HARDWARE AND SOFTWARE FOR APPROXIMATE COMPUTING

ADRIAN SAMPSON

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2015

Reading Committee:
Luis Ceze, Chair
Dan Grossman, Chair
Mark Oskin

Program Authorized to Offer Degree:
Computer Science & Engineering

ABSTRACT

HARDWARE AND SOFTWARE FOR APPROXIMATE COMPUTING

Adrian Sampson

Chairs of the Supervisory Committee:
Associate Professor Luis Ceze
Associate Professor Dan Grossman
Computer Science & Engineering

Approximate computing is the idea that we are hindering computer systems' efficiency by demanding too much accuracy from them. While precision is crucial for some tasks, many modern applications are fundamentally approximate. Perfect answers are unnecessary or even impossible in domains such as computer vision, machine learning, speech recognition, search, graphics, and physical simulation. Today's systems waste time, energy, and complexity to provide uniformly pristine operation for applications that do not require it.

Resilient applications are not, however, a license for computers to abandon predictability in favor of arbitrary errors. We need abstractions that incorporate approximate operation in a *disciplined* way. Application programmers should be able to exploit these richer abstractions to treat accuracy as a resource and trade it off for more traditional resources such as time, space, or energy.

This dissertation explores new abstractions for approximate computing across hardware and software. It develops these abstractions from two perspectives: from the point of view of *programmers*, where the challenge is constraining imprecision to make it acceptable, and from a *system* perspective, where the goal is to exploit programs' constraints to improve efficiency. For approximate programming, this dissertation proposes:

- a type system that uses information flow to separate an application's error-resilient components from its critical control structures;
- an extended type system that restricts the probability that a value is incorrect, along with type inference and optional dynamic tracking for these probabilities; and
- a construct for expressing probabilistic constraints on programs along with a technique for verifying them efficiently using symbolic execution and statistical properties.

For approximate execution, it describes:

- two mechanisms for trading off accuracy for density, performance, energy, and lifetime in solid-state memory technologies; and
- an end-to-end compiler framework for exploiting approximation on commodity hardware, which also serves as research infrastructure for experimenting with new approximation ideas.

The ordered swirl of houses and streets, from this high angle, sprang at her now with the same unexpected, astonishing clarity as the circuit card had… There'd seemed no limit to what the printed circuit could have told her (if she had tried to find out); so in her first minute of San Narciso, a revelation also trembled just past the threshold of her understanding.

— Thomas Pynchon, *The Crying of Lot 49*

CONTENTS

ACKNOWLEDGMENTS

I acknowledge that pursuing a Ph.D. is only made worthwhile by the people who surround you along the way.

This dissertation describes work done by a staggering array of amazing researchers who are not me. My co-authors are too numerous to list exhaustively here, but they can be found hiding in the references [22, 180–182]. Notably, Werner Dietl masterminded EnerJ's semantics, Karin Strauss is the memory technology expert, and Pavel Panchekha is behind the formalism for probabilistic assertions. DECAF in Chapter 4 is Brett Boston's research project for the honors designation on his bachelor's degree. He did all the hard work; I played the role of a meddling mentor.

Most of all, I have had the unique honor of being advised by the two best mentors in the computer-science universe, Luis Ceze and Dan Grossman. Luis is an unstemmable font of creativity, a fierce advocate, and a true research visionary. Dan is a tireless champion of good taste, an enemy of bullshit, and a stalwart encyclopedia of great advice. No page in this disseration could exist without either of them.

Thank you to my staggeringly brilliant collaborators. Emily Fortuna lent a hand early on, along with Danushen Gnanapragasam. Hadi Esmaeilzadeh took the first plunge into approximate hardware with us and his advisor, Doug Burger. Jacob Nelson, Thierry Moreau, Andre Baixo, Ben Ransford, Mark Wyse, and Michael Ringenburg were all instrumental to defining the approximaiton agenda. Ben Wood inducted me into research in the best way possible.

Thank you to the UW undergraduates who withstood my mentoring: Luyi Liu, Chengfeng Shi, Joshua Yip, Brett Boston, Wenjie (Marissa) He, Finn Parnell, and Danushen Gnanapragasam. Their talents are unbounded.

Thank you to my research groups, Sampa and PLSE. Extra-large thanks to my role models in Sampa: Joe Devietti, Brandon Lucia and Jacob Nelson. They defined computer architecture for me. A very real thank you to Sampa's own Owen Anderson, Katelin Bailey, Tom Bergan, James Bornholt, Carlo del Mundo, Hadi Esmaeilzadeh, Emily Fortuna, Brandon Holt, Nick Hunt, Vincent Lee, Eric Mackay, Amrita Mazumdar, Thierry Moreau, Brandon Myers, Ben Ransford, Michael Ringenburg, Ben Wood, and Mark Wyse. They endured my many terrible ideas and made grad school bearable. Thank you to Melody Kadenko, without whom everything would immediately collapse.

Thank you to my menagerie of mentors: Mark Oskin, the godfather of the Sampa group, Călin Caşcaval from Qualcomm, and Karin Strauss, Todd Mytkowicz, and Kathryn McKinley from Microsoft Research. Thank you to my patient PhD committee, including Maryam Fazel, Eric Klavins, Vivesh Sathe, and Hank Levy, who deserves a whole second category of gratitude. Thank you to the entire faculty at UW CSE, who said so many encouraging things over the years, and who occasionally endured merciless holiday-party-skit lampooning.

Many exceptionally emphatic thanks to Lindsay Michimoto, who did so many things to make grad school work for me that they cannot possibly fit here, and whose constant kindness and inexhaustible work ethic continue to inspire me. Thank you to my surprisingly helpful officemate-mentors, Ivan Beschastnikh and Chloé Kiddon.

Thank you to my advisors and mentors at Harvey Mudd, Ran Libeskind-Hadas, Melissa O'Neill, Bob Keller, and Geoff Kuenning, who are more responsible than anyone for getting me into this in the first place. Double thanks to Ran, from whom I first really learned what computer science is, and who is still an inspiration every day. Triple thanks to Ran for his prescient advice to pursue anything else over theory.

Thank you to my parents, Elizabeth Dequine and Ed Sampson III, for everything and also for their tolerance of my inexplicable interests. Thank you to my brother, Devon Sampson, who beat me to a Ph.D. (but only by a few days). Infinite and everlasting thanks to the magnificent Ariana Taylor-Stanley, the basis of the whole thing.

Part I

APPROXIMATE COMPUTING

1

OVERVIEW

1.1 INTRODUCTION

Accuracy and reliability are fundamental tenets in computer system design. Programmers can expect that the processor never exposes timing errors, and networking stacks typically aim to provide reliable transports even on unreliable physical media. When errors do occasionally happen, we treat them as exceptional outliers, not as part of the system abstraction. Cosmic rays can silently flip bits in DRAM, for example, but the machine will typically use error-correcting codes to maintain the illusion for programmers that the memory is infinitely reliable.

But abstractions with perfect accuracy come at a cost. Chips need to choose conservative clock rates to banish timing errors, storage and communication channels incur error-correction overhead, and parallelism requires expensive synchronization.

Meanwhile, many applications have intrinsic tolerance to inaccuracy. Applications in domains like computer vision, media processing, machine learning, and sensor data analysis already incorporate imprecision into their design. Large-scale data analytics focus on aggregate trends rather than the integrity of individual data elements. In domains such as computer vision and robotics, there are no perfect answers: results can vary in their usefulness, and the output quality is always in tension with the resources that the software needs to produce them. All these applications are *approximate programs*: a range of possible values can be considered "correct" outputs for a given input.

From the perspective of an approximate program, today's systems are overprovisioned with accuracy. Since the program is resilient, it does not need every arithmetic operation to be precisely correct and every bit of memory to be preserved at the same level of reliability. *Approximate computing* is a research agenda that seeks to better match the accuracy in system abstractions with the needs of approximate programs.

DISCIPLINED APPROXIMATION The central challenge in approximate computing is forging abstractions that make imprecision *controlled and predictable* without sacrificing its efficiency benefits. This goal of this dissertation is to design hardware and software around approximation-aware abstractions that, together, make accuracy–efficiency trade-offs attainable for programmers. My work examines approximate abstractions in the contexts of programming lan-

guages, computer architecture, memory technologies, compilers, and software development tools.

1.2 RESEARCH PRINCIPLES

The work in this dissertation is organized around five principles for the design of disciplined approximate abstractions. These themes represent the collective findings of the concrete research projects described later. The principles are:

1. Result quality is an application-specific property.
2. Approximate abstractions should distinguish between safety properties and quality properties.
3. Hardware and software need to collaborate to reach the best potential of approximate computing.
4. Approximate programming models need to incorporate probability and statistics.
5. The granularity of approximation represents a trade-off between generality and potential efficiency.

This section outlines each finding in more detail.

1.2.1 *Result Quality is Application Specific*

Since approximate computing navigates trade-offs between efficiency and result quality, it needs definitions of both sides of the balance. While *efficiency* can have universal definitions—the time to completion, for example, or the number of joules consumed—output *quality* is more subtle. A key tenet in this work is is that applications must define "output quality" case by case: the platform cannot define quality without information from the programmer.

Following this philosophy, the system designs in this dissertation assume that each approximate program comes with a *quality metric*, expressed as executable code, that scores the program's output on a continuous scale from 0.0 to 1.0. A quality metric is the approximate-computing analog to a traditional software *specification*, which typically makes a binary decision about whether an implementation is correct or incorrect. Just as ordinary verification and optimization tools start from a specification, approximate-computing tools start with a quality metric.

1.2.2 *Safety vs. Quality*

At first glance, a quality metric seems like sufficient information to specify an application's constraints on approximation. If the system can guarantee that a program's output will always have a quality score above q, and the programmer decides that q is good enough, what could possibly go wrong?

In reality, it can be difficult or impossible for systems to prove arbitrary quality bounds with perfect certainty. Realistic tools can often only certify, for example, that any output's quality score will be at least q *with high probability*, or that *nearly every output* will exceed quality q but rare edge cases may do worse. Even more fundamentally, it can be difficult for programmers to devise formal quality metrics that capture every possible factor in their intuitive notion of output quality. Quality metrics can be simpler if their scope is narrowed to data where they are most relevant: the pixels in an output image, for example, but not the header data.

To that end, this dissertation embraces *safety* as a separate concept from quality. A safety property, in the context of approximate computing, is a guarantee that part of a program *never* deviates from its precise counterpart—in other words, that it matches the semantics of a traditional, non-approximate system. A quality property, in contrast, constrains the *amount* that approximate program components deviate from their precise counterparts.

In practice, we find a first-order distinction between *no approximation at all* and *approximation of some nonzero degree* both simplifies reasoning for programmers and makes tools more tractable. My work has demonstrated that the two kinds of properties can be amenable to very different techniques: information flow tracking (Chapter 3) is appropriate for safety, for example, but statistical hypothesis testing (Chapter 5) is better for quality.

1.2.3 *Hardware–Software Co-Design*

Some of the most promising ideas unlock new sources of efficiency that are only available in hardware: exploiting the analog behavior of transistors, for example, or mitigating the cost of error correction in memory modules. Because approximation techniques have subtle and wide-ranging effects on program behavior, however, designs that apply them *obliviously* are unworkable. Instead, researchers should co-design hardware techniques with their software abstractions to ensure that programmers can control imprecision.

Hardware designs can also rely on guarantees from software—the language or compiler—to avoid unnecessary complexity. The Truffle approximate CPU [59], for example, avoids expensive hardware consistency checks by exploiting EnerJ's compile-time enforcement of type safety. Wherever possible, hardware researchers should offload responsibilities to complementary software systems.

1.2.4 *Programming with Probabilistic Reasoning*

Often, the most natural ways to reason about approximation and quality use probabilistic tools. Probabilistic reasoning lets us show show statements such as *this output will be high-quality with at least probability* P or *an input randomly selected from this distribution leads to a high-quality output with probability* P'. These probabilistic statements can simultaneously match the nondeterministic behavior of approximate systems [59, 60, 181] and correspond to software quality criteria [22, 182].

To support reasoning about quality, approximate programming models need to incorporate abstractions for statistical behavior. The DECAF type system, in Chapter 4, and probabilistic assertions, in Chapter 5, represent two complementary approaches to reasoning about probabilistic quality properties.

These approaches dovetail with the recent expansion of interest in *probabilistic programming languages*, which seek to augment machine-learning techniques with language abstractions [69]. Approximate programming systems can adapt lessons from this body of research.

1.2.5 *Granularity of Approximation*

The *granularity* at which approximate computing applies is a nonintuitive but essential factor in its success. My and other researchers' work has explored approximation strategies at granularities of both extremes: fine-grained approximations that apply to individual instructions and individual words of memory (e.g., Truffle [59]); and coarse-grained approximations that holistically transform entire algorithms (e.g., neural acceleration [60]).

A technique's granularity affects its generality and its efficiency potential. A fine-grained approximation can be very general: an approximate multiplier unit, for example, can potentially apply to any multiplication in a program. But the efficiency gains are fundamentally limited to *non-control* components, since control errors can disrupt execution arbitrarily. Even if an approximate multiplier unit can be very efficient, the same technique can never improve the efficiency of a branch, an address calculation, or even the scheduling of an approximate multiply instruction. Approximations that work at a coarser granularity can address control costs, so their potential gains are larger. But these techniques tend to apply more narrowly: techniques that pattern-match on algorithm structures [176], for example, place nuanced restrictions on the code they can transform.

The EnerJ language in Chapter 3 was initially designed for fine-grained hardware approximation techniques such as low-voltage functional units. While the granularity was good for programmability, it was bad for efficiency: our detailed hardware design for fine-grained hardware approximation [59] demonstrated limited benefit. The ACCEPT compiler in Chapter 7 bridges the gap: its analysis library and optimizations exploit the fine-grained annotations from EnerJ to safely apply coarse-grained optimizations.

1.3 ABSTRACTIONS FOR DISCIPLINED APPROXIMATION

This dissertation supports the above research principles using a set of concrete system designs. The systems comprise programming-language constructs that express applications' resilience to approximation along with system-level techniques for exploiting that latent resilience to gain efficiency. This section serves as an overview of the interlocking designs; Parts II and III give the full details.

1.3.1 *Controlling Safety and Quality*

The first set of projects consists of language abstractions that give programmers control over safety and quality in approximate programs.

1.3.1.1 *Information Flow Tracking for General Safety*

EnerJ, described in Chapter 3, is a type system for enforcing safety in the presence of approximation. The key insight in EnerJ is that approximate programs tend to consist of two intermixed kinds of storage and computation: critical control components and non-critical data components. The latter, which typically form the majority of the program's execution, are good candidates for approximation, while the former should be protected from error and carry traditional semantics.

EnerJ lets programmers enforce a separation between critical and non-critical components. It uses a type system that borrows from static information flow systems for security [138, 174] to provide a static noninterference guarantee for precise data. EnerJ extends Java with two type qualifiers, `@Approx` and `@Precise`, and uses a subtyping relationship to prevent approximate-to-precise information flow. Using EnerJ, programmers can rely on a proof that data marked as precise remains untainted by the errors arising from approximation.

A key design goal in EnerJ is its *generality:* the language aims to encapsulate a range of approximation strategies under a single abstraction. Its type system covers approximate storage via the types of variables and fields; approximate processor logic via overloading of arithmetic operators; and even user-defined approximate algorithms using dynamic method dispatch based on its approximating qualifiers.

EnerJ addresses safety, not quality: a variable with the type `@Approx float` can be arbitrarily incorrect and EnerJ does not seek to bound its incorrectness. By leaving the complementary concern of controlling quality to separate mechanisms, EnerJ keeps its type system simple.

1.3.1.2 *Extending EnerJ with Probability Types*

DECAF, in Chapter 4, extends EnerJ's type-based approach to safety with quality guarantees. The idea is to generalize the original `@Approx` type qualifier to a parameterized qualifier `@Approx(`p`)`, where p dictates the *degree* of approximation. Specifically, in DECAF, p is the lower bound on the probability that a value is *correct:* that the value in an approximate execution equals its counterpart in a completely precise execution of the same program. DECAF defines sound type rules for introducing and propagating these correctness probabilities.

DECAF's added sophistication over EnerJ's simple two-level system comes at a cost in complexity: a type system that requires probability annotations on every expression would quickly become infeasible for programmers. To mitigate annotation overhead, DECAF adds type inference. Sparse probability annotations on the inputs and outputs of coarse-grained subcomputations are typically enough for DECAF's inference system to determine the less-intuitive probabili-

ties for intermediate values. Crucially, DECAF places no constraints on where programmers can write explicit annotations: developers can write probabilities where they make the most sense and leave the remaining details to the compiler.

DECAF addresses the limitations of a conservative quality analysis using an optional dynamic-tracking mechanism. The inference system also allows efficient code reuse by specializing functions according to the accuracy constraints of their calling contexts.

1.3.1.3 *Probabilistic Assertions*

DECAF's approach to controlling quality achieves strong probabilistic guarantees by constraining the range of possible approximation strategies: it works only with techniques where errors appear at an operation granularity; whey they occur randomly but rarely; and when the error probability is independent of the input values.

A complementary project takes the opposite approach: it accommodates any probability distribution, but it offers weaker guarantees. The idea is to use statistical hypothesis tests to prove properties up to a *confidence level:* to allow a small probability of "verifying" a false property.

The technique is based on a new language construct called a *probabilistic assertion*. The construct is analogous to a traditional assertion: `assert` e expresses that the expression e must always be true. A probabilistic assertion:

`passert` e`,` p`,` c

indicates that e must be true with at least probability p, and the system has to prove the property at confidence level c. These assertions can encode important quality properties in approximate programs, such as bounds on the frequency of "bad" pixels produced by an image renderer. The same construct is useful in other domains where probabilistic behavior is essential, such as when dealing with noisy sensors.

Chapter 5 describes probabilistic assertions in more detail along with a workflow for verifying them efficiently. The verifier uses a symbolic-execution technique to extract a representation of a program's probabilistic behavior: a Bayesian network. The verifier can optimize this Bayesian-network representation using off-the-shelf statistical properties that are difficult to apply to the original program code. The complete workflow can make probabilistic-assertion verification dozens of times faster to check than a naive stress-testing approach.

1.3.2 *Exploiting Resilience for Efficiency*

The second category of research is on the implementation of systems that exploit programs' tolerance for approximation to improve efficiency. This dissertation describes two projects: an architectural technique and an end-to-end compiler toolchain. A primary concern in both systems is exposing an abstraction that fits with the safety and quality constraints introduced in the above language abstractions.

1.3.2.1 *Approximate Storage for Solid-State Memory Technologies*

One system design, detailed in Chapter 6, builds on a trend in hardware technologies. It exploits unique properties of new solid-state memories, such as flash memory and phase-change memory, to implement two orthogonal trade-offs between resource and accuracy.

The first technique recognizes that the underlying material in these memory technologies is analog. Traditional designs build a clean digital abstraction on top of a fundamentally analog memory cell. Our technique addresses the cost of that digital abstraction by letting applications opt into stochastic data retention.

The second technique embraces resistive memory technologies' tendency to wear out. Ordinarily, architectures need to detect failed memory blocks and avoid storing data in them—limiting the memory module's useful lifetime. Instead, in the context of an approximate application, we can harvest the otherwise-unusable blocks and store approximate data in them.

Both strategies need a new set of common CPU and operating-system interfaces to let software communicate error resilience and bit layout information. We develop these abstractions to match the structure and semantics of EnerJ.

1.3.2.2 *ACCEPT: An Approximate Compiler*

The final system design takes a different tactic: rather than simulating hypothetical hardware, the idea is to build a practical infrastructure for experimenting with approximation in the nearer term. Chapter 7 introduces ACCEPT, an open-source compiler workflow designed both for practitioners, to try out approximation techniques on their code, and for researchers, to prototype and evaluate new ideas for approximation strategies.

The first challenge that ACCEPT faces is to bridge the granularity gap (see Section 1.2.5, above). EnerJ's fine-grained annotations can be more general and easier to apply to programs, but coarse-grained optimizations can offer better efficiency gains—especially in the pure-software domain. ACCEPT's interactive optimization architecture, compiler analysis library, and auto-tuner infrastructure help connect fine-grained safety annotations to coarse-grained optimizations.

ACCEPT also addresses a second persistent challenge in approximate programmability: balancing automation with programmer control. Fully manual approximation can be tedious and error prone, but fully automatic systems can also frustrate developers by isolating them from decisions that can break their code. ACCEPT relies on the distinction between quality and safety (see Section 1.2.2) to reconcile the extremes. Type annotations resembling EnerJ's enforce safety, but programmers are kept in the loop with an interactive optimization workflow to rule out unexpected quality effects. Together, the systems leverage the best of both factors: programmer insight for preserving application-specific properties and automatic compiler reasoning for identifying obscure data flows.

1.4 OTHER WORK

The work in this document is intimately connected to other research I collaborated on while at the University of Washington. While this dissertation does not fully describe these related projects, their influence is evident in the trajectory of projects that do appear here. For context, this section describes a handful of other projects on approximate hardware and developer tools.

1.4.1 *An Approximate CPU and ISA*

Truffle is a processor architecture that implements EnerJ's semantics to save energy [59]. It uses a secondary, subcritical voltage that allows timing errors in a portion of the logic and retention errors in a portion of the SRAM.

To expose the two voltages to software, we designed an ISA extension that includes a notation of abstract approximation. The code can choose dynamically to enable approximation per instruction, per register, and per cache line. A key challenge in the design was supporting an ISA that could efficiently support an EnerJ-like programming model, where the precise and approximate components of a program remain distinct but interleave at a fine grain.

Our simulation of the Truffle design space yielded results ranging from a 5% energy consumption *increase* to a 43% reduction. These results emphasize the efficiency limits of very fine-grained approximation (see the granularity principle in Section 1.2.5). Even in a maximally approximate program—in which every arithmetic instruction and every byte of memory is marked as approximate—much of Truffle's energy is spent on precise work. Fetching code, scheduling instructions, indexing into SRAMs, computing addresses, and tracking precision state all must be performed reliably. Modern processors spend as much energy on control as they do on computation itself, so any technique that optimizes only computation will quickly encounter Amdahl's law.

The Truffle work in appears in the dissertation of Hadi Esmaeilzadeh [57].

1.4.2 *Neural Acceleration*

Neural acceleration is a technique that explores the opposite end of the granularity spectrum [60, 137]. The idea is to use machine learning to imitate a portion of a computation by observing its input–output behavior. Then, we build a configurable hardware accelerator to efficiently execute the learned model in place of the original code. Our specific design uses neural networks: since neural networks have efficient hardware implementations, the transformed function can be much faster and lower-power than the original code.

The coarse granularity pays off in efficiency: our simulations demonstrated a 3× average energy reduction. But the coarser granularity comes at a cost of programmer visibility and control. Since the NPU technique treats the target code as a black box, the programmer has no direct influence over the performance and accuracy of the resulting neural network. These conflicting objectives demonstrate the need for techniques that bridge the granularity gap.

The original neural acceleration work also appears in Hadi Esmaeilzadeh's dissertation [57]. I also worked on a recent extension of the idea for programmable logic [137].

1.4.3 *Monitoring and Debugging Quality*

Many approaches to making approximation programmable focus on proving conservative, static bounds. As in traditional software development, approximate computing also needs complementary dynamic techniques. To this end, I contributed to a pair of techniques for dynamically controlling result quality [169].

The first dynamic system is a framework for monitoring quality in deployment. The goal is to raise an exception whenever the program produces a "bad" output. While the ideal monitoring system would directly measure the quality degradation of every output, perfect measurement is too expensive for run-time deployment. Our framework provides a range of techniques for specific scenarios where we can make monitoring cheap enough to be feasible.

