broad potential application scope of hardware neural network accelerators. In *IISWC*, 2012.

Clang. Clang: a C language family frontend for LLVM. `http://clang.llvm.org`.

G. B. Dantzig. Discrete-variable extremum problems. *Operations Research*, 5(2):266–277, 1957.

M. D. Ernst. Type annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`.

H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012a.

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

S. Fang, Z. Du, Y. Fang, Y. Huang, Y. Chen, L. Eeckhout, O. Temam, H. Li, Y. Chen, and C. Wu. Performance portability across heterogeneous socs using a generalized library-based approach. *ACM TACO*, 11(2):21:1–21:25, June 2014.

J. S. Foster. *Type Qualifiers: Lightweight Specifications to Improve Software Quality*. PhD thesis, University of California, Berkeley, Dec. 2002.

S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. The D editor: a new interactive parallel programming tool. In *Supercomputing*, 1994.

H. Hoffmann, S. Sidiroglou, M. Carbin, S. Misailovic, A. Agarwal, and M. Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, 2004.

S. Liu, K. Pattabiraman, T. Moscibroda, and B. G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

S. Misailovic, D. Kim, and M. Rinard. Parallelizing sequential programs with statistical accuracy tests. Technical Report MIT-CSAIL-TR-2010-038, MIT, Aug. 2010a.

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

S. Misailovic, D. M. Roy, and M. C. Rinard. Probabilistically accurate program transformations. In *SAS*, 2011.

S. Misailovic, S. Sidiroglou, and M. C. Rinard. Dancing with uncertainty. In *Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

S. Misailovic, M. Carbin, S. Achour, Z. Qi, and M. C. Rinard. Chisel: Reliability- and accuracy-aware optimization of approximate computational kernels. In *OOPSLA*, 2014.

T. Moreau, M. Wyse, J. Nelson, A. Sampson, H. Esmaeilzadeh, L. Ceze, and M. Oskin. SNNAP: Approximate computing on programmable SoCs via neural acceleration. 2015. Preprint provided by authors.

M. M. Papi, M. Ali, T. L. C. Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA*, 2008.

L. Renganarayanan, V. Srinivasan, R. Nair, and D. Prener. Programming with relaxed synchronization. In *Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012.

M. Rinard. Parallel synchronization-free approximate data structure construction. In *HotPar*, 2013.

M. Ringenburg and S.-E. Choi. Optimizing loop-level parallelism in Cray XMT applications. In *Cray User Group Proceedings*, May 2009.

C. Rubio-González, C. Nguyen, H. D. Nguyen, J. Demmel, W. Kahan, K. Sen, D. H. Bailey, C. Iancu, and D. Hough. Precimonious: Tuning assistant for floating-point precision. In *SC*, Nov. 2013.

M. Samadi, J. Lee, D. A. Jamshidi, A. Hormati, and S. Mahlke. Sage: Self-tuning approximation for graphics engines. In *MICRO*, 2013.

M. Samadi, D. A. Jamshidi, J. Lee, and S. Mahlke. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS*, 2014.

A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. Design of an RFID-based battery-free programmable sensing platform. *IEEE Transactions on Instrumentation and Measurement*, 57(11):2608–2615, Nov. 2008.

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.

A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze. Expressing and verifying probabilistic assertions. In *PLDI*, 2014.

E. Schkufza, R. Sharma, and A. Aiken. Stochastic optimization of floating-point programs with tunable precision. In *PLDI*, 2014.

U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, Aug. 2001.

S. Sidiroglou-Douskos, S. Misailovic, H. Hoffmann, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

R. St. Amant, A. Yazdanbakhsh, J. Park, B. Thwaites, H. Esmaeilzadeh, A. Hassibi, L. Ceze, and D. Burger. General-purpose code acceleration with limited-precision analog computation. In *ISCA*, 2014.

O. Temam. A defect-tolerant accelerator for emerging high-performance applications. In *ISCA*, 2012.

Texas Instruments, Inc. MSP430 Ultra-Low Power Microcontrollers. http://www.ti.com/msp430.

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):273–286, 2000.

Q. Zhang, F. Yuan, R. Ye, and Q. Xu. ApproxIt: An approximate computing framework for iterative methods. In *DAC*, 2014.

Z. A. Zhu, S. Misailovic, J. A. Kelner, and M. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.