The second system is a debugging tool. The idea is that certain subcomputations can be more important to quality than others, but that this difference is not necessarily obvious to programmers. The tool identifies and blames specific approximation decisions in a large codebase when they are responsible for too much quality degradation.

The work on dynamic quality analysis appears in the dissertation of Michael F. Ringenburg [167].

1.5 ORGANIZATION

The next chapter is a literature survey of work on efficiency–accuracy trade-offs. Historical context is particularly important to this dissertation because the fundamental idea of exchanging accuracy for returns in efficiency is so old: analog computers and floating-point numbers, for example, are prototypical examples of approximate-computing strategies.

Parts II and III form the core of the dissertation. They comprise five independent but interlocking research projects that together build up abstractions for making approximate computing both tractable and efficient. Part II describes three approaches to abstracting approximation in programming languages: EnerJ, a type system that uses type qualifiers to make approximation safe; DECAF, an extension of EnerJ that adds probabilistic reasoning about the likelihood that data is correct; and probabilistic assertions, a strategy for efficiently verifying complex probabilistic properties via sampling. Part III describes two system designs for implementing efficiency–accuracy trade-offs: a hardware architecture that exploits the nuances of resistive memory technologies such as phase-change memory; and an open-source compiler toolkit that provides the scaffolding to quickly implement new approximation strategies while balancing programmability with approximation's potential benefits.

Finally, Chapters 8 and 9 look forward and backward, respectively. The retrospective chapter distills lessons from the work in this dissertation about approximate computing and hardware–software co-design in general, and the prospective chapter suggests next steps for bringing approximation into the mainstream.

This dissertation also includes appendices that formalize the programming-languages techniques in Part II and prove their associated theorems.

1.6 PREVIOUSLY PUBLISHED MATERIAL

This dissertation comprises work published elsewhere in conference papers:

- Chapter 3: *EnerJ: Approximate Data Types for Safe and General Low-Power Computation.* Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. In Programming Language Design and Implementation (PLDI), 2011. [180]
- Chapter 4: *Probability Type Inference for Flexible Approximate Programming.* Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. To appear in Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015. [22]
- Chapter 5: *Expressing and Verifying Probabilistic Assertions.* Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn McKinley, Dan Grossman, and Luis Ceze. In Programming Language Design and Implementation (PLDI), 2014. [182]
- Chapter 6: *Approximate Storage in Solid-State Memories.* Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. In the IEEE/ACM International Symposium on Microarchitecture (MICRO), 2013. [181]

The appendices draw on expanded material accompanying these papers: Appendix A reflects the EnerJ technical report [179], Appendix B uses text from the DECAF paper's included appendix [22], and Appendix C corresponds to the accompanying digital material for the probabilistic assertions paper [183].

2

SURVEY

Approximate computing research combines insights from hardware engineering, architecture, system design, programming languages, and even application domains like machine learning. This chapter summarizes research on implementing, exploiting, controlling, and reasoning about approximation in computer systems. To confine the scope, the survey focuses on work that exposes error to applications (unlike fault tolerance, which seeks to *hide* errors), and on work that is in some sense general (not, for example, a new approximation strategy for one specific graphics algorithm).

2.1 APPLICATION TOLERANCE STUDIES

Many authors have identified the property of error tolerance in existing "soft" applications. A large class of studies have examined this property by injecting errors into certain parts of applications and assessing the execution quality in terms of both crashes and output fidelity [42, 63, 76, 96, 108–110, 123, 172, 206, 207, 226, 230]. Related studies have evaluated error-resilience in integrated circuit designs [24, 44]. This category of study repeatedly finds that different parts of the application have different impacts on reliability and fidelity. Some conclude that there is a useful distinction between critical and non-critical program points, typically instructions [76, 113, 206, 207]. This conclusion reflects the safety principle in Section 1.2.2: certain program components, especially those involved in control flow, need to be protected from all of approximation's effects.

This work tends to assume an existing, domain-specific notion of "quality" for each application. As the principle in Section 1.2.1 suggests, these quality metrics need careful consideration: one quality metric is not necessarily just as good as another. Recent work has proposed guidelines for rigorous quality measurement [4].

2.2 EXPLOITING RESILIENCE IN ARCHITECTURE

Hardware techniques for approximation can lead to gains in energy, performance, manufacturing yield, or verification complexity. We categorize hardware-based approximation strategies according to the hardware component they affect: computational units, memories, or entire system architectures.

2.2.1 *Functional Units*

Researchers have designed floating-point units that dynamically adapt mantissa width [210, 229], "fuzzily" memoize similar arithmetic computations [5], or tolerate timing errors [78, 86, 136]. Alternative number representations work in tandem with relaxed functional units to bound the numerical error that can result from bit flips [198].

The VLSI community has paid particular attention to variable-accuracy adder designs, which are allowed to yield incorrect results for some minority of input combinations [72, 73, 87, 90, 111, 126, 191, 218, 223, 228, 238].

2.2.2 *Memory*

SRAM structures spend significant static power on retaining data, so they represent another opportunity for fidelity trade-offs [35, 99, 193]. Similarly, DRAM structures can reduce the power spent on refresh cycles where bit flips are allowed [113, 117]. In persistent memories where storage cells can wear out, approximate systems can reduce the number of bits they flip to lengthen the useful device lifetime [64]. Similarly, low-power writes to memories like flash can exploit its probabilistic properties while hiding them from software [112, 175, 211]. Spintronic memories exhibit similarly favorable trade-offs between access cost and error [161].

These memory approximation techniques typically work by exposing soft errors and other analog effects. Recent work in security has exploited patterns in these variability-related errors to deanonymize users [158].

2.2.3 *Circuit Design*

A broad category of work has proposed general techniques for making quality trade-offs when synthesizing and optimizing general hardware circuits [11, 20, 125, 157, 160, 215, 216, 227]. Other tools focus on analyzing approximate circuit designs [212, 217].

Near-threshold voltage domains also present a new opportunity for embracing unpredictable circuit operation [89].

2.2.4 *Relaxed Fault Tolerance*

As a dual to adding errors in some circuits, some researchers have explored differential fault protection in the face of universally unreliable circuits. As process sizes continue to shrink, it is likely that reliable transistors will become the minority; redundancy and checking will be necessary to provide reliable operation [106]. Circuit design techniques have been proposed that reduce the cost of redundancy by providing it selectively for certain instructions in a CPU [202], certain blocks in a DSP [6, 75, 88], or to components of a GPU [143]. Other work has used criticality information to selectively allocate software-level error detection and correction resources [92, 97, 192].

2.2.5 *Microarchitecture*

Microarchitectural mechanisms can exploit different opportunities from circuit-level techniques. Specifically, "soft coherence" relaxes intercore communication [116], and load value approximation [128, 208] approximates numerical values instead of fetching them from main memory on cache misses.

Recent work has proposed system organizations that apply approximation at a coarser grain. One set of techniques uses external monitoring to allow errors even in processor control logic [232, 233]. Other approaches compose separate processing units with different levels of reliability [103]. Duwe [53] proposes run-time coalescing of approximate and precise computations to reduce the overhead of switching between modes. Other work allocates approximation among the lanes of a SIMD unit [2]. In all cases, the gains from approximation can be larger than for lower-level techniques that affect individual operations. As the granularity principle from Section 1.2.5 outlines, techniques like these that approximate entire computations, including control flow, have the greatest efficiency potential.

2.2.6 *Stochastic Computing*

Stochastic computing is an alternative computational model where values are represented using probabilities [9, 34, 43, 120, 139, 142, 219]. For example, a wire could carry a random sequence of bits, where the wire's value corresponds to the probability that a given bit is a 1. Multiplication can be implemented in this model using a single *and* gate, so simple circuits can be low-power and area-efficient. A persistent challenge in stochastic circuits, however, is that reading and output value requires a number of bits that is exponential in the value's magnitude. Relaxing this constraint represents an opportunity for an time–accuracy trade-off.

2.3 EXPLOITING RESILIENCE WITH PROGRAM TRANSFORMATIONS

Aside from hardware-level accuracy trade-offs, there are opportunities for adapting *algorithms* to execute with varying precision. Algorithmic quality–complexity trade-offs are not new, but recent work has proposed tools for *automatically* transforming programs to take advantage of them. Transformations include removing portions of a program's dynamic execution (termed *code perforation*) [194], unsound parallelization of serial programs [131], eliminating synchronization in parallel programs [124, 134, 162, 164], identifying and adjusting parameters that control output quality [80], randomizing deterministic programs [132, 239], dynamically choosing between different programmer-provided implementations of the same specification [7, 8, 14, 62, 214, 222], and replacing subcomputations with invocations of a trained neural network [60].

Some work on algorithmic approximation targets specific hardware: notably, general-purpose GPUs [70, 176, 177, 185]. In a GPU setting, approximation

strategies benefit most by optimizing for memory bandwidth and control divergence.

Recently, a research direction has developed in *automated program repair* and other approaches to heuristically patching software according to programmer-specified criteria. These techniques are typically approximate in that they abandon a traditional compiler's goal of perfectly preserving the original program's semantics. Notably, Schulte et al. [188] propose to use program evolution to optimize for energy.

Precimonious [173] addresses the problem of choosing appropriate floating-point widths, which amount to a trade-off between numerical accuracy and space or operation cost. Similarly, STOKE's floating-point extension [187] synthesizes new versions of floating-point functions from scratch to meet different accuracy requirements with optimal efficiency.

Neural acceleration is a recent technique that treats code as a black box and transforms it into a neural network [40, 60, 121, 204]. It is, at its core, an algorithmic transformation, but it integrates tightly with hardware support: a digital accelerator [60], analog circuits [197], FPGAs [137], GPUs [70], or, recently, new analog substrates using resistive memory [105] or memristors [114]. See Section 1.4.2 for a more detailed overview of neural acceleration.

2.4 EXPLOITING RESILIENCE IN OTHER SYSTEMS

While architecture optimizations and program transformations dominate the field of proposed exploitations of approximate software, some recent work has explored the same trade-off in other components of computer systems.

Network communication, with its reliance on imperfect underlying channels, exhibits opportunities for fidelity trade-offs [84, 118, 189, 199]. Notably, SoftCast [84] transmits images and video by making the signal magnitude directly proportional to pixel luminance. BlinkDB, a recent instance of research on *approximate query answering*, is a database system that can respond to queries that include a required accuracy band on their output [3]. Uncertain<T> [21] and Lax [200] propose to expose the probabilistic behavior of sensors to programs. In a distributed system or a supercomputer, approximation techniques can eschew redundancy and recovery for efficiency [79].

2.5 LANGUAGES FOR EXPRESSING APPROXIMATION

Recently, language constructs that express and constrain approximation have become a focus in the programming-languages research community. Relax [97] is a language with ISA support for tolerating architectural faults in software. Rely [29] uses specifications that relate the reliability of the input to an approximate region of code to its outputs.

A related set of recent approximate-programming tools attempt to *adapt* a program to meet accuracy demands while using as few resources as possible. Chisel [130] is an extension to Rely that searches for the subset of operations in a program that can safely be made approximate. ExpAX [58] finds safe-to-

approximate operations automatically and uses a metaheuristic to find which subset of them to actually approximate.

Some other programming systems that focus on energy efficiency include approximation ideas: Eon [196] is a language for long-running embedded systems that can drop tasks when energy resources are low, and the Energy Types language [48] incorporates a variety of strategies for expressing energy requirements.

2.6 PROGRAMMER TOOLS

Aside from programming languages, separate programmer tools can help analyze and control the effects of approximation.

A quality-of-service profiler helps programmers identify parts of programs that may be good targets for approximation techniques [133]. Conversely, debugging tools can identify components where approximation is too aggressive [169]. Some verification tools and proof systems help the programmer prove relationships between the original program and a candidate relaxed version [27, 28, 30, 224].

As an alternative to statically bounding errors, dynamic techniques can monitor quality degradation at run time. The critical challenge for these techniques is balancing detection accuracy with the added cost, which takes away from the efficiency advantages of approximation. Some work has suggested that programmers can provide domain-specific checks on output quality [71, 169]. Recent work has explored automatic generation of error detectors [91]. A variety of techniques propose mechanisms for run-time or profiling feedback to adapt approximation parameters [8, 14, 80, 236].

2.7 PROBABILISTIC LANGUAGES

One specific research direction, *probabilistic programming languages*, focuses on expressing statistical models, especially for machine learning [18, 33, 69, 93, 94, 150, 184, 225]. The goal is to enable efficient statistical inference over arbitrary models written in the probabilistic programming language.

Earlier work examines the semantics of probabilistic behavior in more traditional programming models [95]. Similarly, the probability monad captures a variable's discrete probability distribution in functional programs [159]. Statistical model checking tools can analyze programs to prove statistical properties [100, 104]. Recently, Bornholt et al. [21] proposed a construct for explicitly representing probability distributions in a mainstream programming language.

2.8 ROBUSTNESS ANALYSIS

As the studies in Section 2.1 repeatedly find, error tolerance varies greatly in existing software, both within and between programs. Independent of approximate computing, programming-languages researchers have sought to identify and enhance error resilience properties.

SJava analyzes programs to prove that errors only temporarily disrupt the execution path of a program [54]. Program smoothing [36–38] and *robustification* [195] both find continuous, mathematical functions that resemble the input–output behavior of numerical programs. Auto-tuning approaches can help empirically identify error-resilient components [171]. Finally, Cong and Gururaj describe a technique for automatically distinguishing between critical and non-critical instructions for the purpose of selective fault tolerance [49].

Part II

PROGRAMMABLE APPROXIMATION

3

A SAFE AND GENERAL LANGUAGE ABSTRACTION

3.1 INTRODUCTION

Studies repeatedly show that approximate applications consist of both *critical* and *non-critical* components [96, 97, 103, 108, 110, 113, 133, 166, 194, 226]. For example, an image renderer can tolerate errors in the pixel data it outputs—a small number of erroneous pixels may be acceptable or even undetectable. However, an error in a jump table could lead to a crash, and even small errors in the image file format might make the output unreadable.

Distinguishing between the critical and non-critical portions of a program is difficult. Prior proposals have used annotations on code blocks (e.g., [97]) and data allocation sites (e.g., [113]). These annotations, however, do not offer any guarantee that the fundamental operation of the program is not compromised. In other words, these annotations are either *unsafe* and may lead to unacceptable program behavior or *need dynamic checks* that end up consuming energy. We need a way to allow programmers to compose programs from approximate and precise components safely. Moreover, we need to guarantee safety statically to avoid spending energy checking properties at runtime. The key insight in this work is the application of type-based information-flow tracking [174] ideas to address these problems.

This chapter proposes a model for approximate programming that is both *safe* and *general*. We use a type system that isolates the precise portion of the program from the approximate portion. The programmer must explicitly delineate flow from approximate data to precise data. The model is thus *safe* in that it guarantees precise computation unless given explicit programmer permission. Safety is statically enforced and no dynamic checks are required, minimizing the overheads imposed by the language.

We present EnerJ, a language for principled approximate computing. EnerJ extends Java with type qualifiers that distinguish between *approximate* and *precise* data types. Data annotated with the "approximate" qualifier can be stored approximately and computations involving it can be performed approximately. EnerJ also provides *endorsements*, which are programmer-specified points at which approximate-to-precise data flow may occur. The language supports programming constructs for algorithmic approximation, in which the programmer produces different implementations of functionality for approximate and precise data. We formalize a core of EnerJ and prove a non-interference property in the absence of endorsements.

Our programming model is *general* in that it unifies approximate data storage, approximate computation, and approximate algorithms. Programmers use a single abstraction to apply all three forms of approximation. The model is also *high-level and portable:* the implementation (compiler, runtime system, hardware) is entirely responsible for choosing the energy-saving mechanisms to employ and when to do so, guaranteeing correctness for precise data and "best effort" for the rest.

While EnerJ is designed to support general approximation strategies and therefore ensure full portability and backward-compatibility, we demonstrate its effectiveness using a proposed approximation-aware architecture with approximate memory and imprecise functional units. We have ported several applications to EnerJ to demonstrate that a small amount of annotation can allow a program to save a large amount of energy while not compromising quality of service significantly.

3.2 A TYPE SYSTEM FOR APPROXIMATE COMPUTATION

This section describes EnerJ's extensions to Java, which are based on a system of type qualifiers. We first describe the qualifiers themselves. We next explain how programmers precisely control when approximate data can affect precise state. We describe the implementation of approximate operations using overloading. We then discuss conditional statements and the prevention of implicit flows. Finally, we describe the type system's extension to object-oriented programming constructs and its interaction with Java arrays.

EnerJ implements these language constructs as backwards-compatible additions to Java extended with type annotations [56]. Table 1 summarizes our extensions and their concrete syntax.

3.2.1 *Type Annotations*

Every value in the program has an approximate or precise type. The programmer annotates types with the `@Approx` and `@Precise` qualifiers. Precise types are the default, so typically only `@Approx` is made explicit. It is illegal to assign an approximate-typed value into a precise-typed variable. Intuitively, this prevents direct flow of data from approximate to precise variables. For instance, the following assignment is illegal:

```
@Approx int a = ...;
int p; // precise by default
p = a; // illegal
```

Approximate-to-precise data flow is clearly undesirable, but it seems natural to allow flow in the opposite direction. For primitive Java types, we allow precise-to-approximate data flow via subtyping. Specifically, we make each precise primitive Java type a subtype of its approximate counterpart. This choice permits, for instance, the assignment `a = p;` in the above example.

For Java's reference (class) types, this subtyping relationship is unsound. The qualifier of a reference can influence the qualifiers of its fields (see Section 3.2.5),

Construct	Purpose	Section
`@Approx`, `@Precise`, `@Top`	Type annotations: qualify any type in the program. (Default is `@Precise`.)	3.2.1
`endorse(`e`)`	Cast an approximate value to its precise equivalent.	3.2.2
`@Approximable`	Class annotation: allow a class to have both precise and approximate instances.	3.2.5
`@Context`	Type annotation: in approximable class definitions, the precision of the type depends on the precision of the enclosing object.	3.2.5.1
`_APPROX`	Method naming convention: this implementation of the method may be invoked when the receiver has approximate type.	3.2.5.2

Table 1: Summary of EnerJ's language extensions.

so subtyping on mutable references is unsound for standard reasons. We find that this limitation is not cumbersome in practice.

We also introduce a `@Top` qualifier to denote the common supertype of `@Approx` and `@Precise` types.

SEMANTICS OF APPROXIMATION EnerJ takes an all-or-nothing approach to approximation. Precise values carry traditional guarantees of correctness; approximate values have no guarantees. The language achieves generality by leaving approximation patterns unspecified, but programmers can informally expect approximate data to be "mostly correct" and adhere to normal execution semantics except for occasional errors.

An approximate program's result quality is an orthogonal concern (see Section 1.2.2). Separate systems should complement EnerJ by tuning the frequency and intensity of errors in approximate data. The next two chapters in this part of the dissertation, on probability types and probabilistic assertions, propose systems that address the output-quality question.

3.2.2 *Endorsement*

Fully isolating approximate and precise parts of a program would likely not be very useful. Eventually a program needs to store data, transmit it, or present it to the programmer—at which point the program should begin behaving precisely. As a general pattern, programs we examined frequently had a phase of fault-tolerant computation followed by a phase of fault-sensitive reduction or output. For instance, one application consists of a resilient image manipulation phase followed by a critical checksum over the result (see Section 3.6.3). It is es-

sential that data be occasionally allowed to break the strict separation enforced by the type system.

We require the programmer to control explicitly when approximate data can affect precise state. To this end, we borrow the concept (and term) of *endorsement* from past work on information-flow control [10]. An explicit static function `endorse` allows the programmer to use approximate data as if it were precise. The function acts as a cast from any approximate type to its precise equivalent. Endorsements may have implicit runtime effects; they might, for example, copy values from approximate to precise memory.

The previous example can be made legal with an endorsement:

```
@Approx int a = ...;
int p; // precise by default
p = endorse(a); // legal
```

By inserting an endorsement, the programmer certifies that the approximate data is handled intelligently and will not cause undesired results in the precise part of the program.

3.2.3 *Approximate Operations*

The type system thus far provides a mechanism for approximating *storage*. Clearly, variables with approximate type may be located in unreliable memory modules. However, approximate *computation* requires additional features.

We introduce approximate computation by overloading operators and methods based on the type qualifiers. For instance, our language provides two signatures for the + operator on integers: one taking two precise integers and producing a precise integer and the other taking two approximate integers and producing an approximate integer. The latter may compute its result approximately and thus may run on low-power hardware. Programmers can extend this concept by overloading methods with qualified parameter types.

BIDIRECTIONAL TYPING The above approach occasionally applies precise operations where approximate operations would suffice. Consider the expression `a = b + c` where `a` is approximate but `b` and `c` are precise. Overloading selects precise addition even though the result will only be used approximately. It is possible to force an approximate operation by upcasting either operand to an approximate type, but we provide a slight optimization that avoids the need for additional annotation. EnerJ implements an extremely simple form of bidirectional type checking [45] that applies approximate arithmetic operators when the result type is approximate: on the right-hand side of assignment operators and in method arguments. We find that this small optimization makes it simpler to write approximate arithmetic expressions that include precise data.

3.2.4 *Control Flow*

To provide the desired property that information never flows from approximate to precise data, we must disallow *implicit flows* that occur via control flow. For example, the following program violates the desired isolation property:

```
@Approx int val = ...;
boolean flag; // precise
if (val == 5) { flag = true; } else { flag = false; }
```

Even though `flag` is precise and no endorsement is present, its value is affected by the approximate variable `val`.

EnerJ avoids this situation by prohibiting approximate values in conditions that affect control flow (such as `if` and `while` statements). In the above example, `val == 5` has approximate type because the approximate version of == must be used. Our language disallows this expression in the condition, though the programmer can work around this restriction using `if(endorse(val == 5))`.

This restriction is conservative: it prohibits approximate conditions even when the result can affect only approximate data. A more sophisticated approach would allow only approximate values to be produced in statements conditioned on approximate data. We find that our simpler approach is sufficient; endorsements allow the programmer to work around the restriction when needed.

3.2.5 *Objects*

EnerJ's type qualifiers are not limited to primitive types. Classes also support approximation. Clients of an *approximable* class can create precise and approximate instances of the class. The author of the class defines the meaning of approximation for the class. Approximable classes are distinguished by the `@Approximable` class annotation. Such a class exhibits qualifier polymorphism [67]: types within the class definition may depend on the qualifier of the instance.

Precise class types are not subtypes of their approximate counterparts, as is the case with primitive types (Section 3.2.1). Since Java uses references for all object types, this subtyping relationship would allow programs to create an approximate alias to a precise object; the object could then be mutated through that reference as if it were approximate. To avoid this source of unsoundness, we make object types invariant with respect to EnerJ's type qualifiers.

3.2.5.1 *Contextual Data Types*

The `@Context` qualifier is available in definitions of non-static members of approximable classes. The meaning of the qualifier depends on the precision of the instance of the enclosing class. (In terms of qualifier polymorphism, `@Context` refers to the class' qualifier parameter, which is determined by the qualifier placed on the instance.) Consider the following class definition:

```
@Approximable class IntPair {
  @Context int x;
  @Context int y;
```

```
  @Approx int numAdditions = 0;
  void addToBoth(@Context int amount) {
    x += amount;
    y += amount;
    numAdditions++;
  }
}
```

If `a` is an approximate instance of `IntPair`, then the three fields on the object, `a.x`, `a.y`, and `a.numAdditions`, are all of approximate integer type. However, if `p` is a precise instance of the class, then `p.x` and `p.y` are precise but `p.numAdditions` is still approximate. Furthermore, the argument to the invocation `p.addToBoth()` must be precise; the argument to `a.addToBoth()` may be approximate.

3.2.5.2 *Algorithmic Approximation*

Approximable classes may also specialize method definitions based on their qualifier. That is, the programmer can write two implementations: one to be called when the receiver has precise type and another that can be called when the receiver is approximate. Consider the following implementations of a mean calculation over a list of floats:

```
@Approximable class FloatSet {
  @Context float[] nums = ...;
  float mean() {
    float total = 0.0f;
    for (int i = 0; i < nums.length; ++i)
      total += nums[i];
    return total / nums.length;
  }
  @Approx float mean_APPROX() {
    @Approx float total = 0.0f;
    for (int i = 0; i < nums.length; i += 2)
      total += nums[i];
    return 2 * total / nums.length;
  }
}
```

EnerJ uses a naming convention, consisting of the `_APPROX` suffix, to distinguish methods overloaded on precision. The first implementation of `mean` is called when the receiver is precise. The second implementation calculates an approximation of the mean: it averages only half the numbers in the set. This implementation will be used for the invocation `s.mean()` where `s` is an approximate instance of `FloatSet`. Note that the compiler automatically decides which implementation of the method to invoke depending on the receiver type; the same invocation is used in either case.

It is the programmer's responsibility to ensure that the two implementations are similar enough that they can be safely substituted. This is important for backwards compatibility (a plain Java compiler will ignore the naming convention

and always use the precise version) and "best effort" (the implementation may use the precise version if energy is not constrained).

This facility makes it simple to couple algorithmic approximation with data approximation—a single annotation makes an instance use both approximate data (via `@Context`) and approximate code (via overloading).

3.2.6 *Arrays*

The programmer can declare arrays with approximate element types, but the array's length is always kept precise for memory safety. We find that programs often use large arrays of approximate primitive elements; in this case, the elements themselves are all approximated and only the length requires precise guarantees.

EnerJ prohibits approximate integers from being used as array subscripts. That is, in the expression `a[i]`, the value `i` must be precise. This makes it easier for the programmer to prevent out-of-bounds errors due to approximation.

3.3 FORMAL SEMANTICS

To study the formal semantics of EnerJ, we define the minimal language FEnerJ. The language is based on Featherweight Java [82] and adds precision qualifiers and state. The formal language omits EnerJ's endorsements and thus can guarantee isolation of approximate and precise program components. This isolation property suggests that, *in the absence of endorsement*, approximate data in an EnerJ program cannot affect precise state.

Appendix A formalizes this language and proves type soundness as well as a non-interference property that demonstrates the desired isolation of approximate and precise data.

3.3.1 *Programming Language*

Figure 1 presents the syntax of FEnerJ. Programs consist of a sequence of classes, a main class, and a main expression. Execution is modeled by instantiating the main class and then evaluating the main expression.

A class definition consists of a name, the name of the superclass, and field and method definitions. The `@Approximable` annotation is not modeled in FEnerJ; all classes in the formal language can have approximate and precise instances and `this` has `@Context` type. The annotation is required only in order to provide backward-compatibility with Java so that `this` in a non-approximable class has `@Precise` type.

We use C to range over class names and P for the names of primitive types. We define the precision qualifiers q as discussed in Section 3.2.1, but with the additional qualifier `lost`; this qualifier is used to express situations when context information is not expressible (i.e., lost). Types T include qualifiers.

Field declarations consist of the field type and name. Method declarations consist of the return type, method name, a sequence of parameter types and identifiers, the method precision, and the method body. We use the method precision

$$
\begin{array}{rcl}
Prg & ::= & \overline{Cls},\ C,\ e \\
Cls & ::= & \texttt{class}\ Cid\ \texttt{extends}\ C\ \{\ \overline{fd}\ \overline{md}\ \} \\
C & ::= & Cid \mid \texttt{Object} \\
P & ::= & \texttt{int} \mid \texttt{float} \\
q & ::= & \texttt{precise} \mid \texttt{approx} \mid \texttt{top} \mid \texttt{context} \mid \texttt{lost} \\
T & ::= & q\ C \mid q\ P \\
fd & ::= & T\ f; \\
md & ::= & T\ m(\overline{T\ pid})\ q\ \{\ e\ \} \\
x & ::= & pid \mid \texttt{this} \\
e & ::= & \texttt{null} \mid \mathcal{L} \mid x \mid \texttt{new}\ q\ C() \mid e.f \mid e_0.f{:=}e_1 \mid e_0.m(\overline{e}) \\
 & & \mid (q\ C)\ e \mid e_0 \oplus e_1 \mid \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\}
\end{array}
$$

f	field identifier	pid	parameter identifier
m	method identifier	Cid	class identifier

Figure 1: The syntax of the FEnerJ programming language. The symbol $\overline{A}$ denotes a sequence of elements A.

qualifier to denote overloading of the method based on the precision of the receiver as introduced in Section 3.2.5.2. Variables are either a parameter identifier or the special variable `this`, signifying the current object.

The language has the following expressions: the null literal, literals of the primitive types, reads of local variables, instantiation, field reads and writes, method calls, casts, binary primitive operations, and conditionals. We present the representative rules for field reads, field writes, and conditionals.

SUBTYPING Subtyping is defined using an ordering of the precision qualifiers and subclassing.

The following rules define the ordering of precision qualifiers:

$\boxed{q <:_{\mathrm{q}} q'}$ ordering of precision qualifiers

$$\frac{q \neq \texttt{top}}{q <:_{\mathrm{q}} \texttt{lost}} \qquad \frac{}{q <:_{\mathrm{q}} \texttt{top}} \qquad \frac{}{q <:_{\mathrm{q}} q}$$

Recall that `top` qualifies the common supertype of `precise` and `approx` types. Every qualifier other than `top` is below `lost`; every qualifier is below `top`; and the relation is reflexive. Note that the `precise` and `approx` qualifiers are not related.

Subclassing is the reflexive and transitive closure of the relation induced by the class declarations. Subtyping takes both ordering of precision qualifiers and subclassing into account. For primitive types, we additionally have that a precise type is a subtype of the approximate type as described in Section 3.2.1.

CONTEXT ADAPTATION We use *context adaptation* to replace the `context` qualifier when it appears in a field access or method invocation. Here the left-hand side of $\triangleright$ denotes the qualifier of the receiver expression; the right-hand side is the precision qualifier of the field or in the method signature.

$\boxed{q \triangleright q' = q''}$ combining two precision qualifiers

$$\frac{q' = \texttt{context} \wedge (q \in \{\texttt{approx}, \texttt{precise}, \texttt{context}\})}{q \triangleright q' = q}$$

$$\frac{q' = \texttt{context} \wedge (q \in \{\texttt{top}, \texttt{lost}\})}{q \triangleright q' = \texttt{lost}} \qquad \frac{q' \neq \texttt{context}}{q \triangleright q' = q'}$$

Note that `context` adapts to `lost` when the left-hand-side qualifier is `top` because the appropriate qualifier cannot be determined.

We additionally define $\triangleright$ to take a type as the right-hand side; this adapts the precision qualifier of the type.

We define partial look-up functions FType and MSig that determine the field type and method signature for a given field/method in an access or invocation. Note that these use the adaptation rules described above.

TYPE RULES The static type environment ${}^{s}T$ maps local variables to their declared types.

Given a static environment, expressions are typed as follows:

$\boxed{{}^{s}T \vdash e : T}$ expression typing

$$\frac{{}^{s}T \vdash e_0 : q\,C \qquad \text{FType}(q\,C, f) = T}{{}^{s}T \vdash e_0.f : T}$$

$$\frac{{}^{s}T \vdash e_0 : q\,C \qquad \text{FType}(q\,C, f) = T \qquad \texttt{lost} \notin T \qquad {}^{s}T \vdash e_1 : T}{{}^{s}T \vdash e_0.f := e_1 : T}$$

$$\frac{{}^{s}T \vdash e_0 : \texttt{precise}\,P \qquad {}^{s}T \vdash e_1 : T \qquad {}^{s}T \vdash e_2 : T}{{}^{s}T \vdash \texttt{if}(e_0)\,\{e_1\}\,\texttt{else}\,\{e_2\} : T}$$

A field read determines the type of the receiver expression and then uses FType to determine the adapted type of the field.

A field write similarly determines the adapted type of the field and checks that the right-hand side has an appropriate type. In addition, we ensure that the adaptation of the declared field type did not lose precision information. Notice that we can read a field with lost precision information, but that it would be unsound to allow the update of such a field.

Finally, for the conditional expression, we ensure that the condition is of a precise primitive type and that there is a common type T that can be assigned to both subexpressions.

3.3.2 *Operational Semantics*

The runtime system of FEnerJ models the heap h as a mapping from addresses ι to objects, where objects are a pair of the runtime type T and the field values v of the object. The runtime environment ${}^{r}T$ maps local variables x to values v.

The runtime system of FEnerJ defines a standard big-step operational semantics:

$\boxed{{}^{r}T \vdash h, e \rightsquigarrow h', v}$ big-step operational semantics

$$\frac{{}^{r}T \vdash h, e_0 \rightsquigarrow h', \iota_0 \qquad h'(\iota_0.f) = v}{{}^{r}T \vdash h, e_0.f \rightsquigarrow h', v}$$

$$\frac{{}^rT \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \qquad {}^rT \vdash h_0, e_1 \rightsquigarrow h_1, v \qquad h_1[\iota_0.f := v] = h'}{{}^rT \vdash h, e_0.f := e_1 \rightsquigarrow h', v}$$

$$\frac{{}^rT \vdash h, e_0 \rightsquigarrow h_0, (q, {}^r\mathcal{L}) \qquad {}^r\mathcal{L} \neq 0 \qquad {}^rT \vdash h_0, e_1 \rightsquigarrow h', v}{{}^rT \vdash h, \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\} \rightsquigarrow h', v}$$

$$\frac{{}^rT \vdash h, e_0 \rightsquigarrow h_0, (q, 0) \qquad {}^rT \vdash h_0, e_2 \rightsquigarrow h', v}{{}^rT \vdash h, \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\} \rightsquigarrow h', v}$$

These rules reflect precise execution with conventional precision guarantees. To model computation on an execution substrate that supports approximation, the following rule could be introduced:

$$\frac{{}^rT \vdash h, e \rightsquigarrow h', v \qquad h' \cong \tilde{h}' \qquad v \cong \tilde{v}}{{}^rT \vdash h, e \rightsquigarrow \tilde{h}', \tilde{v}}$$

We use $\cong$ to denote an equality that disregards approximate values for comparing heaps and values with identical types. The rule permits any approximate value in the heap to be replaced with any other value of the same type and any expression producing a value of an approximate type to produce any other value of that type instead. This rule reflects EnerJ's lack of guarantees for approximate values.

3.3.3 *Properties*

We prove two properties about FEnerJ: type soundness and non-interference. Appendix A proves these theorems.

The usual type soundness property expresses that, for a well-typed program and corresponding static and runtime environments, we know that (1) the runtime environment after evaluating the expression is still well formed, and (2) a static type that can be assigned to the expression can also be assigned to the value that is the result of evaluating the expression. Formally:

$$\left.\begin{array}{l} \vdash Prg\ \texttt{OK} \wedge \vdash h, {}^rT : {}^sT \\ {}^sT \vdash e : T \\ {}^rT \vdash h, e \rightsquigarrow h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} \vdash h', {}^rT : {}^sT \\ h', {}^rT(\texttt{this}) \vdash v : T \end{array}\right.$$

The proof is by rule induction over the operational semantics; in separate lemmas we formalize that the context adaptation operation $\triangleright$ is sound.

The non-interference property of FEnerJ guarantees that approximate computations do not influence precise values. Specifically, changing approximate values in the heap or runtime environment does not change the precise parts of the heap or the result of the computation. More formally, we show:

$$\left.\begin{array}{l} \vdash Prg\ \texttt{OK} \wedge \vdash h, {}^rT : {}^sT \\ {}^sT \vdash e : T \\ {}^rT \vdash h, e \rightsquigarrow h', v \\ h \cong \tilde{h} \wedge {}^rT \cong {}^r\tilde{T} \\ \vdash \tilde{h}, {}^r\tilde{T} : {}^sT \end{array}\right\} \Longrightarrow \left\{\begin{array}{l} {}^r\tilde{T} \vdash \tilde{h}, e \rightarrow \tilde{h}', \tilde{v} \\ h' \cong \tilde{h}' \\ v \cong \tilde{v} \end{array}\right.$$

For the proof of this property we introduced a *checked* operational semantics that ensures in every evaluation step that the precise and approximate parts are separated. We can then show that the evaluation of a well-typed expression always passes the checked semantics of the programming language.

3.4 EXECUTION MODEL

While an EnerJ program distinguishes abstractly between approximate and precise data, it does not define the particular approximation strategies that are applied to the program. (In fact, one valid execution is to ignore all annotations and execute the code as plain Java.) An approximation-aware execution substrate is needed to take advantage of EnerJ's annotations. We examine approximation mechanisms at the architecture level that work at granularity of individual instructions and individual memory locations [59, 213]. This section describes our hardware model, the ISA extensions used for approximation, and how the extensions enable energy savings. The Truffle paper [59] explores the ISA design and microarchitectural mechanisms for approximation in more detail.

As a complement to the approximate hardware considered here, a compiler or runtime system on top of commodity hardware can also offer approximate execution features: lower floating point precision, elision of memory operations, etc. (Algorithmic approximation, from Section 3.2.5, is independent of the execution substrate.) The ACCEPT compiler infrastructure in Chapter 7 exploits this category of approximations using an annotation language similar to EnerJ.

3.4.1 *Approximation-Aware ISA Extensions*

We want to leverage both approximate *storage* and approximate *operations*. Our hardware model offers approximate storage in the form of unreliable registers, data caches, and main memory. Approximate and precise registers are distinguished based on the register number. Approximate data stored in memory is distinguished from precise data based on address; regions of physical memory are marked as approximate and, when accessed, are stored in approximate portions of the data cache. For approximate operations, we assume specific instructions for approximate integer ALU operations as well as approximate floating point operations. Approximate instructions can use special functional units that perform approximate operations. Figure 2 summarizes our assumed hardware model.

An instruction stream may have a mix of approximate and precise instructions. Precise instructions have the same guarantees as instructions in today's ISAs. Note that an approximate instruction is simply a "hint" to the architecture that it may apply a variety of energy-saving approximations when executing the given instruction. The particular approximations employed by a given architecture are not exposed to the program; a processor supporting no approximations just executes approximate instructions precisely and saves no energy. An approximation-aware ISA thus allows a single binary to benefit from new approximations as they are implemented in future microarchitectures.

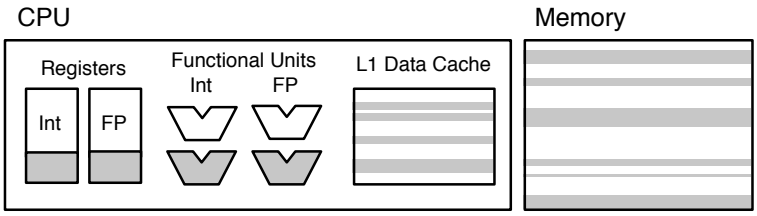


Figure 2: Hardware model assumed in our system. Shaded areas indicate components that support approximation. Registers and the data cache have SRAM storage cells that can be made approximate by decreasing supply voltage. Functional units support approximation via supply voltage reduction. Floating point functional units also support approximation via smaller mantissas. Main memory (DRAM) supports approximation by reducing refresh rate.

LAYOUT OF APPROXIMATE DATA Our hardware model supports approximate memory data at a cache line granularity, in which software can configure any line as approximate. This can be supported by having a bit per line in each page that indicates whether the corresponding line is approximate. Based on that bit, a cache controller determines the supply voltage of a line (lower for approximate lines), and the refresh rate for regions of DRAM. This bitmap needs to be kept precise. With a typical cache line size of 64 bytes, this is less than 0.2% overhead. Note that both selective supply voltage for caches [65] and selective refresh rate for DRAM [68] are hardware techniques that have been proposed in the past.

Setting approximation on a cache line basis requires the runtime system to segregate approximate and precise data in different cache lines. We propose the following simple technique for laying out objects with both approximate and precise fields. First, lay out the precise portion of the object (including the `vtable` pointer) contiguously. Each cache line containing at least one precise field is marked as precise. Then, lay out the approximate fields after the end of the precise data. Some of this data may be placed in a precise line (that is, a line containing some precise data already); in this case, the approximate data stays precise and saves no memory energy. (Note that wasting space in the precise line in order to place the data in an approximate line would use more memory and thus more energy.) The remaining approximate fields that do not fit in the last precise line can be placed in approximate lines.

Fields in superclasses may not be reordered in subclasses. Thus, a subclass of a class with approximate data may waste space in an approximate line in order to place precise fields of the subclass in a precise line.

While we simulate the artifacts of this layout scheme for our evaluation, a finer granularity of approximate memory storage would mitigate or eliminate the resulting loss of approximation. More sophisticated layout algorithms could also improve energy savings; this is a target for compile-time optimization. Note that even if an approximate field ends up stored in precise memory, it will still be loaded into approximate registers and be subject to approximate operations and algorithms.

The layout problem is much simpler for arrays of approximate primitive types. The first line, which contains the length and type information, must be precise, with all remaining lines approximate.

3.4.2 *Hardware Techniques for Saving Energy*

There are many strategies for saving energy with approximate storage and data operations. This section discusses some of the techniques explored in prior research. We assume these techniques in our simulations, which we describe later. The techniques are summarized in Table 2.

VOLTAGE SCALING Aggressive voltage scaling can result in over 30% energy reduction with $\sim 1\%$ error rate [55] and 22% reduction with $\sim 0.01\%$ error rate. Recent work [97] proposed to expose the errors to applications that can tolerate it and saw similar results. In our model, we assume aggressive voltage scaling for the processor units executing approximate instructions, including integer and floating-point operations. As for an error model, the choices are single bit flip, last value, and random value. We consider all three but our evaluation mainly depicts the random-value assumption, which is the most realistic.

FLOATING POINT OPERATION WIDTH A direct approach to approximate arithmetic operations on floating point values is to ignore part of the mantissa in the operands. As observed in [210], many applications do not need the full mantissa. According to their model, a floating-point multiplier using 8-bit mantissas uses 78% less energy per operation than a full 24-bit multiplier.

DRAM REFRESH RATE Reducing the refresh rate of dynamic RAM leads to potential data decay but can substantially reduce power consumption with a low error rate. As proposed by Liu et al. [113], an approximation-aware DRAM system might reduce the refresh rate on lines containing approximate data. As in that work, we assume that reducing the refresh rate to 1 Hz reduces power by about 20%. In a study performed by Bhalodia [17], a DRAM cell not refreshed for 10 seconds experiences a failure with per-bit probability approximately 10^{-5}. We conservatively assume this error rate for the reduced refresh rate of 1 Hz.

SRAM SUPPLY VOLTAGE Registers and data caches in modern CPUs consist of static RAM (SRAM) cells. Reducing the supply voltage to SRAM cells lowers the leakage current of the cells but decreases the data integrity [65]. As examined by Kumar [98], these errors are dominated by *read upsets* and *write failures*, which occur when a bit is read or written. A read upset occurs when the stored bit is flipped while it is read; a write failure occurs when the wrong bit is written. Reducing SRAM supply voltage by 80% results in read upset and write failure probabilities of $10^{-7.4}$ and $10^{-4.94}$ respectively. *Soft failures*, bit flips in stored data due to cosmic rays and other events, are comparatively rare and depend less on the supply voltage.

	Mild	Medium	Aggressive
DRAM refresh: per-second bit flip probability	10^{-9}	10^{-5}	10^{-3}
Memory power saved	17%	22%	24%
SRAM read upset probability	$10^{-16.7}$	$10^{-7.4}$	10^{-3}
SRAM write failure probability	$10^{-5.59}$	$10^{-4.94}$	10^{-3}
Supply power saved	70%	80%	90%*
`float` mantissa bits	16	8	4
`double` mantissa bits	32	16	8
Energy saved per operation	32%	78%	85%*
Arithmetic timing error probability	10^{-6}	10^{-4}	10^{-2}
Energy saved per operation	12%*	22%	30%

Table 2: Approximation strategies simulated in our evaluation. Numbers marked with * are educated guesses by the authors; the others are taken from the sources described in Section 3.4.2. Note that all values for the Medium level are taken from the literature.

Section 3.5.4 describes the model we use to combine these various potential energy savings into an overall CPU/memory system energy reduction. To put the potential energy savings in perspective, according to recent studies [61, 119], the CPU and memory together account for well over 50% of the overall system power in servers as well as notebooks. In a smartphone, CPU and memory account for about 20% and the radio typically close to 50% of the overall power [31].

3.5 IMPLEMENTATION

We implement EnerJ as an extension to the Java programming language based on the pluggable type mechanism proposed by Papi et al. [148]. EnerJ is implemented using the Checker Framework[1] infrastructure, which builds on the JSR 308[2] extension to Java's annotation facility. JSR 308 permits annotations on any explicit type in the program. The EnerJ type checker extends the rules from Section 3.3 to all of Java, including arrays and generics. We also implement a simulation infrastructure that emulates an approximate computing architecture as described in Section 3.4. [3]

3.5.1 *Type Checker*

EnerJ provides the type qualifiers listed in Table 1—`@Approx`, `@Precise`, `@Top`, and `@Context`—as JSR 308 type annotations. The default type qualifier for unannotated types is `@Precise`, meaning that any Java program may be compiled as

1 `http://types.cs.washington.edu/checker-framework/`
2 `http://types.cs.washington.edu/jsr308/`
3 The EnerJ type checker and simulator are available online: `http://sampa.cs.washington.edu/research/approximation/enerj.html`

an EnerJ program with no change in semantics. The programmer can add approximations to the program incrementally.

While reference types may be annotated as `@Approx`, this only affects the meaning of `@Context` annotations in the class definition and method binding on the receiver. Our implementation never approximates pointers.

3.5.2 *Simulator*

To evaluate our system, we implement a compiler and runtime system that executes EnerJ code as if it were running on an approximation-aware architecture as described in Section 3.4. We instrument method calls, object creation and destruction, arithmetic operators, and memory accesses to collect statistics and inject faults. The runtime system is implemented as a Java library and is invoked by the instrumentation calls. It records memory-footprint and arithmetic-operation statistics while simultaneously injecting transient faults to emulate approximate execution.

To avoid spurious errors due to approximation, our simulated approximate functional units never raise divide-by-zero exceptions. Approximate floating-point division by zero returns the NaN value; approximate integer divide-by-zero returns zero.

3.5.3 *Approximations*

Our simulator implements the set of approximation techniques enumerated in Section 3.4.2. Table 2 summarizes the approximations used, their associated error probabilities, and their estimated energy savings.

Floating-point bit-width reduction is performed when executing Java's arithmetic operators on operands that are approximate `float` and `double` values. SRAM read upsets and write failures are simulated by flipping each bit read or written with a constant probability. For DRAM refresh reduction, every bit also has an independent probability of inversion; here, the probability is proportional to the amount of time since the last access to the bit.

For the purposes of our evaluation, we distinguish SRAM and DRAM data using the following rough approximation: data on the heap is considered to be stored in DRAM; stack data is considered SRAM. Future evaluations not constrained by the abstraction of the JVM could explore a more nuanced model.

3.5.4 *Energy Model*

To summarize the effectiveness of EnerJ's energy-saving properties, we estimate the potential overall savings of the processor/memory system when executing each benchmark approximately. To do so, we consider a simplified model with three components to the system's energy consumption: instruction execution, SRAM storage (registers and cache), and DRAM storage. Our model omits overheads of implementing or switching to approximate hardware. For example, we do not model any latency in scaling the voltage on the logic units. For this reason,

our results can be considered optimistic; the Truffle paper [59] models approximate hardware in more detail.

To estimate the savings for instruction execution, we assign abstract energy units to arithmetic operations. Integer operations take 37 units and floating point operations take 40 units; of each of these, 22 units are consumed by the instruction fetch and decode stage and may not be reduced by approximation strategies. These estimations are based on three studies of architectural power consumption [25, 107, 140]. We calculate energy savings in instruction execution by scaling the non-fetch, non-decode component of integer and floating-point instructions.

We assume that SRAM storage and instructions that access it account for approximately 35% of the microarchitecture's power consumption; instruction execution logic consumes the remainder. To compute the total CPU power savings, then, we scale the savings from SRAM storage by 0.35 and the instruction power savings, described above, by 0.65.

Finally, we add the savings from DRAM storage to get an energy number for the entire processor/memory system. For this, we consider a server-like setting, where DRAM accounts for 45% of the power and CPU 55% [61]. Note that in a mobile setting, memory consumes only 25% of power so power savings in the CPU will be more important [31].

3.6 RESULTS

We evaluate EnerJ by annotating a variety of existing Java programs. Table 3 describes the applications we used; they have been selected to be relevant in both mobile and server settings.

APPLICATIONS We evaluate the FPU-heavy kernels of the SciMark2 benchmark suite to reflect scientific workloads.[4] ZXing is a bar code reader library targeted for mobile devices based on the Android operating system.[5] Our workload decodes QR Code two-dimensional bar code images. jMonkeyEngine is a 2D and 3D game engine for both desktop and mobile environments.[6] We run a workload that consists of many 3D triangle intersection problems, an algorithm frequently used for collision detection in games.

ImageJ is an image-manipulation program; our workload executes a flood fill operation.[7] This workload was selected as representative of error-resilient algorithms with primarily integer—rather than floating point—data. Because the code already includes extensive safety precautions such as bounds checking, our annotation for ImageJ is extremely aggressive: even pixel coordinates are marked as approximate. Raytracer is a simple 3D renderer; our workload executes ray plane intersection on a simple scene.

4 SciMark2: `http://math.nist.gov/scimark2/`
5 ZXing: `http://code.google.com/p/zxing/`
6 jMonkeyEngine: `http://www.jmonkeyengine.com/`
7 ImageJ: `http://rsbweb.nih.gov/ij/`

Application	Description	Error metric	Lines of Code	Proportion FP	Total Decls	Annotated Decls	Endorsements
FFT		Mean entry difference	168	38.2%	85	33%	2
SOR		Mean entry difference	36	55.2%	28	25%	0
MonteCarlo	SciMark2 kernels	Normalized difference	59	22.9%	15	20%	1
SparseMatMult		Mean normalized difference	38	39.7%	29	14%	0
LU		Mean entry difference	283	31.4%	150	23%	3
ZXing	Smartphone bar code decoder	1 if incorrect, 0 if correct	26171	1.7%	11506	4%	247
jMonkeyEngine	Mobile/desktop game engine	Fraction of correct decisions normalized to 0.5	5962	44.3%	2104	19%	63
ImageJ	Raster image manipulation	Mean pixel difference	156	0.0%	118	34%	18
Raytracer	3D image renderer	Mean pixel difference	174	68.4%	92	33%	10

Table 3: Applications used in our evaluation, application-specific metrics for quality of service, and metrics of annotation density. "Proportion FP" indicates the percentage of dynamic arithmetic instructions observed that were floating-point (as opposed to integer) operations.

ANNOTATION APPROACH We annotated each application manually. While many possible annotations exist for a given program, we attempted to strike a balance between reliability and energy savings. As a rule, however, we attempted to annotate the programs in a way that never causes them to crash (or throw an unhandled exception); it is important to show that EnerJ allows programmers to write approximate programs that never fail catastrophically. In our experiments, each benchmark produces an output on every run. This is in contrast to approximation techniques that do not attempt to prevent crashes [108, 113, 226]. Naturally, we focused our effort on code where most of the time is spent.

Three students involved in the project ported the applications used in our evaluation. In every case, we were unfamiliar with the codebase beforehand, so our annotations did not depend on extensive domain knowledge. The annotations were not labor intensive.

QUALITY METRICS For each application, we measure the degradation in output quality of approximate executions with respect to the precise executions. To do so, we define application-specific quality metrics following the principle in Section 1.2.1. The third column in Table 3 shows our metric for each application.

Output error ranges from 0 (indicating output identical to the precise version) to 1 (indicating completely meaningless output). For applications that produce lists of numbers (e.g., SparseMatMult's output matrix), we compute the error as the mean entry-wise difference between the pristine output and the degraded output. Each numerical difference is limited by 1, so if an entry in the output is `NaN`, that entry contributes an error of 1. For benchmarks where the output is not numeric (i.e., ZXing, which outputs a string), the error is 0 when the output is correct and 1 otherwise.

3.6.1 *Energy Savings*

Figure 3 divides the execution of each benchmark into DRAM storage, SRAM storage, integer operations, and FP operations and shows what fraction of each was approximated. For many of the FP-centric applications we simulated, including the jMonkeyEngine and Raytracer as well as most of the SciMark applications, nearly all of the floating point operations were approximate. This reflects the inherent imprecision of FP representations; many FP-dominated algorithms are inherently resilient to rounding effects. The same applications typically exhibit very little or no approximate integer operations. The frequency of loop induction variable increments and other precise control-flow code limits our ability to approximate integer computation. ImageJ is the only exception with a significant fraction of integer approximation; this is because it uses integers to represent pixel values, which are amenable to approximation.

We quantify DRAM and SRAM approximation using the proportion of the total byte-seconds in the execution. The data shows that both storage types are frequently used in approximate mode. Many applications have DRAM approximation rates of 80% or higher; it is common to store large data structures (often

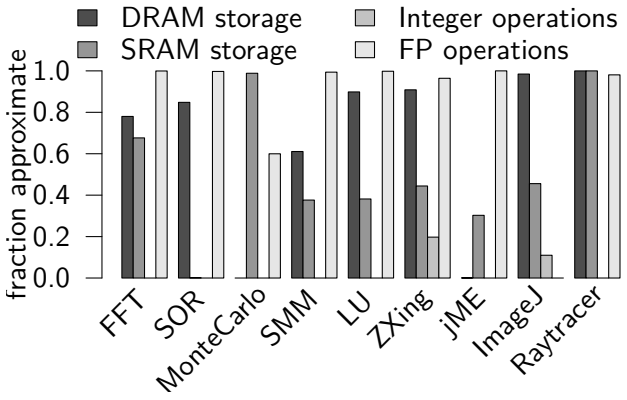


Figure 3: Proportion of approximate storage and computation in each benchmark. For storage (SRAM and DRAM) measurements, the bars show the fraction of byte-seconds used in storing approximate data. For functional unit operations, we show the fraction of dynamic operations that were executed approximately.

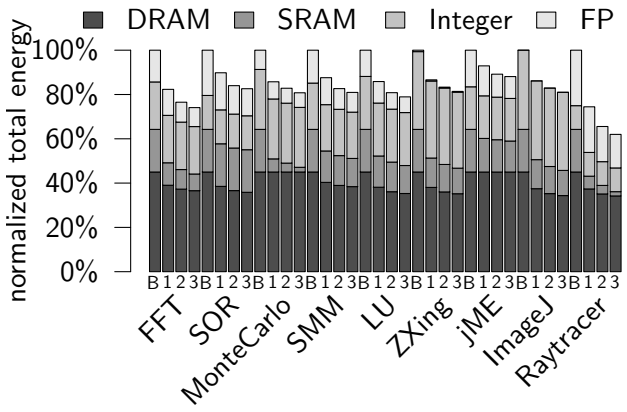


Figure 4: Estimated CPU/memory system energy consumed for each benchmark. The bar labeled "B" represents the baseline value: the energy consumption for the program running without approximation. The numbered bars correspond to the Mild, Medium, and Aggressive configurations in Table 2.

arrays) that can tolerate approximation. MonteCarlo and jMonkeyEngine, in contrast, have very little approximate DRAM data; this is because both applications keep their principal data in local variables (i.e., on the stack).

The results depicted assume approximation at the granularity of a 64-byte cache line. As Section 3.4.1 discusses, this reduces the number of object fields that can be stored approximately. The impact of this constraint on our results is small, in part because much of the approximate data is in large arrays. Finer-grain approximate memory could yield a higher proportion of approximate storage.

To give a sense of the energy savings afforded by our proposed approximation strategies, we translate the rates of approximation depicted above into an estimated energy consumption. Figure 4 shows the estimated energy consumption for each benchmark running on approximate hardware relative to fully precise execution. The energy calculation is based on the model described in Section 3.5.4. These simulations apply all of the approximation strategies described in Section 3.4.2 simultaneously at their three levels of aggressiveness. As expected, the total energy saved increases both with the amount of approximation in the application (depicted in Figure 3) and with the aggressiveness of approximation used.

Overall, we observe energy savings from 7% (SOR in the Mild configuration) to 38% (Raytracer in the Aggressive configuration). The three levels of approximation do not vary greatly in the amount of energy saved—the three configurations yield average energy savings of 14%, 19%, and 21% respectively. The majority of the energy savings come from the transition from zero approximation to mild approximation. As discussed in the next section, the least aggressive configuration results in very small losses in output fidelity across all applications studied.

The fifth column of Table 3 shows the proportion of floating point arithmetic in each application. In general, applications with principally integer computation (e.g., ZXing and ImageJ) exhibit less opportunity for approximation than do floating-point applications (e.g., Raytracer). Not only do floating-point instructions offer more energy savings potential in our model, but applications that use them are typically resilient to their inherent imprecision.

3.6.2 *Result Quality Trade-off*

Figure 5 presents the sensitivity of each annotated application to the full suite of approximations explored. This output quality reduction is the trade-off for the energy savings shown in Figure 4.

While most applications show negligible error for the Mild level of approximation, applications' sensitivity to error varies greatly for the Medium and Aggressive configurations. Notably, MonteCarlo, SparseMatMult, ImageJ, and Raytracer exhibit very little output degradation under any configuration whereas FFT and SOR lose significant output fidelity even under the Medium configuration. This variation suggests that an approximate execution substrate for EnerJ could benefit from tuning to the characteristics of each application, either

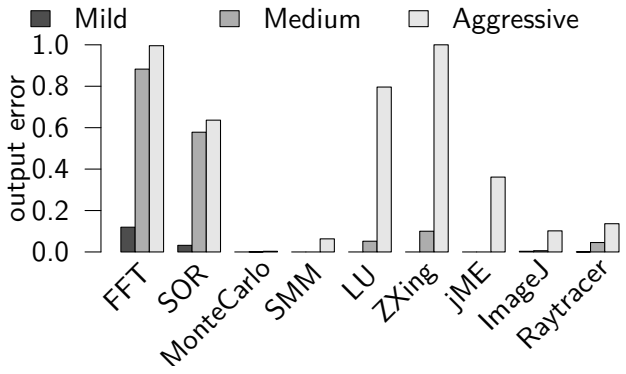


Figure 5: Output error for three different levels of approximation varied together. Each bar represents the mean error over 20 runs.

offline via profiling or online via continuous quality measurement as in Green [14]. However, even the conservative Mild configuration offers significant energy savings.

Qualitatively, the approximated applications exhibit gradual degradation of perceptible output quality. For instance, Raytracer always outputs an image resembling its precise output, but the amount of random pixel "noise" increases with the aggressiveness of approximation. Under the Mild configuration, it is difficult to distinguish the approximated image from the precise one.

We also measured the relative impact of various approximation strategies by running our benchmark suite with each optimization enabled in isolation. The DRAM errors we modeled have a nearly negligible impact on application output; floating-point bit width reduction similarly results in at most 12% quality loss in the Aggressive configuration. SRAM write errors are much more detrimental to output quality than read upsets. Functional unit voltage reduction had the greatest impact on correctness. We considered three possibilities for error modes in functional units: the output has a single bit flip; the last value computed is returned; or a random value is returned. The former two models resulted in significantly less quality loss than the random-value model (25% vs. 40%). However, we consider the random-value model to be the most realistic, so we use it for the results shown in Figure 5.

3.6.3 *Annotation Effort*

Table 3 lists the number of qualifiers and endorsements used in our annotations. Only a fraction of the types in each program must be annotated: at most 34% of the possible annotation sites are used. Note that most of the applications are short programs implementing a single algorithm (the table shows the lines of code in each program). Our largest application, ZXing, has about 26,000 lines of code and only 4% of its declarations are annotated. These rates suggest that the principal data amenable to approximation is concentrated in a small portion

of the code, even though approximate data typically dominates the program's dynamic behavior.

Endorsements are also rare, even though our system requires one for every approximate condition value. The outlier is ZXing, which exhibits a higher number of endorsements due to its frequency of approximate conditions. This is because ZXing's control flow frequently depends on whether a particular pixel is black.

Qualitatively, we found EnerJ's annotations easy to insert. The programmer can typically select a small set of data to approximate and then, guided by type checking errors, ascertain associated data that must also be marked as approximate. The requirements that conditions and array indices be precise helped quickly distinguish data that was likely to be sensitive to error. In some cases, such as jMonkeyEngine and Raytracer, annotation was so straightforward that it could have been largely automated: for certain methods, every `float` declaration was replaced indiscriminately with an `@Approx float` declaration.

Classes that closely represent data are perfect candidates for `@Approximable` annotations. For instance, ZXing contains `BitArray` and `BitMatrix` classes that are thin wrappers over binary data. It is useful to have approximate bit matrices in some settings (e.g., during image processing) but precise matrices in other settings (e.g., in checksum calculation). Similarly, the jMonkeyEngine benchmark uses a `Vector3f` class for much of its computation, which we marked as approximable. In this setting, approximate vector declarations:

```
@Approx Vector3f v;
```

are syntactically identical to approximate primitive-value declarations:

```
@Approx int i;
```

We found that the `@Context` annotation helped us to approach program annotation incrementally. A commonly-used class that is a target for approximation can be marked with `@Context` members instead of `@Approx` members. This way, all the clients of the class continue to see precise members and no additional annotation on them is immediately necessary. The programmer can then update the clients individually to use the approximate version of the class rather than addressing the whole program at once.

An opportunity for algorithmic approximation also arose in ZXing. The approximable class `BitArray` contains a method `isRange` that takes two indices and determines whether all the bits between the two indices are set. We implemented an approximate version of the method that checks only some of the bits in the range by skipping some loop iterations. We believe that application domain experts would use algorithmic approximation more frequently.

In one case, we found it convenient to introduce a slight change to increase the fault tolerance of code dealing with approximate data. ZXing has a principally floating-point phase that performs an image perspective transform. If the transform tried to access a coordinate outside of the image bounds, ZXing would catch the `ArrayIndexOutOfBoundsException` and print a message saying that the image transform failed. We modified the algorithm to silently return a white pixel in this case. The result was that the image transform became more resilient to transient faults in the transformation coordinates. We marked these coordinates as approximate and then endorsed them at the point they are used

as array indices. In no case, however, does an application as we annotated it do *more* computation than the pristine version.

3.7 DISCUSSION

EnerJ is a language for enforcing *safety* in approximate programing. The key observation is that approximate programs tend to intermix error-resilient and error-vulnerable work within the same program. The former makes up the bulk of the computation and data, while the latter provides critical structure and control. EnerJ's brand of approximate safety protects the control components while allowing errors in most of the program. It borrows ideas from information-flow tracking for enforcing security to isolate the critical data from the corrupting effects of approximation.

The next two chapters shift focus from enforcing safety to controlling quality. The systems described next all benefit from the separation of concerns that EnerJ offers: they only need to analyze the approximate component of the program. EnerJ's focus on simpler safety properties makes it a foundation for the more sophisticated abstractions necessary for reasoning about quality.

4

PROBABILITY TYPES

4.1 INTRODUCTION

In approximate computing, we recognize that not every operation in a program needs the same level of accuracy. But while programmers may know which outputs can withstand occasional errors, it is tedious and error-prone to compose individual approximate operations to achieve the desired result. Fine-grained reliability choices can have subtle and far-reaching implications for the efficiency and reliability of a whole computation. Programmers need a way to easily maximize the efficiency of fine-grained operations while controlling the impact of unreliability on overall accuracy properties.

The EnerJ language in the previous chapter demonstrates that a type system can ensure that approximation never corrupts essential program state [180]. But as the *safety vs. quality* principle from Section 1.2.2 emphasizes, safety properties are only part of approximate computing's programmability challenge. More nuanced *quality* properties dictate how much an output can deviate from its precise equivalent.

This chapter presents DECAF (**D**ECAF, an **E**nergy-aware **C**ompiler to make **A**pproximation **F**lexible), a type-based approach to controlling quality in approximate programs. DECAF's goal is to let programmers specify important quality constraints while leaving the details to the compiler. Its design explores five critical research questions in approximate programming:

How can programmers effectively use complex hardware with many available degrees of approximation? Current languages for approximate programming assume that approximation will be an all-or-nothing affair [29, 130, 180]. But recent work has suggested that more sophisticated architectures, supporting multiple levels of reliability, are a better match for application demands [213]. DECAF is a language abstraction that shields the programmer from reasoning about individual operators to compose reliable software. Its probability type system constrains the likelihood that any expression in the relaxed program differs from its equivalent in a reliable execution.

How can automated tuning interact with programmer control? Compiler assistance can help reduce the annotation burden of approximate programming [58, 130, 176]. But fully automated approaches impede programmers from bringing intuition to bear when fine-grained control is more appropriate. DECAF's solver-aided type inference adds flexibility: programmers add accuracy requirements where they are most crucial and omit them where they can be implied.

Programmers in early development phases can opt to rely more heavily on inference, while later-stage optimization work can exert total control over any type in the program.

When static reasoning is insufficient, how can a program safely opt into dynamic tracking? Purely static systems for reasoning about approximation can be overly conservative when control flow is dynamic [29] while dynamic monitoring incurs run-time overhead [169]. DECAF's optional dynamic typing interoperates with its static system to limit overheads to code where static constraints are insufficient. We prove a soundness theorem that shows that DECAF's hybrid system of static types, dynamic tracking, and run-time checks conservatively bounds the chance of errors.

How should compilers reuse approximate code in contexts with different accuracy demands? An approximate program can invoke a single function in some contexts that permit more approximation and others with stricter reliability requirements. A fixed degree of "aggressiveness" for a function's approximation can therefore be conservative. DECAF's type inference can automatically synthesize specialized versions of approximate functions at multiple levels of reliability.

What do language-level constraints imply for the design of approximate hardware? Approximate hardware designs remain in the research stage. As designs mature, architectures will need to choose approximation parameters that fit a wide range of approximate software. We use DECAF's architecture-aware tuning to examine the implications of programs' language-level constraints on approximate hardware. Our evaluation finds that using a solver to optimize for a hardware configuration can lead to significant gains over a hardware-oblivious approach to assigning probabilities. We also demonstrate that multi-level architectures can better exploit the efficiency potential in approximate programs than simpler two-level machines, and we suggest a specific range of probability levels that a general-purpose approximate ISA should support.

DECAF consists of a static type system that encodes an expression's probability of correctness, a type inference and code specialization mechanism based on an SMT solver, and an optional dynamic type. We begin with an overview of DECAF and its goals before detailing each component in turn. We formalize a core language, prove its soundness in Appendix B, and report on its implementation and our empirical findings.

4.2 LANGUAGE OVERVIEW

The goal of DECAF is to enforce quality constraints on programs that execute on approximate hardware. Some proposals for approximate hardware, and our focus in this work, provide "relaxed" operations that have a high probability of yielding a correct output but a nonzero chance of producing arbitrarily wrong data [59]. Architectures that allow even a very small probability of error can conserve a large fraction of operation energy [89, 223]. Recently, Venkataramani et al. [213] suggested that hardware with multiple reliability levels—i.e., multiple probabilities of correctness—could provide better efficiency by adapting to the specific demands of approximate software. However, these fine-grained proba-

bilistic operations compose in subtle ways to impact the correctness of coarser-grained outputs.

Consider, for example, a Euclidean distance computation from a clustering algorithm:

```
float distance(float[] v1, float[] v2) {
  float total = 0.0;
  for (int i = 0; i < v1.length; ++i) {
    float d = v1[i] - v2[i];
    total += d * d;
  }
  return sqrt(total);
}
```

This distance function has been shown to be resilient to approximation in clustering algorithms [60]. To manually approximate the function, a programmer would need to select the reliability of each arithmetic operator and determine the overall reliability of the output.

In DECAF, the programmer can instead specify only the reliability of the output: here, the return value. For other values, where the "right" reliability levels are less obvious, the programmer can leave the probability inferred. The programmer decides only which variables may tolerate *some* degree of approximation and which must remain fully reliable. The programmer may write, for example, `@Approx(0.9) float` for the return type to specify that the computed value should have at least a 90% probability of being correct. The intermediate value `d` can be given the unparameterized type `@Approx float` to have its reliability inferred, and the loop induction variable `i` can be left reliable to avoid compromising control flow. The programmer never needs to annotate the operators `-`, `*`, and `+`; these reliabilities are inferred. More simply, the programmer places annotations where she can make sense of them and relies on inference where she cannot. Sections 4.3 and 4.4 describe the type system and inference.

DECAF also adapts reused code for different reliability levels. The `sqrt` function in the code above, for example, may be used in several contexts with varying reliability demands. To adapt the `sqrt` function to the reliability contexts in `distance` and other code, DECAF's type inference creates a limited number of *clones* of `sqrt` based on the (possibly inferred) types of the function's arguments and result. The operations in each clone are specialized to provide the optimal efficiency for its quality demands. Section 4.4.1 describes how DECAF specializes functions.

Finally, DECAF provides optional dynamic tracking to cope with code that is difficult or impossible to analyze statically. In our Euclidean-distance example, the `for` loop has a data-dependent trip count, so a sound static analysis would need to conservatively assume it executes an unbounded number of times. Multiplying an operator's accuracy probability approaches zero in the limit, so any conservative estimate, as in Rely [29], must assign the `total` variable the probability 0.0—no guarantees. DECAF's `@Dyn` type qualifier adds dynamic analysis for these situations. By giving the type `@Dyn float` to `total`, the programmer requests limited dynamic reliability tracking—the compiler adds code to the loop

$$
\begin{aligned}
s &\equiv T\, v := e \mid v := e \mid s\,;s \mid \textbf{if}\, e\, s\, s \mid \textbf{while}\, e\, s \mid \textbf{skip} \\
e &\equiv c \mid v \mid e \oplus_p e \mid \mathsf{endorse}(p, e) \mid \mathsf{check}(p, e) \mid \mathsf{track}(p, e) \\
\oplus &\equiv + \mid - \mid \times \mid \div \\
T &\equiv q\, \tau \\
q &\equiv \texttt{@Approx}(p) \mid \texttt{@Dyn} \\
\tau &\equiv \mathsf{int} \mid \mathsf{float} \\
v &\in \text{variables},\ c \in \text{constants},\ p \in [0.0, 1.0]
\end{aligned}
$$

(a) Core language.

$$
\begin{aligned}
e &\equiv \cdots \mid e \oplus e \mid \mathsf{check}(e) \\
q &\equiv \cdots \mid \texttt{@Approx}
\end{aligned}
$$

(b) With type inference.

Figure 6: Syntax of the DECAF language. The inferred forms (b) allow omission of the explicit probabilities in the core language (a).

to compute an upper bound on the reliability loss at run time. The programmer then requests a dynamic check, and a transition back to static tracking, with an explicit `check()` cast. Section 4.5 describes DECAF's dynamic type and run-time checks.

By combining all of these features, one possible approximate implementation of `distance` in DECAF reads:

```
@Approx(0.9) float distance(float[] v1, float[] v2) {
  @Dyn float total = 0.0;
  for (int i = 0; i < v1.length; ++i) {
    @Approx float d = v1[i] - v2[i];
    total += d * d;
  }
  return sqrt(check(total));
}
```

4.3 PROBABILITY TYPE SYSTEM

The core concept in DECAF is an expression's *probability of correctness:* the goal is to specify and control the likelihood that, in any given execution, a value equals the corresponding value in an error-free execution. This section describes DECAF's basic type system, in which each type and operation is explicitly qualified to encode its correctness probability. Later sections add inference, functions and function cloning, and optional dynamic tracking.

Figure 6 depicts the syntax for a simplified version of DECAF. A type qualifier q indicates the probability that an expression is correct: for example, the type `@Approx(0.9) int` denotes an integer that is correct in least 90% of executions.

The basic language also provides approximate operators, denoted $\oplus_p$ where p is the chance that the operation produces a correct answer *given correct inputs*. (We assume that any operator given an incorrect input produces an incorrect output, although this assumption can be conservative—for example, when multiplying an incorrect value by zero.)

The language generalizes the EnerJ language from the previous chapter, where types are either completely precise or completely approximate (providing no guarantees). DECAF has no distinct "precise" qualifier; instead, the `@Precise` annotation is syntactic sugar for `@Approx(1.0)`. EnerJ's `@Approx` is equivalent to DECAF's `@Approx(0.0)`. In our implementation, as in EnerJ, the precise qualifier, `@Approx(1.0)`, is the default, so programmers can incrementally annotate reliable code to safely enable approximation.

INFORMATION FLOW AND SUBTYPING For soundness, DECAF's type system permits data flow from high probabilities to lower probabilities but prevents low-to-high flow:

```
@Approx(0.9) int x = ...;
@Approx(0.8) int y = ...;
y = x; // sound
x = y; // error
```

Specifically, we define a subtyping rule so that a type is a subtype of other types with lower probability:

$$\frac{p \geq p'}{\texttt{@Approx}(p)\ \tau \prec \texttt{@Approx}(p')\ \tau}$$

We control implicit flows by enforcing that only fully reliable types, of the form `@Approx(1.0)` τ, may appear in conditions in `if` and `while` statements. (Appendix B gives the full type type system.)

Endorsement expressions provide an unsound escape hatch from DECAF's information flow rules. If an expression `e` has a type $q\ \tau$, then `endorse(0.8, e)` has the type `@Approx(0.8)` τ regardless of the original qualifier q.

APPROXIMATE OPERATIONS DECAF provides primitive arithmetic operations parameterized by a correctness probability. For example, the expression `x` $+_{0.9}$ `y` produces the sum of `x` and `y` at least 90% of the time but may return garbage otherwise. These operators encapsulate approximate arithmetic instructions implemented in approximate hardware architectures, such as Truffle [59] and QUORA [213]. These architectures operate more efficiently when performing operations with lower probabilities. The annotation on an operator in DECAF is a lower bound on the correctness probability for the instruction that implements it. For example, if the hardware provides an approximate add instruction with a correctness probability of 0.99, then it suffices to implement $+_{0.9}$ in DECAF. Similarly, a reliable add instruction suffices to implement an approximate addition operator with any probability (although it saves no energy).

The correctness probability for an operation `x` $+_{0.9}$ `y` is at least the product of the probabilities that `x` is correct, `y` is correct, and the addition behaves correctly

(i.e., 0.9). To see this, let $\Pr[e]$ denote the probability that the expression e is correct and $\Pr[\oplus_p]$ be the probability that an operator behaves correctly. Then the joint probability for a binary operation's correctness is:

$$\begin{aligned}\Pr[\mathsf{x} \oplus_p \mathsf{y}] &= \Pr[\mathsf{x}, \mathsf{y}, \oplus_p] \\ &= \Pr[\mathsf{x}] \cdot \Pr[\mathsf{y} \mid \mathsf{x}] \cdot \Pr[\oplus_p \mid \mathsf{x}, \mathsf{y}]\end{aligned}$$

The operator's correctness is independent of its inputs, so $\Pr[\oplus_p \mid \mathsf{x}, \mathsf{y}]$ is p. The conditional probability $\Pr[\mathsf{y} \mid \mathsf{x}]$ is at least $\Pr[\mathsf{y}]$. This bound is tight when the operands are independent but conservative when they share some provenance, as in `x + x`. So we can bound the overall probability:

$$\Pr[\mathsf{x} \oplus_p \mathsf{y}] \geq \Pr[\mathsf{x}] \cdot \Pr[\mathsf{y}] \cdot p$$

DECAF's formal type system captures this reasoning in its rule defining the result type qualifier for operators:

$$\frac{\Gamma \vdash e_1 : \texttt{@Approx}(p_1)\ \tau_1 \qquad \Gamma \vdash e_2 : \texttt{@Approx}(p_2)\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2) \qquad p' = p_1 \cdot p_2 \cdot p_{\text{op}}}{\Gamma \vdash e_1 \oplus_{p_{\text{op}}} e_2 : \texttt{@Approx}(p')\ \tau_3}$$

where optype defines the unqualified types. Appendix B lists the full set of rules.

This basic type system soundly constrains the correctness probability for every expression. The next two sections describe extensions that improve its expressiveness.

4.4 INFERRING PROBABILITY TYPES

We introduce type inference to address the verbosity of the basic system. Without inference, DECAF requires a reliability level annotation on every variable and every operation in the program. We want to allow the programmer to add reliability annotations only at outputs where requirements are intuitive. In the Euclidean distance example above, we want to uphold a 90% correctness guarantee on the returned value without requiring explicit probabilities on each `+`, `*`, and `float`. If a programmer wants to experiment with different overall output reliabilities for the `distance` function, she should not need to manually adjust the individual operators and the `sqrt` call to meet a new requirement. Instead, the programmer should express only important output correctness requirements while letting the compiler infer the details.

We extend DECAF to make probability annotations optional on both types and operations. The wildcard type qualifier is written `@Approx` without a parameter. Similarly, $\oplus$ without a probability denotes an inferred operator.

DECAF uses a constraint-based type inference approach to determine operation reliabilities and unspecified types. While constraint-based type inference is nothing new, our type system poses a distinct challenge in that its types are *continuous*. The situation is similar to Chlorophyll's spatial-scheduling type system [151], where a type assignment incurs a computational cost that needs to be

minimized. We use an SMT solver to find real-valued type assignments given constraints in the form of inequalities.

As an example, consider a program with three unknown reliabilities: two variables and one operator.

```
@Approx int a, b; ...;
@Approx(0.8) int c = a + b;
```

The program generates a trivial equality constraint for the annotated variable `c`, a subtyping inequality for the assignment, and a product constraint for the binary operator:

$$p_c = 0.8 \qquad p_c \leq p_{\text{expr}} \qquad p_{\text{expr}} = p_a \cdot p_b \cdot p_{\text{op}}$$

Here, p_{op} denotes the reliability of the addition itself and p_{expr} is the reliability of the expression `a + b`. Solving the system yields a valuation for p_{op}, the operator's reliability.

DECAF's constraint systems are typically underconstrained. In our example, the valuation $p_{\text{a}} = p_{\text{b}} = 1$, $p_{\text{op}} = 0.8$ satisfies the system, but other valuations are also possible. We want to find a solution that maximizes energy savings. Energy consumption is a dynamic property, but we can optimize a proxy: specifically, we minimize the total reliability over all operations in the program while respecting the explicitly annotated types. We encode this proxy as an objective function and emit it along with the constraints. We leave other approaches to formulating objective functions, such as profiling or static heuristics, to future work.

DECAF generates the constraints for a program and invokes the Z3 SMT solver [52] to solve them and to minimize the objective function. The compiled binary, including reliability values for each operator, may be run on a hardware simulator to observe energy usage.

4.4.1 *Function Specialization*

DECAF's inference system is interprocedural: parameters and return values can have inferred approximate types. In the Euclidean distance code above, for example, the square root function can be declared with wildcard types:

```
@Approx float sqrt(@Approx float arg) { ... }
```

A straightforward approach would infer a single type for `sqrt` compatible with all of its call sites. But this can be wasteful: if `sqrt` is invoked both from highly reliable code and from code with looser requirements, a "one-size-fits-all" type assignment for `sqrt` will be unnecessarily conservative for the more approximate context. Conversely, specializing a version of `sqrt` for every call site could lead to an exponential explosion in code size.

Instead, we use constraint solving to specialize functions a constant number of times according to calling contexts. The approach resembles traditional procedure cloning [50] but exploits DECAF's SMT formulation to automatically identify the best set of specializations. The programmer enables specialization by giving at least one parameter type or the return type of a function the inferred

`@Approx` qualifier. Each call site to a specializable function can then bind to one of the versions of the callee. The DECAF compiler generates constraints to convey that every call must invoke exactly one specialized version.

For example, in this context for a call to `sqrt`:

```
@Approx(0.9) float a = ...;
@Approx(0.8) float r = sqrt(a);
```

The compiler generates constraints resembling:

$$p_a = 0.9 \qquad p_r = 0.8 \qquad p_r \leq p_{\text{call}}$$

$$(p_{\text{call}} \leq p_{\text{ret1}} \wedge p_{\text{arg1}} \leq p_a) \vee (p_{\text{call}} \leq p_{\text{ret2}} \wedge p_{\text{arg2}} \leq p_a)$$

Here, p_{ret1} and p_{ret2} denote the reliability of `sqrt`'s return value in each of two versions of the function while p_{arg1} and p_{arg2} denote the argument. This disjunction constrains the invocation to be compatible with at least one of the versions.

The compiler also generates constraint variables—not shown above—that contain the index of the version "selected" for each call site. When inferring types for `sqrt` itself, the compiler generates copies of the constraints for the body of the function corresponding to each potential specialized version. Each constraint system binds to a different set of variables for the arguments and return value.

DECAF's optimization procedure produces specialization sets that minimize the overall objective function. The compiler generates code for each function version and adjusts each call to invoke the selected version.

Like unbounded function inlining, unbounded specialization can lead to a combinatorial explosion in code size. To avoid this, DECAF constrains each function to at most k versions, a compile-time parameter. It also ensures that all specialized function versions are *live*—bound to at least one call site—to prevent the solver from "optimizing" the program by producing dead function variants and reducing their operator probabilities to zero.

The compiler also detects recursive calls that lead to cyclic dependencies and emits an error. Recursion requires that parameter and return types be specified explicitly.

4.5 OPTIONAL DYNAMIC TRACKING

A static approach to constraining reliability avoids run-time surprises but becomes an obstacle when control flow is unbounded. Case-by-case solutions for specific forms of control flow could address some limitations of static tracking but cannot address all dynamic behavior. Instead, we opt for a general dynamic mechanism.

Inspired by languages with gradual and optional typing [209], we provide optional run-time reliability tracking via a dynamic type. The data-dependent loop in Section 4.2's Euclidean distance function is one example where dynamic tracking fits. Another important pattern where static approaches fall short is convergent algorithms, such as simulated annealing, that iteratively refine a result:

```
@Approx float result = ...;
while (fitness(result) > epsilon)
  result = refine(result);
```

In this example, the `result` variable flows into itself. A conservative static approach, such as our type inference, would need to infer the degenerate type `@Approx(0.0) float` for `result`. Fundamentally, since the loop's trip count is data-dependent, purely static solutions are unlikely to determine an appropriate reliability level for `result`. Previous work has acknowledged this limitation by abandoning guarantees for any code involved in dynamically bounded loops [29].

To cope with these situations, we add optional dynamic typing via a `@Dyn` type qualifier. The compiler augments operations involving `@Dyn`-qualified types with bookkeeping code to compute the probability parameter for each result. Every dynamically tracked value has an associated *dynamic correctness probability field* that is managed transparently by the compiler. This dynamic tracking follows the typing rules analogous to those for static checking. For example, an expression `x` $+_{0.9}$ `y` where both operands have type `@Dyn float` produces a new `@Dyn float`; at run time, the bookkeeping code computes the dynamic correctness as the product of `x`'s dynamic probability value, `y`'s probability, and the operator's probability, 0.9.

Every dynamic type `@Dyn` τ is a supertype of its static counterparts `@Approx` τ and `@Approx(`p`)` τ. When a statically typed value flows into a dynamic variable, as in:

```
@Approx(0.9) x = ...;
@Dyn y = x;
```

The compiler initializes the run-time probability field for the variable `y` with `x`'s static reliability, 0.9.

Flows in the opposite direction—from dynamic to static—require an explicit dynamic cast called a *checked endorsement*. For an expression e of type `@Dyn` τ, the programmer writes `check(`p`,` e`)` to generate code that checks that the value's dynamic probability is at least p and produce a static type `@Approx(`p`)` τ. If the check succeeds, the static type is sound. If it fails, the checked endorsement raises an exception. The program can handle these exceptions to take corrective action or fall back to reliable re-execution.

This dynamic tracking strategy ensures that run-time quality exceptions are predictable. In a program without (unchecked) endorsements, exceptions are raised deterministically: the program either always raises an exception or never raises one for a given input. This is because control flow is fully reliable and the dynamic probability tracking depends only on statically-determined operator probabilities, not the dynamic outcomes of approximate operations.

In our experience, `@Dyn` is only necessary when an approximate variable forms a loop-carried dependency. Section 4.8 gives more details on the placement and overhead of the `@Dyn` qualifier.

INTERACTION WITH INFERENCE Like explicitly parameterized types, DECAF's inferred static types can interact bidirectionally with the `@Dyn`-qualified types. When a value with an inferred type flows into a dynamic type, as in:

```
@Approx x = ...;
@Dyn y = x;
```

The assignment into `y` generates no constraints on the type of `x`; any inferred type can transition to dynamic tracking. (The compiler emits a warnings when no other code constrains `x`, a situation that can also arise in the presence of endorsements. See the next section.)

Inference can also apply when transitioning from dynamic to static tracking with a checked endorsement. DECAF provides a `check(`*e*`)` variant that omits the explicit probability threshold and infers it. This inferred parameter is useful when other constraints apply, as in the last line of the Euclidean distance example above:

```
return sqrt(check(total));
```

The result of the `sqrt` call needs to meet the programmer's `@Approx(0.9) float` constraint on the return type, but the correctness probability required on `total` to satisfy this demand is not obvious—it depends on the implementation of `sqrt`. The compiler can infer the right check threshold, freeing the programmer from manual tuning.

Operators with `@Dyn`-typed operations cannot be inferred. Instead, operations on dynamic values are reliable by default; the programmer can explicitly annotate intermediate operations to get approximate operators.

4.6 USING THE LANGUAGE

This section details two practical considerations in DECAF beyond the core mechanisms of inference, specialization, and dynamic tracking.

CONSTRAINT WARNINGS In any type inference system, programmers can encounter unintended consequences when constraints interact in unexpected ways. To guard against two common categories of mistakes, the DECAF compiler emits warnings when a program's constraint system ether *allows* a probability variable to be 0.0 or *forces* a probability to 1.0. Each case indicates a situation that warrants developer attention.

An inferred probability of 0.0 indicates that a variable is unconstrained—no dependency chain connects the value to an explicit annotation. Unconstrained types can indicate dead code, but they can also signal some legitimate uses that require additional annotation. If an inferred variable flows only into endorsements and `@Dyn` variables, and never into explicitly annotated types, it will have no constraints. Without additional annotation, the compiler will use the most aggressive approximation parameter available in the hardware. The programmer can add explicit probabilities to constrain these cases.

Conversely, an inferred probability of 1.0—i.e., no approximation at all—can indicate a variable that flows into itself, as in the iterative refinement example in the previous section or the `total` accumulation variable in the earlier Euclidean distance example. This self-flow pattern also arises when updating a variable as in `x = x + 1` where `x` is an inferred `@Approx int`. In these latter situations, a simple solution is to introduce a new variable for the updated value (approximating a static single assignment transformation). More complex situations require a `@Dyn` type.

HARDWARE PROFILES While DECAF's types and inference are formulated using a continuous range of probabilities, realistic approximate hardware is likely to support only a small number of discrete reliability levels [59, 213]. The optimal number of levels remains an open question, so different machines will likely provide different sets of operation probabilities. A straightforward and portable approach to exploiting this hardware is to round each operation's probability up to the nearest hardware-provided level at deployment time. When there is no sufficiently accurate approximation level, a reliable operation can be soundly substituted.

We also implement and evaluate an alternative approach that exploits the hardware profile of the intended deployment platform at compile time. The compiler can use such an *a priori* hardware specification to constrain each variable to one of the available levels. The SMT solver can potentially find a better valuation of operator probabilities than with post-hoc rounding. (This advantage is analogous to integer linear programming, where linear programming relaxation followed by rounding typically yields a suboptimal but more efficient solution.)

In our evaluation, we study the effects of finite-level hardware with respect to a continuous ideal and measure the advantage of *a priori* hardware profiles.

4.7 FORMALISM

A key feature in DECAF is its conservative quality guarantee. In the absence of unchecked endorsements, a DECAF program's probability types are *sound:* an expression's static type gives a lower bound on the actual run-time probability that its value is correct. The soundness guarantee applies even to programs that combine static and dynamic tracking. To make this guarantee concrete, we formalize a core of DECAF and prove its soundness.

The formal language represents a version of DECAF where all types have been inferred. Namely, the core language consists of the syntax in Figure 6a. It excludes the inferred expressions and types in Figure 6b but includes approximate operators, dynamic tracking, and endorsements. (While we define the semantics for both kinds of endorsements for completeness, we will prove a property for programs having only *checked* endorsements. Unchecked endorsements are an unsound escape hatch.)

The core language also includes one expression that is unnecessary in the full version of DECAF: `track(`p`, `e`)`. This expression is a cast from any static type `@Approx(`p'`)` τ to its dynamically-tracked equivalent, `@Dyn` τ. At run time, it initializes the dynamic probability field for the expression. In the full language, the compiler can insert this coercion transparently, as with implicit int-to-float coercion in Java or C.

This section gives an overview of the formalism's type system, operational semantics, and main soundness theorem. Appendix B gives the full details and proofs.

TYPES There are two judgments in DECAF's type system: one for expressions, $\Gamma \vdash e : T$, and another for statements, $\Gamma \vdash s : \Gamma'$, which builds up the static context Γ'.

One important rule gives the static type for operators, which multiplies the probabilities for both operands with the operator's probability:

$$\frac{\Gamma \vdash e_1 : \texttt{@Approx}(p_1)\ \tau_1 \qquad \Gamma \vdash e_2 : \texttt{@Approx}(p_2)\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2) \qquad p' = p_1 \cdot p_2 \cdot p_{\text{op}}}{\Gamma \vdash e_1 \oplus_{p_{\text{op}}} e_2 : \texttt{@Approx}(p')\ \tau_3}$$

Here, optype is a helper judgment defining operators' unqualified types.

OPERATIONAL SEMANTICS We present DECAF's run-time behavior using operational semantics: small-step for statements and large-step for expression evaluation. Both sets of semantics are nondeterministic: the operators in DECAF can produce either a correct result number, c, or a special error value, denoted $\square$.

To track the probability that a value is correct (that is, not $\square$), the judgments maintain a probability map S for all defined variables. There is a second probability map, D, that reflects the compiler-maintained dynamic probability fields for @Dyn-typed variables. Unlike D, the bookkeeping map S is an artifact for defining our soundness criterion—it does not appear anywhere in our implementation.

The expression judgment $H; D; S; e \Downarrow_p V$ indicates that the expression e evaluates to the value V and is correct with probability p. We also use a second judgment, $H; D; S; e \Downarrow_p V, p_d$, to denote dynamically-tracked expression evaluation, where p_d is the computed shadow probability field. As an example, the rules for variable lookup retrieve the "true" probability from the S map and the dynamically-tracked probability field from D:

VAR
$$\frac{v \notin D}{H; D; S; v \Downarrow_{S(v)} H(v)}$$

VAR-DYN
$$\frac{v \in D}{H; D; S; v \Downarrow_{S(v)} H(v), D(v)}$$

The statement step judgment is $H; D; S; s \longrightarrow H'; D'; S'; s'$. The rule for mutation is representative:

$$\frac{H; D; S; e \Downarrow_p V}{H; D; S; v := e \longrightarrow H, v \mapsto V; D; S, v \mapsto p; \textbf{skip}}$$

It updates both the heap H and the bookkeeping map S. A similar rule uses the dynamically-tracked expression judgment and also updates D.

SOUNDNESS To express our soundness property, we define a *well-formedness* criterion that states that a dynamic probability field map D and a static context Γ together form lower bounds on the "true" probabilities in S. We write this property as $\vdash D, S : \Gamma$.

Definition 1 (Well-Formed). $\vdash D, S : \Gamma$ *iff for all* $v \in \Gamma$,

- *If* $\Gamma(v) = \texttt{@Approx}(p)\ \tau$, *then* $p \leq S(v)$ *or* $v \notin S$.
- *If* $\Gamma(v) = \texttt{@Dyn}\ \tau$, *then* $D(v) \leq S(v)$ *or* $v \notin S$.

The soundness theorem states that D and S remain well-formed through the small-step statement evaluation semantics.

Theorem 1 (Soundness)**.** *For all programs* s *with no* endorse *expressions, for all* $n \in \mathrm{N}$ *where* $\cdot;\cdot;\cdot;s \longrightarrow^n H;D;S;s'$, *if* $\cdot \vdash s : \Gamma$, *then* $\vdash D, S : \Gamma$.

See Appendix B for the full proof of the theorem. The appendix also states an erasure theorem to show that S does not affect the actual operation of the program: its only purpose is to define soundness for the language.

4.8 EVALUATION

We implemented DECAF and evaluated it using a variety of approximate applications. The goals of this evaluation were twofold: to gain experience with DECAF's language features; and to apply it as a testbed to examine the implications of application-level constraints for hardware research.

4.8.1 *Implementation*

We implemented a type checker, inference system, and runtime for DECAF as an extension to Java. The implementation extends the simpler EnerJ type system [180] and is similarly based on Java 8's extensible type annotations [56]. The compiler uses AST instrumentation and a runtime library to implement dynamic tracking for the @Dynqualifier. For Java arrays, the implementation uses conservative object-granularity type checking and dynamic tracking.

The compiler generates constraints for the Z3 SMT solver [52] to check satisfiability, emit warnings, and tune inferred operator probabilities. The constraint systems exercise Z3's complete solver for nonlinear real arithmetic. To stay within the reach of this complete solver, we avoided generating any integer-valued constraints, which can quickly cause Z3's heuristics to reject the query as potentially undecidable.

Z3 does not directly support optimization problems, so we use a straightforward search strategy to minimize the objective function. The linear search executes queries repeatedly while reducing the bound on the objective until the solver reports unsatisfiability or times out (after 1 minute in our experiments). The optimization strategy's dependence on real-time behavior means that the optimal solutions are somewhat nondeterministic. Also, more complex constraint systems can time out earlier and lead to poorer optimization results—meaning that adding constraints meant to improve the solution can paradoxically worsen it. In practice, we observe this adverse effect for two benchmarks where hardware constraints cause an explosion in solver time (see below).

We optimize programs according to a static proxy for a program's overall efficiency (see Section 4.4). Our evaluation tests this objective's effectiveness as a static proxy for dynamic behavior by measuring dynamic executions.

Application	Description	Build Time	LOC	@Approx	@Approx(p)	@Dyn	Approx	Dyn
fft	Fourier transform	2 sec	747	37	11	23	7%	55%
imagefill	Bar code recognition	14 min	344	76	20	0	45%	<1%
lu	LU decomposition	1 min	775	63	9	12	24%	<1%
mc	Monte Carlo approximation	2 min	562	67	8	6	21%	<1%
raytracer	3D image reading	1 min	511	38	4	2	12%	44%
smm	Sparse matrix multiply	1 min	601	37	4	4	28%	28%
sor	Successive over-relaxation	19 min	589	43	3	3	63%	<1%
zxing	Bar code recognition	16 min	13180	220	98	4	31%	<1%

Table 4: Benchmarks used in the evaluation. The middle set of columns show the static density of DECAF annotations in the Java source code. The final two columns show the dynamic proportion of operations in the program that were approximate (as opposed to implicitly reliable) and dynamically tracked (both approximate and reliable operations can be dynamically tracked).

4.8.2 *Experimental Setup*

We consider an approximate processor architecture where arithmetic operations may have a probability of failure, mirroring recent work in hardware design [59, 89, 213, 223]. Because architecture research on approximate computing is at an early stage, we do not model a specific CPU design: there is no consensus in the literature surrounding which reliability parameters are best or how error probabilities translate into energy savings. Instead, we design our evaluation to advance the discussion by exploring the constraints imposed by language-level quality demands. We explore error levels in a range commensurate with current proposals—i.e., correctness probabilities 99% and higher—to inform the specific parameters that hardware should support. Researchers can use this platform-agnostic data to evaluate architecture designs.

We implemented a simulation infrastructure that emulates such a machine with tunable instruction reliability. The simulator is based on the freely available implementation from EnerJ (Chapter 3), which uses a source-to-source translation of Java code to invoke a run-time library that injects errors and collects execution statistics. To facilitate simulation, three pieces of data are exported at compile time and imported when the runtime is launched. Every operator used in an approximate expression is exported with its reliability. When an operator is encountered, the simulator looks up its reliability or assumes reliable execution if the operator is not defined. To facilitate `@Dyn` expression tracking, the compiler exports each variable's reliability and the runtime imports this data to initialize dynamic reliability fields. Finally, the run-time uses a mapping from invocations to function variants to look up the reliabilities specialized functions.

Performance statistics were collected on a 4-core, 2.9 GHz Intel Xeon machine with 2-way SMT and 8 GB RAM running Linux. We used OpenJDK 1.7.0's HotSpot VM and Z3 version 4.3.1.

4.8.3 *Benchmarks and Annotation*

We evaluate 8 of the Java benchmarks from the evaluation of EnerJ (see Section 3.6). Table 4 lists the applications and statistics about their source code and annotations.

The original EnerJ annotations distinguish approximation-tolerant variables (using EnerJ's `@Approx`) from reliable variables (the default). To adapt the programs for DECAF, we left most of these type annotations as the inferred `@Approx` annotation. On the output of each benchmark and on a few salient boundaries between software components, we placed concrete `@Approx(`p`)` reliability restrictions. These outputs have a variety of types, including single values, arrays of pixels, and strings. Informed by compiler warnings, we used `@Dyn` for variables involved in loop-carried dependencies where static tracking is insufficient along with `check()` casts to transition back to static types. Finally, we parameterized some `@Approx` annotations to add constraints where they were lacking—i.e., when inferred values flow into endorsements or `@Dyn` variables exclusively.

For each application, we applied the `@Approx(0.9)` qualifier to the overall output of the computation. This and other explicit probability thresholds dictate the required reliability for the program's operations, which we measure in this evaluation. We believe these constraints to be representative of practical deployments, but deployment scenarios with looser or tighter output quality constraints will lead to correspondingly different operator probability requirements.

4.8.4 *Results*

We use these benchmarks to study the implications of our benchmarks on the design of approximate hardware. Key findings (detailed below) include:

- By tuning a application to match a specific hardware profile, a compiler can achieve better efficiency than with hardware-oblivious optimization. Hardware-targeted optimization improves efficiency even on a simple two-level approximate architecture.
- Most benchmarks can make effective use of multiple operator probabilities. Processors should provide at least two probability levels for approximate operations to maximize efficiency.
- Operator correctness probabilities between $1.0 - 10^{-2}$ and $1.0 - 10^{-8}$ are most broadly useful. Probabilities outside this range benefit some benchmarks but are less general.

These conclusions reflect the characteristics of our benchmarks and their annotations, which in turn are based on recent work on approximate computing.

4.8.5 *Sensitivity to Hardware Reliability Levels*

An ideal approximate machine would allow arbitrarily fine-grained reliability tuning to exactly match the demands of every operation in any application. Realistically, however, an architecture will likely need to provide a fixed set of probability levels. The number of levels will likely be small to permit efficient instruction encoding. We use DECAF to evaluate the impact of this restriction by simulating different hardware configurations alongside the ideal, continuously approximable case.

We simulate architectural configurations with two to eight levels of reliability. A two-level machine has one reliable operation mode ($p = 1.0$) and one approximate mode, for which we choose $p = 0.99$. This configuration resembles the Truffle microarchitecture, which provides only one approximate mode [59]. We evaluate multi-level configurations that each add a probability level with one more "nine": $p = 0.999$, $p = 0.9999$, and so on, approaching fully reliable operation. Architecture proposals suggest that even low probabilities of error can yield energy savings [78, 86, 90].

SOLVING VS. ROUNDING LEVELS To run a DECAF-compiled program on realistic approximate hardware, two strategies are possible for selecting the prob-

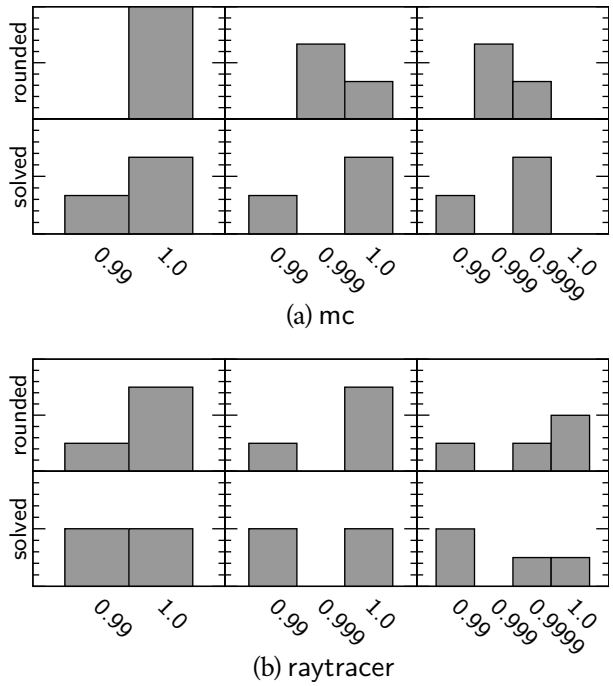


Figure 7: Sensitivity to hardware restrictions for two representative benchmarks. The horizontal axes show the probability levels while the vertical axes reflect the fraction of all approximate operations in an execution assigned to each level. The *rounded* executions were assigned to levels after solving without restrictions; the *solved* executions used the hardware profile during type inference.

ability level for each operation. A simplistic approach rounds the inferred probabilities up to the nearest level. The compiler can potentially do better by using the SMT solver to apply hardware constraints during type inference if the deployment architecture is known ahead of time.

Figure 7 compares the two approaches, denoted *solving* and *rounding*, for two of our benchmarks on two-, three-, and four-level machines. Constrained solving shifts the distribution toward lower probabilities in each of the three machines. When mc runs on a three-level machine, for example, the simple rounding strategy rarely uses the lowest $p = 0.99$ reliability level; if we instead inform the solver that this level is available, the benchmark can use it for a third of its approximate operations. A similar effect arises for raytracer, for which the solver assigns the lowest reliability level to about half of the operations executed while rounding makes the majority of operations fully reliable.

These differences suggest that optimizing an approximate program for a specific hardware configuration can enable significant energy savings, *even for simple approximate machines with only two probability levels*. DECAF's solver-based tuning approach enables this kind of optimization.

While solving for hardware constraints can lead to better efficiency at run time, it can also be more expensive at compile time. The SMT queries for most benchmarks took only a few minutes, but two outliers—`sor` and `zxing`—took much longer when level constraints were enabled. For `sor`, solving succeeded for machine configurations up to four levels but exceeded a 30-minute timeout for larger level counts; `zxing` timed out even in the two-level configuration. In the remainder of this evaluation, we use the more sophisticated solving scheme, except for these cases where solving times out and we fall back to the cheaper rounding strategy.

PROBABILITY GRANULARITY More hardware probability levels can enable greater efficiency gains by closely matching applications' probability requirements. Figure 8 depicts the allocation of approximate operations in benchmark executions to reliability levels for a range of hardware configurations from 2 to 8 levels. In this graphic, white and lighter shades indicate more reliable execution and correspondingly lower efficiency gains; darker shades indicate more opportunity for energy savings.

Five of the eight benchmarks use multiple operator probability levels below 1.0 when optimized for hardware that offers this flexibility. This suggests that multi-level approximate hardware designs like QUORA [213] can unlock more efficiency gains in these benchmarks than simpler single-probability machines like Truffle [59]. The exceptions are `imagefill`, `lu`, and `smm`, where a single probability level seems to suffice for the majority of operations.

Most of our benchmarks exhibit diminishing returns after a certain number of levels. For example, mc increases its amount of approximation up to four levels but does not benefit from higher level counts. Similarly, imagefill's benefits do not increase after six levels. In contrast, raytracer and zxing see improvements for configurations up to eight levels.

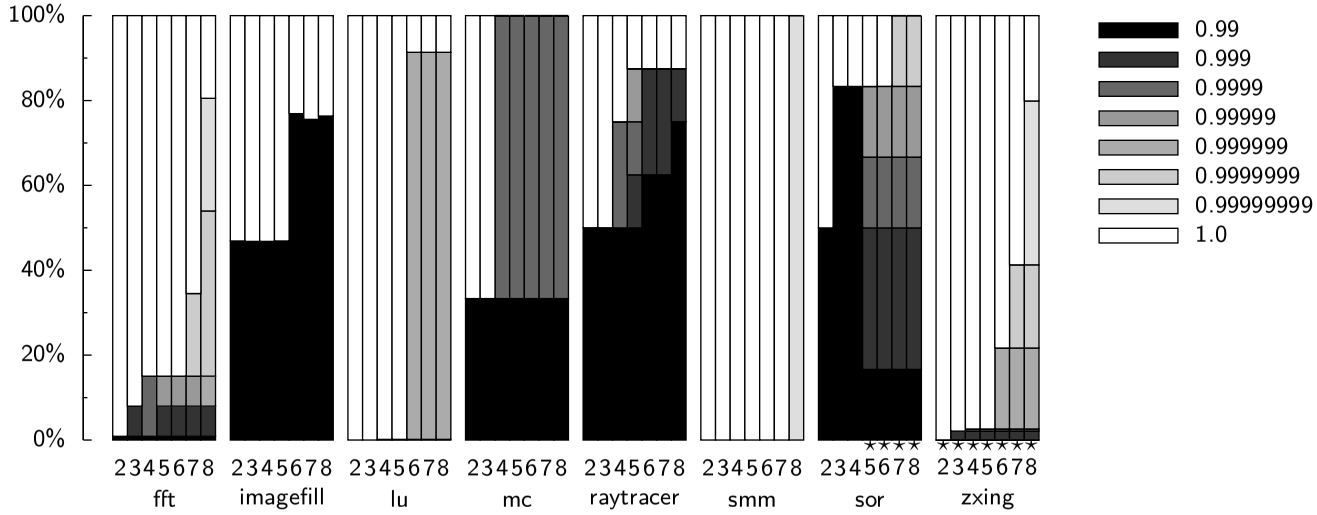


Figure 8: Operator probability breakdowns. Each bar shows a hardware configuration with a different number of levels. Darker shades indicate lower probabilities and correspondingly higher potential energy savings. Bars marked ⋆ used the cheaper rounding strategy instead of hardware-specific solving to determine levels.

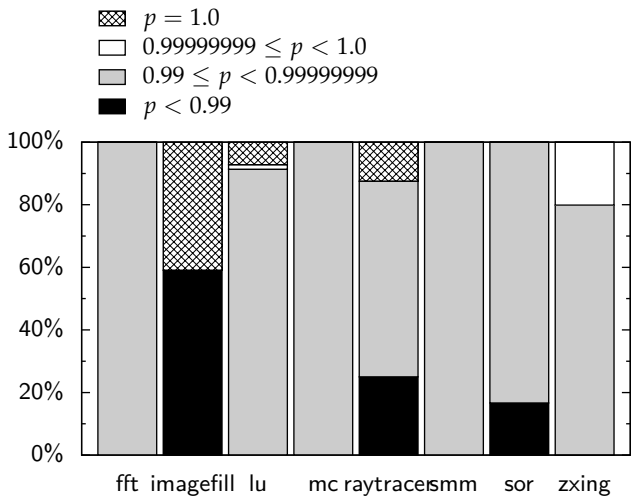


Figure 9: Approximate operation probabilities on an ideal continuous machine. The gray boxes show the probability range accommodated by our simulated discrete-level machines, while the white box represents higher-reliability operations and the black boxes are lower-reliability operations. The hatched box indicates approximate operations that are forced to execute reliably by program constraints, even on ideal "continuous" hardware.

In an extreme case, smm falls back to reliable execution for nearly all of its operations in every configuration we simulated except for the eight-level machine. This suggests that a two-level machine would suffice for this benchmark, provided the single approximate operation probability were high enough. On the other hand, specializing a two-level architecture to this outlier would limit potential efficiency gains for other applications.

Increasing reliability levels do not strictly lead to efficiency benefits in DECAF's solver-based approach. For sor, the added constraints for more granular hardware levels lead to a more complex SMT solver query and eventually timeouts. After four levels, the solver failed to optimize the benchmark and we fell back to the naïve rounding strategy, which leads to lower efficiency gains. These timeouts are partially due to DECAF's straightforward encoding of program and hardware constraints; future work on optimizing DECAF's constraint systems for efficient solving could make larger level counts more tractable.

COMPARISON TO IDEAL An ideal approximate architecture that features arbitrary probability levels could offer more flexibility at the extremes of the probability range. To evaluate the importance of higher and lower levels, we simulated an ideal continuous machine. Figure 9 shows the fraction of approximate operations in executions of each benchmark that used probabilities below the range of our realistic machines (below 99% probability) and above the range (above $p = 1.0 - 10^{-8}$). The figure also shows the operations that executed with probability exactly 1.0 even on this continuous architecture, indicating that they were constrained by program requirements rather than hardware limitations.

For all but one application, most operations lie in the range of probabilities offered by our discrete machine simulations. Only three benchmarks show a significant number of operations with probabilities below 99%, and one outlier, `imagefill`, uses these low-probability operations for nearly all of its approximable computations. The only benchmark that significantly uses higher-probability operations is `zxing`, where about 20% of the operations executed had a probability greater than $1.0 - 10^{-8}$. Among our benchmarks, the $0.99 \leq p \leq 0.99999999$ probability range suffices to capture most of the flexibility offered by an ideal machine.

EXAMPLE ENERGY MODEL The goal of measuring error probabilities in this evaluation is to allow hardware designers to plug in energy models. To give a sense of the potential savings, we apply a simple energy model based on EnerJ [180]: a correctness probability of 0.99 yields 30% energy savings over a precise operation, $p = 10^{-4}$ saves 20%, $p = 10^{-6}$ saves 10%, and other levels are interpolated. More optimistic hardware proposals exist (e.g., Venkataramani et al. [213]), but EnerJ's conservative CPU-based model serves as a useful point of comparison. On an eight-level machine, the total operation energy saved is:

Benchmark	Rounded	Solved
fft	<1%	<1%
imagefill	7%	22%
lu	<1%	9%
mc	5%	23%
raytracer	1%	20%
smm	2%	2%
sor	12%	—
zxing	1%	—

The table shows the modeled energy reduction for both the hardware-oblivious rounding strategy and the platform-specific solving strategy (except where the solver timed out). The results reflect the above finding that solving yields better savings than rounding after the fact.

4.8.6 *Interprocedural Inference and Specialization*

In all of our benchmarks, we used the inferred `@Approx` qualifier on function parameters and return types to let the compiler propagate constraints interprocedurally. This let us write simpler annotations that directly encoded our desired output correctness constraints and avoid artificially aligning them with function boundaries. In some benchmarks—namely, lu and zxing—multiple call sites to these inferred functions allowed the compiler to specialize variants and improve efficiency.

In lu, for example, specialization was critical to making the benchmark take advantage of approximate hardware. That benchmark uses a utility function that copies approximate arrays. The factorization routine has three calls to the copying function, and each of the intermediate arrays involved have varying impact on the output of the benchmark. When we limit the program to $k = 1$ function

variants—disabling function specialization—all three of these intermediates are constrained to have identical correctness probability, and all three must be as reliable as the least tolerant among them. As a result, the benchmark as a whole exhibits very little approximate execution: more than 99% of its approximate operations are executed reliably ($p = 1.0$). By allowing $k = 2$ function specializations, however, the compiler reduces the fraction to 8%, and $k = 3$ specializations further reduce it to 7%. A similar pattern arises in the zxing benchmark, where utility functions on its central data structure—a bit-matrix class used to hold black-and-white pixel values—are invoked from different contexts throughout the program.

4.8.7 *Dynamic Tracking*

The `@Dyn` type qualifier lets programmers request dynamic probability tracking, in exchange for run-time overhead, when DECAF's static tracking is too conservative. Table 4 shows the number of types we annotated with `@Dyn` in each benchmark. Dynamic tracking was necessary at least once in every benchmark except one (`imagefill`). Most commonly, `@Dyn` applied in loops that accumulate approximate values. For example, `zxing` has a loop that searches for image patterns that suggest the presence of parts of a QR code. The actual detection of each pattern can be statically tracked, but the loop also accumulates the total size of the image patterns. Since the loop's trip count depends on the input image, dynamic tracking was necessary for precision: no nonzero static bound on the size variable's probability would suffice.

Table 4 also shows the fraction of operations in an execution of each benchmark that required dynamic tracking. In five of our eight benchmarks, less than 1% of the operations in the program need to be dynamically tracked, suggesting that energy overhead would be minimal. In the remaining three benchmarks, a significant portion of the application's approximate and reliable operations required dynamic tracking. (Recall that operations on `@Dyn`-typed variables are reliable by default but still require propagation of probability information.) In the worst case, `fft` uses dynamic tracking for 55% of the operations in its execution.

In a simple implementation, each dynamically tracked operation expands out to two operations, so the percentage of dynamically tracked operations is equivalent to the overhead incurred. An optimizing compiler, however, can likely coalesce and strength-reduce the multiplications-by-constants that make up tracking code. In `fft`, for example, an inner loop reads two array elements, updates them each with a series of four approximate operations, and writes them back. A standard constant-propagation optimization could coalesce the four tracking operations to a single update. In other cases, such as `zxing`'s pattern-search loop described above, the correctness probability loss is directly proportional to the loop trip count. Standard loop optimizations could hoist these updates out of the loop and further reduce overhead.

4.8.8 *Tool Performance*

Table 4 lists the running time of the inference system for each benchmark. The total time includes time spent on the initial system-satisfiability query, the optimization query series, parsing and analyzing the Java source code, and checking for DECAF constraint warnings. Most of the time is spent in optimization, so it can be faster to produce a satisfying but suboptimal type assignment. The optimization queries have a timeout of one minute, so the final SMT query in the series can take at most this long; for several benchmarks, the solver returns *unsatisfiable* before this timeout is reached. The compiler typically runs in about 1–20 minutes. One outlier is `fft`, whose constraint system is fast to solve because of the benchmark's reliance on dynamic tracking.

These measurements are for a continuous configuration of the system rather than a more expensive level-constrained version. Solver times for hardware-constrained inference are comparable, except for the two benchmarks mentioned above that scale poorly and eventually time out: `sor` and `zxing`.

4.9 DISCUSSION

DECAF is a quality-focused complement to EnerJ. The basic idea is simple: generalize EnerJ's all-or-nothing approximation binary to a continuum of accuracy levels. Type inference, code specialization, and optional dynamic typing all extend the core idea to make the full system ergonomic.

DECAF's type-inference approach in particular holds an important lesson for convenient quality control mechanisms: programmers should be able to *choose* where to control quality explicitly and, conversely, where to leave the details up to the compiler. The next chapter, on probabilistic assertions, departs from a type-based paradigm but preserves the same philosophy: it lets programmers constrain quality where it makes the most sense.

5

PROBABILISTIC ASSERTIONS

5.1 INTRODUCTION

Traditional assertions express logical properties that help programmers design and reason about the correctness of their program. Verification tools guarantee that every execution will satisfy an assertion, such as the absence of null dereferences or a legal value range for a variable. However, many applications produce or consume probabilistic data, such as the relevance of a document to a search, the distance to the nearest coffee shop, or the estimated arrival time of the next bus. From smartphones with sensors to robots to machine learning to big data to approximate computation, many applications use probabilistic values.

Current assertion languages and verification tools are insufficient in this domain. Traditional assertions do not capture probabilistic correctness because they demand that a property hold on *every* execution. Recent work on inference in probabilistic programming languages builds language abstractions to aid programmers in describing machine learning models but does not deal with verification of probabilistic correctness properties [69, 129, 149, 159]. Sankaranarayanan et al. [184] address the verification of programs in probabilistic programming languages through polyhedral volume estimation, but this approach limits the domain to programs with linear arithmetic over constrained probability distributions. In contrast, this work defines a semantics for computing in mainstream languages over a broader set of distributions with sampling functions but does not verify programs.

We propose *probabilistic assertions* (passerts), which express probabilistic program properties, and *probabilistic evaluation*, which verifies them. A passert statement is a probabilistic logical statement over random variables. *Probabilistic evaluation* extracts, optimizes, and evaluates the distribution specified in a passert by combining techniques from static verification, symbolic execution, and dynamic testing.

A probabilistic assertion:

```
passert e, p, cf
```

means that the probability that the Boolean expression `e` holds in a given execution of the program should be at least `p` with confidence `cf`. The parameters `p` (defaults to 0.5) and `cf` (defaults to 95%) are optional. Our analysis estimates the likelihood that `e` is true, bounds any error in that estimate, and determines whether that estimate is significantly different from `p`. For example, consider the

following function, which adds Gaussian noise to users' true locations to protect their privacy.

```
def obfuscate_location(location):
  noise = random.gauss(0,1)
  d = distance(location, location + noise)
  passert d < 10, 0.9, 95%
  return location + noise
```

To ensure that obfuscation does not change a user's true location too much, the programmer asserts that the Euclidean distance between the true and obfuscated location should be within 10 miles at least 90% of the time with 95% confidence. While occasional outliers are acceptable, the programmer wants to ensure that the common case is sufficiently accurate and therefore useful.

A traditional assertion—`assert d < 10`—does not express this intent. Since the Gaussian distribution has a non-zero chance of adding any amount of noise, some executions will make `d` greater than 10. Since these infrequent outlier cases are possible, traditional verification must conclude that the assertion does not hold.

Probabilistic evaluation checks the probabilistic logical statement over random variables expressed by the passert. It first performs *distribution extraction*, which is a symbolic execution that builds a Bayesian network, a directed, acyclic graphical model. Nodes represent random variables from the program and edges between nodes represent conditional dependences between those random variables. This process defines a probabilistic semantics in which *all* variables are distributions. Constants (e.g., `x = 3`) are point-mass distributions. Random distributions, both simple (uniform, Gaussian, etc.) and programmer-defined, are represented symbolically. Other variables are defined in terms of these basic distributions.

For example, let L, D, and N be the random variables corresponding to the variables `location`, `d`, and `noise` in the above program. The passert constrains the probability $\Pr[D < 10]$ given that L is a point-mass distribution and that N is drawn from a Gaussian:

$$\Pr[D < 10 \mid L = \mathsf{location}, N \sim \mathcal{N}(0,1)] > 0.9$$

This inequality constrains the probability of correctness for a particular input location. Alternatively, programmers may express a distribution over expected input locations by, for example, setting the `location` variable to sample from a uniform distribution. The passert then measures the likelihood that the obfuscation will yield acceptable results for uniformly distributed input locations:

$$\Pr[D < 10 \mid L \sim \mathcal{U}, N \sim \mathcal{N}(0,1)] > 0.9$$

Our key insight is that, with this probabilistic semantics for passerts, we can optimize the Bayesian network representation and significantly improve the efficiency of verification. Using known statistical properties, our optimizations produce a simplified but equivalent Bayesian network. For example, we exploit identities of common probability distributions and Chebyshev's inequality. In

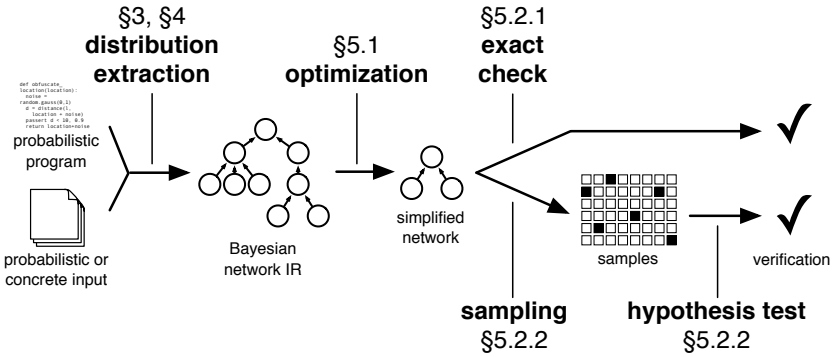


Figure 10: MAYHAP's workflow to verify probabilistic assertions.

some cases, these simplifications are sufficient to facilitate direct computation and verify the passert precisely. Otherwise, we sample the simplified Bayesian network and perform a hypothesis test to statistically verify the passert. We use *acceptance sampling*, a form of hypothesis testing, to bound the chance of both false positives and false negatives subject to a confidence level. Programmers can adjust the confidence level to trade off between analysis time and verification accuracy.

We implement this approach in a tool called MAYHAP that takes C and C++ programs with passerts as inputs. MAYHAP emits either *true*, *false*, or *unverifiable* along with a confidence interval on the assertion's probability. Figure 10 gives an overview. We implement the entire toolchain for MAYHAP in the LLVM compiler infrastructure [101]. First, MAYHAP transforms a probabilistic C/C++ program into a Bayesian network that expresses the program's probabilistic semantics. For program inputs, developers provide concrete values or describe input distributions. MAYHAP optimizes the Bayesian-network representation using statistical properties and then either evaluates the network directly or performs sampling.

We implement case studies that check the accuracy of approximate programs. We also explore domains beyond approximate computing where probabilistic correctness is also important: using data from sensors and obfuscating data for user privacy. We show that passerts express their correctness properties and that MAYHAP offers an average speedup of 24× over stress testing with rote sampling. MAYHAP's benefits over simple stress testing—repeated execution of the original program—are threefold. First, statistical simplifications to the Bayesian network representation reduce the work required to compute each sample: for example, reducing the sum of two Gaussian distributions into a single Gaussian halves the necessary number of samples. Second, distribution extraction has the effect of partially evaluating the probabilistic program to slice away the non-probabilistic parts of the computation. Sampling the resulting Bayesian network eliminates wasteful re-execution of deterministic code. Third, our approach either directly evaluates the passert or derives a number of samples sufficient for statistical significance. It thereby provides statistical guarantees on the results of sampling that blind stress testing does not guarantee.

5.2 PROGRAMMING MODEL

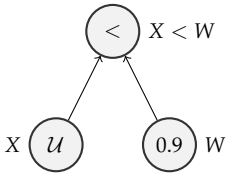
This section presents an intuitive view of programs as probabilistic computations over random variables. For our purposes, a probabilistic program is an ordinary imperative program that calls sampling functions for probability distributions [95]. Consider this simple program:

```
x = random.uniform(0,1)
w = 0.9
passert x < w, 0.90
```

This program samples from a uniform distribution, ranging from 0 to 1, assigns a concrete value to `w`, and then asserts that the sample is less than 0.9 using the comparison `x < w` with 90% probability. An invocation of `random.uniform` returns one sample from the distribution. The language provides a library of sampling functions for common distributions, such as uniform, Gaussian, and Bernoulli distributions. Programmers may define sampling functions for new distributions using arbitrary code.

Programmers write specifications of correctness in passerts. The above passert is satisfied because the probability that a random sample from $\mathcal{U}(0,1)$ is less than 0.9 is exactly 90%.

To formalize this reasoning, we represent programs as Bayesian networks. A Bayesian network is a directed, acyclic graphical model wherein nodes represent random variables and edges represent conditional dependence between those random variables.



Much like an expression tree, each node in the Bayesian network corresponds to a value produced by the program. Unlike an expression tree, however, each node represents a distribution rather than a single value. This network, for example, contains three random variables (X, W, and $X < W$), one for each expression executed in the program (`random.uniform(0,1)`, `0.9`, and `x < w`). The directed edges represent how these random variables conditionally depend on one another. For example, the node for the random variable $X < W$ has edges from two other nodes: X and W.

Because each variable is dependent *only* on its parents in a Bayesian network, the probability distributions for each node are defined locally. In our example, the distribution for the $X < W$ node, a Bernoulli random variable, is:

$$\Pr[X < W \mid X \sim \mathcal{U}, W = 0.9]$$

Computing the distribution for $X < W$ requires only the distributions for its parents, X and W. In this case, both parents are leaves in the Bayesian network: a uniform distribution and a point-mass distribution.

One way to compute the distribution is to sample it. Sampling the root node consists of generating a sample at each leaf and then propagating the values

through the graph. Since Bayesian networks are acyclic, every node generates only a single value per sample and the running time of each sample is bounded.

In this example, we can exploit the Bayesian network formulation to simplify the graph and compute an exact solution without sampling. By definition, when X is uniformly distributed, for any constant $c \in [0, 1]$, $\Pr[X < c] = c$. Using this statistical knowledge, we replace the tree in our example with a single node representing a Bernoulli distribution with probability 0.9.

The Bayesian network abstraction for probabilistic programs yields two major advantages. First, it gives a probabilistic semantics to programs and passert statements. Appendix C formalizes our probabilistic semantics and proves that sampling the Bayesian representation is equivalent to sampling the original program. Second, we exploit probabilistic algebras and statistical properties of random variables to optimize the verification process. In some cases, we verify passerts without sampling. Section 5.4.1 introduces these optimizations.

5.3 DISTRIBUTION EXTRACTION

Given a program with a passert e and either a concrete input or a distribution over inputs, MAYHAP performs a probabilistic evaluation by building and optimizing a Bayesian-network representation of the statements required to evaluate the passert. This section describes distribution extraction, which is the first step in this process. Distribution extraction produces a symbolic Bayesian network representation that corresponds to the slice of the program contributing to e. MAYHAP treats randomness as symbolic and deterministic components as concrete. The process is similar to symbolic execution and to lazy evaluation in functional languages.

Distribution extraction produces a Bayesian network that is equivalent to the original program but is more amenable to statistical optimizations (enumerated in the next section). Appendix C formalizes the process and proves an equivalence theorem.

DISTRIBUTIONS AS SYMBOLIC VALUES MAYHAP performs a forward pass over the program, concretely evaluating deterministic computations and introducing symbolic values—probability-distribution expression trees—to represent probabilistic values. For example, the following statement:

```
a = b + 2
```

computes a concretely when b is not probabilistic. If, prior to the above statement, the program assigns `b = 5`, then we perform the addition and set $a = 7$. However, if `b = gaussian()`, we add a node to the Bayesian network, representing b symbolically as a Gaussian distribution. We then create a sum node for a with two parents: b's Gaussian and 2's constant (point mass) distribution.

As this mixed symbolic and concrete execution proceeds, it eagerly evaluates any purely deterministic statements but builds a Bayesian-network representation of the forward slice of any probabilistic statements. This process embodies a symbolic execution in which the symbolic values are probability distributions.

Our approach differs from typical symbolic execution in how it handles control flow (see below).

When the analysis reaches a statement passert e, the tool records the Bayesian network rooted at e. It then optimizes the network and samples the resulting distribution. Compared to sampling the entire program repeatedly, sampling the extracted distribution can be more efficient even without optimizations since it eliminates redundant, non-probabilistic computation.

CONDITIONALS When conditionals and loops are based on purely concrete values, distribution extraction proceeds down one side of the control flow branch as usual. When conditions operate on probabilistic variables, the analysis must capture the effect of both branch directions.

To analyze the probability distribution of a conditional statement, we produce conditional probabilities based on control-flow divergence. For example, consider this simple program:

```
if a:
  b = c
else:
  b = d
```

in which a is probabilistic. Even if both c and d are discrete, the value of b is probabilistic since it depends on the value of a. We can write the conditional probability distributions $\Pr[B]$ for b conditioned on both possible values for a:

$$\Pr[B \mid A = \text{true}] = \Pr[C]$$
$$\Pr[B \mid A = \text{false}] = \Pr[D]$$

Instead, to enable more straightforward analysis, we *marginalize* the condition a to produce an unconditional distribution for b. Using marginalization, we write the unconditional distribution $\Pr[B]$ as:

$$\begin{aligned}\Pr[B] &= \sum_a \Pr[B \mid A = a] \Pr[A = a] \\ &= \Pr[B \mid A = \text{true}] \cdot \Pr[A = \text{true}] \\ &\quad + \Pr[B \mid A = \text{false}] \cdot \Pr[A = \text{false}] \\ &= \Pr[C] \cdot \Pr[A = \text{true}] + \Pr[D] \cdot (1 - \Pr[A = \text{true}])\end{aligned}$$

This expression computes the distribution for b as a function of the distributions for a, c, and d. Intuitively, the probabilistic evaluation rewrites the condition to read `b = a * c + (1 - a) * d`. This algebraic representation enables some optimizations described in Section 5.4.1.

LOOPS AND EXTERNAL CODE Loops with probabilistic conditions can, in general, run for an unbounded number of iterations. Representing unbounded execution would induce cycles in our graphical model and violate the acyclic definition of a Bayesian network. For example, consider a loop that accumulates samples and exits when the sum reaches a threshold:

```
v = 0.0
while v < 10.0:
  v += random.uniform(-0.5, 1.0)
```

If the random sample is negative in every iteration, then the loop will never exit. The probability of this divergence is small but non-zero.

Prior work has dealt with probabilistic loops by restricting the program to linear operators [184]. MAYHAP relaxes this assumption by treating a loop as a black box that generates samples (i.e., the loop may run for an unbounded but finite number of iterations), similar to a known probability distribution such as `random.uniform`. This representation avoids creating cycles. In particular, MAYHAP represents a loop body with a *summary node*, where variables read by the loop are edges *into* the node and variables written by the loop are edges *out* of the node.

In practice, many loops have non-probabilistic bounds. For example, we evaluated an image filter (sobel) that loops over the pixel array and applies a probabilistic convolution to each window. The nested loops resemble:

```
for x in 0..width:
  for y in 0..height:
    filter(image[x][y])
```

While the computed pixel array contains probabilistic data, the dimensions `width` and `height` are fixed for a particular image. MAYHAP extracts complete distributions from these common concrete-bounded loops without black-box sampling.

MAYHAP uses a similar black-box mechanism when interfacing with library code whose implementation is not available for analysis—for example, when passing a probabilistic value to the `cos()` function from the C standard library. This straightforward approach prevents statistical optimizations inside the library function or loop body but lets MAYHAP analyze more programs.

ANALYZING LOOPS WITH PROBABILISTIC PATH PRUNING We propose another way to analyze loops with probabilistic bounds by building on the path pruning techniques used in traditional symbolic execution. Typically, path pruning works by proving that some paths are infeasible. If the analysis determines that a path constraint is unsatisfiable, it halts exploration of that path. Probabilistic evaluation instead needs to discover when a given path is *unlikely* rather than impossible, i.e., when the conditions that lead to following this path at run time have a probability that falls below a threshold. We propose tracking a path probability expression for each explored path and periodically sampling these distributions to prune unlikely paths. This extension handles general probabilistic control flow in programs that are likely to terminate eventually. Intuitively, the more iterations the loop executes, the less likely it is to execute another iteration. Programs with a significant probability of running forever before reaching a passert can still prevent the analysis from terminating, but this behavior likely indicates a bug. We leave the evaluation of this more precise analysis to future work.

5.4 OPTIMIZATION AND HYPOTHESIS TESTING

To verify a conditional in a passert, probabilistic evaluation extracts a symbolic representation of the conditional, optimizes this representation, and evaluates the conditional. The previous sections described the distribution extraction step and this section describes our optimization and evaluation steps.

Optimizations simplify the Bayesian network by applying known statistical properties to make verification more efficient. In restricted cases, these optimizations simplify the Bayesian network to a closed-form Bernoulli representing the condition in the passert and we thus evaluate the passert exactly. In the general case, we use sampling and hypothesis testing to verify it statistically.

5.4.1 *Optimizing Bayesian Networks*

This section enumerates the statistical properties that MAYHAP applies to simplify distributions.

CLOSED-FORM OPERATIONS OVER KNOWN DISTRIBUTIONS MAYHAP exploits closed-form algebraic operations on the common Gaussian, uniform, and Bernoulli distributions. For example, if $X \sim N(\mu_x, \sigma_x^2)$ and $Y \sim N(\mu_y, \sigma_y^2)$ then $X + Y \sim N(\mu_x + \mu_y, \sigma_x^2 + \sigma_y^2)$. Likewise, if $X \sim N(\mu_x, \sigma_x^2)$ then $X + 3 \sim N(\mu_x + 3, \sigma_x^2)$. MAYHAP optimizes closed form addition of Gaussians and scalar shifts or scaling of Gaussians, uniforms, and Bernoullis. We note there are many distributions and operations which we do not yet encode (e.g., a sum of uniform distributions is an Irwin–Hall distribution). Expanding the framework to capture a larger catalog of statistical properties is left to future work.

INEQUALITIES OVER KNOWN DISTRIBUTIONS MAYHAP uses the cumulative distribution function (CDF) for known distributions to simplify inequalities. The CDF for a real-valued random variable X is the function F_X such that $F_X(x) = \Pr[X < x]$, which provides a closed-form mechanism to evaluate whether a distribution is less than a constant. For example, if $X \sim U(0,1)$ and the programmer writes the inequality $X < 0.9$, we reduce the inequality to a Bernoulli because $F_{Uniform}(0.9) = \Pr[X < 0.9] = 0.9$.

CENTRAL LIMIT THEOREM The sum of a large number of independent random variables with finite variance tends to a Gaussian. MAYHAP uses the Central Limit Theorem to reduce loops which compute a reduction over random variables into a closed-form Gaussian which samples from the body of the loop. This transformation resembles Misailovic et al.'s "mean pattern" [132]. It is particularly effective on the sobel application used in our evaluation, which averages the errors for each pixel in an array. MAYHAP reduces this accumulation to a single Gaussian.

EXPECTATION PROPAGATION The prior optimizations approximately preserve a program's semantics: the transformed Bayesian network is approximately equivalent to the original Bayesian network. However, using statistical laws that apply to inequalities over random variables, it suffices to instead compute only the expected value and variance of a distribution. MAYHAP uses this insight to further simplify Bayesian networks by exploiting (1) the linearity of expected value and (2) statistical properties of inequality.

First, MAYHAP uses the linearity of expectation to produce simpler distributions with the same expected value as the original distribution. This is an important optimization because verifying a passert amounts to calculating the expected value of its underlying Bernoulli distribution. For example, the Bayesian network for $D + D$, which computes two independent samples from D, is not equivalent to the Bayesian network induced from $2 \cdot D$. So an optimization resembling traditional strength reduction does not compute the correct distribution. However, these two Bayesian networks have the same expected value. Specifically, expectation has the property $\mathrm{E}[A + B] = \mathrm{E}[A] + \mathrm{E}[B]$ for all distributions A and B. When only the expected value is needed, MAYHAP optimizes $D + D$ to $2 \cdot D$. A similar property holds for variance when the random variables are uncorrelated.

The reasoning extends to comparisons via Chebyshev's inequality. Given the expectation μ and variance σ^2 of a random variable, Chebyshev's inequality gives a bound on the probability that a sample of a random variable deviates by a given number of standard deviations from its expected value. For example, for a program with **passert** x >= 5, distribution extraction produces a Bayesian network of the form $X \geq 5$. Using the linearity of expectation, suppose we statically compute that $\sigma = 3$ and $\mu = 1$ for X. Chebyshev's inequality states:

$$\Pr[|X - \mu| \geq k\sigma] \leq \frac{1}{k^2}$$

We want to bound the probability that $x \geq 5$. Since we have μ and σ, we can rewrite this condition as:

$$\begin{aligned} x &\geq \mu + 2\sigma \\ x - \mu &\geq 2\sigma \end{aligned}$$

So the passert condition states that x deviates from its mean by at least 2 standard deviations. Using $k = 2$ in Chebyshev's inequality gives the bound:

$$\Pr[X \geq 5] \leq \frac{1}{2^2}$$

We now have a bound on the probability (and hence the expectation) of the inequality x >= 5.

5.4.2 *Verification*

This section describes how we use an extracted and simplified Bayesian network to verify passerts using (1) exact (direct) evaluation or (2) sampling and statistical hypothesis testing.

5.4.2.1 *Direct Evaluation*

In some cases, simplifications on the probability distribution are sufficient to fully evaluate a passert. For example, MAYHAP simplifies the sobel application in our evaluation to produce a distribution of the form $\sum_n D < c$. The Central Limit Theorem optimization replaces the sum with a Gaussian distribution, which then enables the inequality computation to produce a simple Bernoulli distribution with a known probability. When dealing with a single Bernoulli, no sampling is necessary. MAYHAP reports the probability from the simplified distribution.

5.4.2.2 *Statistical Verification via Sampling*

In the general case, optimizations do not completely collapse a probability distribution. Instead, MAYHAP samples the resulting distribution to estimate its probability.

MAYHAP uses acceptance sampling to bound any error in its verification [235]. All passert statements are logical properties over random variables and therefore Bernoulli random variables. Assume $X_i \sim \text{Bernoulli}(p)$ is an independent sample of a passert where p is the *true* probability of the passert, the value MAYHAP is estimating. Let $X = X_1 + X_2 + \ldots + X_n$ be the sum of n independent samples of the passert and let the empirical expected value, $\mathrm{E}[X] = \overline{X} = X/n$, be an estimate of p.[1] To bound error in its estimate, MAYHAP computes $\Pr\left[\overline{X} \in [p-\epsilon, p+\epsilon]\right] \geq 1-\alpha$. In words, it tests whether there is at most an α chance that MAYHAP's estimate of p is wrong. Otherwise, MAYHAP's estimate of p is within ϵ of the truth. A programmer can control the likelihood of a good estimate—or the *confidence*—by decreasing α. Likewise, a programmer can control the *accuracy* of the estimate by decreasing ϵ. Because MAYHAP uses sampling, it provides statistical guarantees by testing whether its confidence interval for $\overline{X}$ includes $p \pm \epsilon$. In concert, these parameters let a programmer trade off false-positives and false-negatives with sample size.

In particular, given α and ϵ, MAYHAP uses the two-sided Chernoff bound to compute n, the minimum number of samples required to satisfy a given level of confidence and accuracy [41]. The two-sided Chernoff bound is an upper-bound on the probability that an estimate, $\overline{X}$, deviates from its true mean, p:

$$\Pr\left[|\overline{X} - p| \geq \epsilon p\right] \leq 2e^{-\frac{\epsilon^2}{2+\epsilon}np}$$

The left-hand side of the equality is α by definition and the worst case (the most samples required) occurs when $p = 1$. Solving for n yields:

$$n \geq \frac{2+\epsilon}{\epsilon^2} \ln \frac{2}{\alpha}$$

For example, at a confidence 95% and an accuracy of 3%:

$$n \geq \frac{2+0.03}{0.03^2} \ln \frac{2}{0.05}$$

1 This section uses $\overline{X}$ instead of $\mathrm{E}[X]$ for notational convenience.

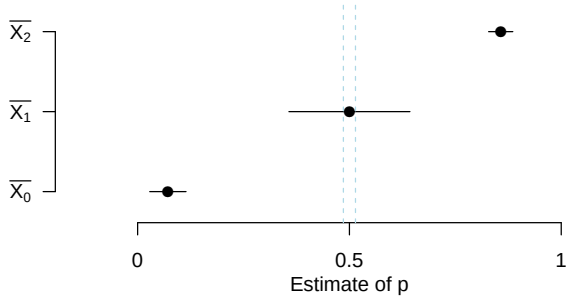


Figure 11: Hypothesis tests for three different passert statements.

meaning that MAYHAP needs to take at least $n = 8321$ samples. Note that this bound is an over-approximation of the true number of samples required for a given level of confidence and accuracy—it only relies on α and ϵ and ignores how good an estimate $\overline{X}$ is of p. An extension, which we leave to future work, is to use Wald's *sequential sampling* to iteratively compute $\Pr\left[\overline{X} \in [p-\epsilon, p+\epsilon]\right] \geq 1-\alpha$ after each sample [221]. Because this approach uses the current estimate of $\overline{X}$ relative to p, it is often able to stop sampling well before reaching our upper bound [234].

STATISTICAL GUARANTEES MAYHAP turns a passert statement into a hypothesis test in order to bound error in its estimate. If the property is sufficiently likely to hold, MAYHAP verifies the passert as true. Likewise, if the passert is verified as false, the programmer needs to iterate, either by changing the program to meet the desired specification or by correctly expressing the probabilistic property of the program.

For example, suppose MAYHAP estimates $\Pr\left[\overline{X}_i \in [p-\epsilon, p+\epsilon]\right] \geq 1-\alpha$ for three distinct, hypothetical passert statements (i.e., $i \in [0,1,2]$). We pictorially show these three estimates in Figure 11. Each estimate shows $\overline{X}_i$ as a point and lines depict the confidence region of that estimate. Because the confidence region of $\overline{X}_0$ is below 0.5, MAYHAP verifies this assertion as false (i.e., the passert rarely holds). Likewise, because $\overline{X}_2 - \epsilon \geq 0.5$, MAYHAP verifies this assertion as true (i.e., the passert often holds).

However, at this confidence level and accuracy, MAYHAP is unable to verify $\overline{X}_1$ as its confidence region and thus estimate *overlaps* with $0.5 \pm \epsilon$. Thus, MAYHAP labels this assertion as unverifiable. To verify this assertion as true or false, the programmer must increase either the confidence or accuracy (or both). In this situation, MAYHAP initiates a dialog with the programmer for guidance on how to proceed.

5.5 IMPLEMENTATION

We implemented MAYHAP using the LLVM compiler infrastructure [101]. MAYHAP compiles source programs written in C and C++ to the LLVM intermediate language, probabilistically evaluates the resulting bitcode programs by extract-

ing probability distributions, optimizes the resulting distributions, and then evaluates the passert distributions either exactly or with sampling.

LANGUAGE AND INTERFACE To use the verifier system, the programmer adds a passert to her program and annotates certain functions as probability distributions or uses a provided library of common distributions. Both constructs are implemented as C macros provided by a `passert.h` header: `PASSERT(e)` marks an expression that MAYHAP will evaluate and `DISTRIBUTION` marks functions that should be treated as a symbolic probability distribution.

To invoke MAYHAP, the programmer provides arguments comprising the source files, command-line arguments for the program under test, and optional α and ϵ values that control confidence and accuracy. MAYHAP reports a confidence interval on the output probability and the results of the hypothesis test (true, false, or unverifiable).

DISTRIBUTION EXTRACTION The distribution extraction analysis is implemented as an instrumented interpreter of LLVM bitcode programs. MAYHAP maintains a symbolic heap and stack. Each symbolic value is a pointer into an object graph representing a Bayesian network. Nodes in the graph correspond to the expression tree language of our formalism: they can be samples, arithmetic operations, comparisons, constants, or conditionals.

The implementation conserves space by coalescing identical expression trees. For example, consider the values $e_1 = \{s_1 + s_2\}$ and $e_2 = \{(s_1 + s_2) + s_3\}$ consisting of sums of samples. In a naive implementation of probabilistic evaluation, these would be independent trees that refer to a global set of samples at their leaves. Instead, MAYHAP implements e_2 as a sum node with two children, one of which is the node for e_1. In this sense, MAYHAP maintains a global Bayesian network for the execution in which values are pointers into the network.

Nodes in the Bayesian network can become unreachable when heap values are overwritten and as stack frames are popped. MAYHAP reclaims memory in these cases by reference-counting all nodes in the Bayesian network. The root set consists of stack and heap values. Since Bayesian networks are acyclic, reference counting is sufficient.

When operating on non-probabilistic values (e.g., when evaluating $1 + 2$), MAYHAP avoids constructing nodes in the Bayesian network and instead maintains a concrete heap and stack. We use LLVM's bitcode interpreter [115] to perform the concrete operations. This process can be viewed as an optimization on Bayesian networks for operations over point-mass distributions.

CONDITIONALS Conditionals appear as branches in LLVM IR. MAYHAP analyzes conditionals by symbolically executing both sides of the branch and merging the resulting heap updates. When the analysis encounters a branch, it finds the immediate post-dominator (ipdom) in the control-flow graph—intuitively, the join point—and begins by taking the branch. In this phase, it buffers all heap writes in a (scoped) hash table. Then, when the ipdom is reached, control returns to the branch and follows the not-taken direction. Writes in this phase do not

		Time (seconds)			Optimization Counts		
Program	Description and passert	Baseline	Analysis	Sampling	Arith	Dist Op	CLT
gpswalk	Location sensing and velocity calculation passert: Velocity is within normal walking speed	537.0	1.6	59.0	1914	0	1
salary	Calculate average of concrete obfuscated salaries passert: Obfuscated mean is close to true mean	150.0	2.5	< 0.1	3	1	1
salary-abs	salary with abstract salaries drawn from a distribution passert: As above	87.0	20.0	0.2	5003	1	1
kmeans	Approximate clustering passert: Total distance is within threshold	1.8	0.3	< 0.1	2149	300	0
sobel	Approximate image filter passert: Average pixel difference is small	37.0	2.8	< 0.1	7880	0	1
hotspot	Approximate CMOS thermal simulation passert: Temperature error is low	422.0	4.7	28.0	1	24064	1
inversek2j	Approximate robotics control passert: Computed angles are close to inputs	4.8	< 0.1	< 0.1	901	200	1

Table 5: Programs used in the evaluation. The passert for each application describes a probabilistic correctness property. The *time* columns indicate the time taken by the baseline stress-testing strategy, MAYHAP's analysis, and MAYHAP's sampling step. The *optimization counts* reflect the three categories of optimizations applied by MAYHAP: arithmetic identities (Arith), operations on known closed-form distributions (Dist Op), and the Central Limit Theorem optimization (CLT).

go into the scope for the current conditional: they propagate to the global heap or, if execution is in a nested outer conditional, to the enclosing hash table scope. When the ipdom is reached the second time, the buffered writes are merged into the outer heap using conditional nodes.

PROBABILISTIC POINTERS MAYHAP partially supports symbolic pointers for probabilistic array indexing. Programs can load and store from `arr[i]` where `i` is probabilistic, which MAYHAP handles with a probabilistic extension of the theory of arrays. Pointers and array indices must be finite discrete distributions so we can enumerate the set of locations to which a pointer p might refer, i.e., those addresses where p's distribution has non-zero probability. Loading from a symbolic pointer p yields a distribution that reflects the set of values at each such location, while storing to p updates each location to compose its old and new value under a conditional distribution.

BAYESIAN NETWORK OPTIMIZATIONS MAYHAP performs statistical optimizations as transformations on the Bayesian network representation as outlined in Section 5.4.1. The optimizations we implemented fall into three broad categories, which we characterize empirically in the next section.

The first category consists of arithmetic identities, including binary operators on constants, comparisons with extremes (e.g., C's `FLT_MAX`), and addition or multiplication with a constant zero. These optimizations do not exploit the probabilistic properties of the Bayesian network but compose with more sophisticated optimizations and enhance the tool's partial-evaluation effect. The next category consists of operations on known probability distributions, including the addition of two normal distributions, addition or multiplication with a scalar, comparison between distributions with disjoint support, comparison between two uniform distributions, and comparison with a scalar (i.e., CDF queries). These optimizations exploit our probabilistic view of the program to apply well-known statistical properties of common distributions. The final optimization we evaluate is the Central Limit Theorem, which collapses a summation of distributions into a single normal.

Some optimizations, such as basic arithmetic identities, are performed opportunistically on-the-fly during analysis to reduce MAYHAP's memory footprint. Others, such as the Central Limit Theorem transformation, operate only on the complete graph. Internally, the on-line optimizer also collapses deep trees of commutative arithmetic operators into "fat" sum and product nodes with many children. This rewriting helps the optimizer identify constants that can be coalesced and inverse nodes that cancel each other out.

VERIFICATION As described in Section 5.4.2, the prior optimizations often produce Bayesian networks that MAYHAP can directly evaluate. In other cases, MAYHAP must sample the optimized Bayesian network, in which case MAYHAP generates LLVM bitcode that samples from the Bayesian network. The tool then compiles the generated program to machine code and executes it repeatedly to perform statistical verification.

5.6 EVALUATION

This section describes our experience expressing passerts in a variety of probabilistic programs and using MAYHAP to verify them.

5.6.1 *Benchmarks*

We evaluate passerts in five probabilistic programs from three domains: sensors, differential privacy, and approximate computing. Table 5 summarizes the set of programs and the passert statements we added to each.

Programs that compute with noisy sensor data, such as GPS, accelerometers, and video game motion sensors, behave probabilistically [21, 149]. To demonstrate our approach on this class of applications, we implemented a common mobile-phone application: estimating walking speed [21]. gpswalk processes a series of noisy coordinate readings from a mobile phone and computes the walking speed after each reading. The GPS error follows a Rayleigh distribution and is determined by the sensor's uncertainty estimate. As Bornholt et al. [21] note, this kind of sensing workload can produce wild results when an individual location reading is wrong. The passert checks that the computed velocity is below a maximum walking speed.

Differential privacy obfuscates personal data at the cost of accuracy [15, 122, 135, 163, 170]. To study how MAYHAP works on this class of application, we implemented two benchmarks. salary reads a list of 5000 salaries of Washington state public employees and computes their average.[2] The program obfuscates each salary by adding a normal distribution ($\sigma^2 = 3000$) to simulate a situation where each employee is unwilling to divulge his or her exact salary. The passert checks whether the obfuscated average is within 25 dollars of the true average. We also evaluate a version of the program, salary-abs, where the input salaries are drawn from a uniform distribution instead of read from a file. This variant highlights a scenario where specific inputs are unavailable and we instead want to check a passert given an input probability distribution.

The final class of applications is drawn from prior work on approximate computing: kmeans, sobel, hotspot, and inversek2j represent programs running on approximate hardware [34, 59, 139]. sobel implements the Sobel filter, an image convolution used in edge detection. kmeans is a clustering algorithm. hotspot simulates thermal activity on a microprocessor. inversek2j uses inverse kinematics to compute a robotic arm's joint angles given a target position. Both kmeans and hotspot are drawn from the Rodinia 2.4 benchmark suite [39] while sobel and inversek2j are approximate applications from Esmaeilzadeh et al. [60]. In all cases, we add random calls that simulate approximate arithmetic operations on inner computations. The passert bounds the error of the program's overall output. For most benchmarks, the error is measured with respect to the output of a precise version of the computation, but in inversek2j, we use the corresponding forward kinematics algorithm to check the result.

2 Source: `http://fiscal.wa.gov/`

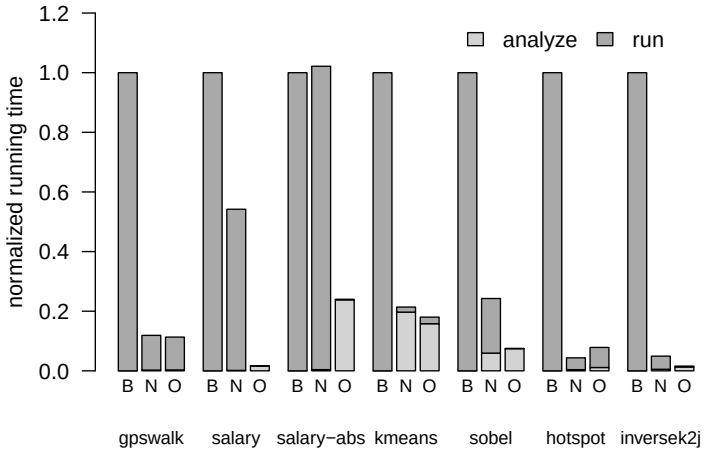


Figure 12: MAYHAP reduces testing time. We normalize to B: the baseline stress-testing technique with 74147 samples. N is MAYHAP without optimizations and O is MAYHAP with optimizations, divided into analysis and execution time. Times are averaged over 5 executions. We elide error bars as they are very small.

For both approximate and data privacy programs, we compare a precise version of a function's output with a perturbed version. In the sensing workload, gpswalk, the data is intrinsically noisy, so there is no "ground truth" to compare against. For the purposes of this evaluation, we manually extended the programs to compute both results. A simple "desugaring" step could help perform this transformation automatically by duplicating the code, removing randomization from one copy, and returning both results.

5.6.2 *Performance*

To evaluate MAYHAP's performance benefits, we compare its total running time against using a simple stress testing baseline. The baseline checker adds a `for` loop around the entire probabilistic program and counts the number of times the passert expression is true. The time taken for a MAYHAP verification includes the time to extract and optimize a probability distribution and to repeatedly sample the result. We test all programs with a confidence of $\alpha = 0.05$ and an accuracy of $\epsilon = 0.01$, which leads to 74147 samples. (Recall from Section 5.4.2.2 that the sample count depends only on the α and ϵ parameters and so we sample all programs the same number of times.) Table 5 lists the absolute running times and Figure 12 visualizes the normalized performance. The timings are averaged over 5 executions collected on a dual-core 2 GHz Intel Core 2 Duo with 4 GB of memory. On average, MAYHAP verification takes 4.2% as long as the strawman checker, for a speedup of 24×.

For most benchmarks, MAYHAP's time is almost exclusively spent on distribution extraction and optimization, which means optimizations are effective at producing a very small distribution that can be sampled much more efficiently than the original program. The exception is gpswalk, where the analysis exe-

cuted in 1.6 seconds but sampling the resulting distribution took over a minute. That program's probability distribution consists of thousands of independent Rayleigh distributions, each with a different parameter as reported by the GPS sensor, so it cannot take advantage of optimizations that exploit many samples from identical distributions.

EFFECT OF OPTIMIZATIONS We evaluated a variant of MAYHAP with optimizations disabled. This version simply performs distribution extraction and samples the resulting distribution. The middle bars labeled N in Figure 12 show the normalized running time of this non-optimizing MAYHAP variant.

The effectiveness of the optimizations varies among the benchmarks. On one extreme, optimizations reduce the execution time for salary from 81 seconds to a fraction of a second. The unoptimized Bayesian network for salary-abs is slightly *less* efficient than the original program. The Central Limit Theorem optimization applies to both and greatly reduces the amount of sampled computation. On the other hand, simply evaluating the extracted distribution delivers a benefit for gpswalk, reducing 537.0 to 62 seconds and then optimizations further reduce this time to just 59.0 seconds. In a more extreme case, enabling optimizations adds to the analysis time for hotspot but fails to reduce its sampling time. These programs benefit from eliminating the deterministic computations involved in timestamp parsing and distance calculation.

CONFIDENCE--PERFORMANCE TRADE-OFF Via the confidence and accuracy parameters α and ϵ, MAYHAP provides rough estimates quickly or more accurate evaluations using more samples. To evaluate this trade-off, we lowered the parameter settings, $\alpha = 0.10$ and $\epsilon = 0.05$, which leads to 2457 samples (about 3% compared to the more accurate settings above). Even accounting for analysis time, MAYHAP yields a harmonic mean 2.3× speedup over the baseline in this relaxed configuration.

5.7 DISCUSSION

Probabilistic assertions express quality constraints, not only for approximate programming but for any computational domain that uses randomness to do its work. In contrast to the other quality-focused work in this dissertation, the probabilistic assertion verification workflow in this chapter makes the closest connections to traditional statistical reasoning. It is also the most general approach: the techniques applies to "probabilistic programming languages" as defined by Kozen [95]: ordinary languages extended with random calls. In exchange for its generality, the approach makes weaker guarantees than, for example, Chapter 4's conservative probability bounds: the basis in sampling always leaves room for false positives. A healthy ecosystem for approximate programming will need techniques from across the strength–generality continuum.

Part III

APPROXIMATE SYSTEMS

6

APPROXIMATE STORAGE

6.1 INTRODUCTION

The majority of work on approximate system architectures focuses on computation [34, 59, 97, 103, 139, 180]. The idea of accuracy–efficiency trade-offs extends naturally to storage: error tolerance in both transient and persistent data is present in a broad range of application domains, from server software to mobile applications.

Meanwhile, the semiconductor industry is beginning to encounter limits to further scaling of common memory technologies like DRAM and flash memory. As a result, new memory technologies and techniques are emerging. Multi-level cells, which pack more than one bit of information in a single cell, are already commonplace and phase-change memory (PCM) is imminent. But both PCM and flash memory wear out over time as cells degrade and become unusable. Furthermore, multi-level cells are slower to write due to the need for tightly controlled iterative programming.

Memories traditionally address wear-out issues and implement multi-level cell operation in ways that ensure perfect data integrity 100% of the time. This has significant costs in performance, energy, area, and complexity. These costs are exacerbated as memories move to smaller device feature sizes along with more process variation. By relaxing the requirement for perfectly precise storage—and exploiting the inherent error tolerance of approximate applications—failure-prone and multi-level memories can gain back performance, energy, and capacity.

We propose techniques that exploit data accuracy trade-offs to provide *approximate storage*. In essence, we advocate exposing storage errors up to the application with the goal of making data storage more efficient. We make this safe by: (1) exploiting application-level inherent tolerance to inaccuracies; and (2) providing an interface that lets the application control which pieces of data can be subject to inaccuracies while offering error-free operation for the rest of the data. We propose two basic techniques. The first technique uses multi-level cells in a way that enables higher density or better performance at the cost of occasional inaccurate data retrieval. The second technique uses blocks with failed bits to store approximate data; to mitigate the effect of failed bits on overall value precision, we prioritize the correction of higher-order bits.

Approximate storage applies to both persistent storage (files or databases) as well as transient data stored in main memory. We explore the techniques in the

context of PCM, which may be used for persistent storage (replacing hard disks) or as main memory (replacing DRAM) [102, 156, 237], but the techniques generalize to other technologies such as flash memory. We simulate main-memory benchmarks and persistent-storage datasets and find that our techniques improve write latencies by 1.7× or extend device lifetime by 23% on average while trading off less than 10% of each application's output quality.

6.2 INTERFACES FOR APPROXIMATE STORAGE

While previous work has considered reducing the energy spent on DRAM and SRAM storage [59, 113, 180], modern non-volatile memory technologies also exhibit properties that make them candidates for storing data approximately. By exploiting the synergy between these properties and application-level error tolerance, we can alleviate some of these technologies' limitations: limited device lifetime, low density, and slow writes.

Approximate storage augments memory modules with software-visible precision modes. When an application needs strict data fidelity, it uses traditional *precise* storage; the memory then guarantees a low error rate when recovering the data. When the application can tolerate occasional errors in some data, it uses the memory's *approximate* mode, in which data recovery errors may occur with non-negligible probability.

This work examines the potential for approximate storage in PCM and other solid-state, non-volatile memories. For both categories of data, the application must determine which data can tolerate errors and which data needs "perfect" fidelity. Following EnerJ's example, we assume that safety constraints need to be part of the programming model, since approximating data indiscriminately can easily lead to broken systems (see Section 1.2.2).

The next sections describe the approximation-aware programming models for main memory and persistent mass storage along with the hardware–software interface features common to both settings. In general, each block (of some appropriate granularity) is logically in either *precise* or *approximate* state at any given time. Every read and write operation specifies whether the access is approximate or precise. These per-request precision flags allow the storage array to avoid the overhead of storing per-block metadata. The compiler and runtime are responsible for keeping track of which locations hold approximate data. Additionally, the interface may also allow software to convey the relative importance of bits within a block, enabling more significant bits to be stored with higher accuracy.

As the evaluation of EnerJ in Chapter 3 found, different applications can tolerate different error rates. And while applications could likely exploit approximation most effectively by using specific error rates for specific data items, as in DECAF (Chapter 4), we explore a simpler hardware–software interface that applies a uniform policy to all approximate data in the program. Empirically, we find that this level of control is sufficient to achieve good quality trade-offs for the applications we evaluate.

6.2.1 *Approximate Main Memory*

PCM and other fast, resistive storage technologies may be used as main memories [102, 156, 237]. Previous work on approximate computing has examined applications with error-tolerant memory in the context of approximate DRAM and on-chip SRAM [59, 113, 180]. This work has found that a wide variety of applications, from image processing to scientific computing, have large amounts of error-tolerant stack and heap data. We extend the programming model and hardware–software interfaces developed by this previous work for our approximate storage techniques.

Programs specify data elements' precision at the programming language level using EnerJ's type annotation system [180]. Using these types, the compiler can statically determine whether each memory access is approximate or precise. Accordingly, it emits load and store instructions with a precision flag as in the Truffle ISA [59].

6.2.2 *Approximate Persistent Storage*

We also consider interfaces for persistent storage: filesystems, database management systems (DBMSs), or, more recently, flat address spaces [47, 220].

Use cases for approximate mass storage range from server to mobile and embedded settings. A datacenter-scale image or video search database, for example, requires vast amounts of fast persistent storage. If occasional pixel errors are acceptable, approximate storage can reduce costs by increasing the capacity and lifetime of each storage module while improving performance and energy efficiency. On a mobile device, a context-aware application may need to log many days of sensor readings to model user behavior. Here, approximate storage can help relieve capacity constraints or, by reducing the cost of accesses, conserve battery life.

We assume a storage interface resembling a key–value store or a flat address space with smart pointers (e.g., NV-heaps [47] or Mnemosyne [220]), although the design also applies to more complex interfaces like filesystems and relational databases. Each object in the store is either approximate or precise. The precision level is set when the object is created (and space is allocated). This constant precision level matches the software model for current approximation-aware programming interfaces such as EnerJ [180].

6.2.3 *Hardware Interface and Allocation*

In both deployment scenarios, the interface to approximate memory consists of read and write operations augmented with a precision flag. In the main-memory case, these operations are load and store instructions (resembling Truffle's `stl.a` and `ldl.a` [59]). In the persistent storage case, these are blockwise read and write requests.

The memory interface specifies a granularity at which approximation is controlled. In PCM, for example, this granularity may be a 512-bit block. The com-

piler and allocator ensure that precise data is always stored in precise blocks. (It is safe to store approximate data in precise blocks.)

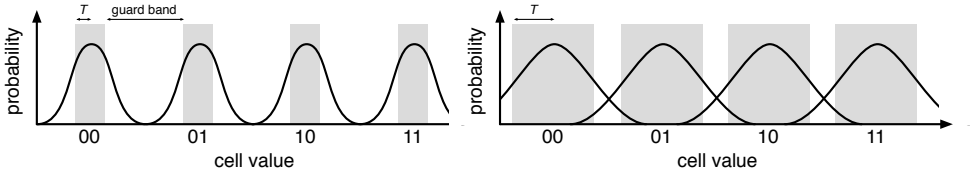
To maintain this property, the allocator uses two mechanisms depending on whether the memory supports software control over approximation. With software control, as in Section 6.3, the program sets the precision state of each block implicitly via the flags on write instructions. (Reads do not affect the precision state.) In a hardware-controlled setting, as in Section 6.4, the operating system maintains a list of approximate blocks reported by the hardware. The allocator consults this list when reserving space for new objects. Section 6.4.2 describes this OS interface in more detail.

In the main memory case and when using object-based persistent stores like NV-heaps [47], objects may consist of interleaved precise and approximate data. To support mixed-precision objects, the memory allocator must lay out fields across precise and approximate blocks. To accomplish this, the allocator can use one of two possible policies: *ordered layout* or *forwarding pointers*. In ordered layout, heterogeneous objects lay out their precise fields (and object header) first; approximate fields appear at the end of the object. When an object's range of bytes crosses one or more block boundaries, the blocks that only contain approximate fields may be marked as approximate. The prefix of blocks that contain at least one precise byte must conservatively remain precise. With forwarding pointers, in contrast, objects are always stored in precise memory but contain a pointer to approximate memory where the approximate fields are stored. This approach incurs an extra memory indirection and the space overhead of a single pointer per heterogeneous object but can reduce fragmentation for small objects.

To specify the relative priority of bits within a block, accesses can also include a data element size. The block is then assumed to contain a homogenous array of values of this size; in each element, the highest-order bits are most important. For example, if a program stores an array of double-precision floating point numbers in a block, it can specify a data element size of 8 bytes. The memory will prioritize the precision of each number's sign bit and exponent over its mantissa in decreasing bit order. Bit priority helps the memory decide where to expend its error protection resources to minimize the magnitude of errors when they occur.

6.3 APPROXIMATE MULTI-LEVEL CELLS

PCM and other solid-state memories work by storing an analog value—resistance, in PCM's case—and quantizing it to expose digital storage. In multi-level cell (MLC) configurations, each cell stores multiple bits. For *precise* storage in MLC memory, there is a trade-off between access cost and density: a larger number of levels per cell requires more time and energy to access. Furthermore, protections against analog sources of error like drift can consume significant error correction overhead [146]. But, where perfect storage fidelity is not required, performance and density can be improved beyond what is possible under strict precision constraints.



(a) Precise MLC (b) Approximate MLC

Figure 13: The range of analog values in a precise (a) and approximate (b) four-level cell. The shaded areas are the target regions for writes to each level (the parameter T is half the width of a target region). Unshaded areas are *guard bands*. The curves show the probability of reading a given analog value after writing one of the levels. Approximate MLCs decrease guard bands so the probability distributions overlap.

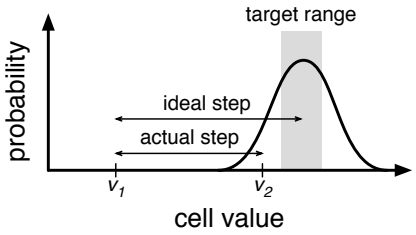


Figure 14: A single step in an iterative program-and-verify write. The value starts at v_1 and takes a step. The curve shows the probability distribution from which the ending value, v_2, is drawn. Here, since v_2 lies outside the target range, another step must be taken.

An *approximate MLC* configuration relaxes the strict precision constraints on iterative MLC writes to improve their performance and energy efficiency. Correspondingly, approximate MLC writes allow for denser cells under fixed energy or performance budgets. Since PCM's write speed is expected to be substantially slower than DRAM's, accelerating writes is critical to realizing PCM as a main-memory technology [102]. Reducing the energy spent on writes conserves battery power in mobile devices, where solid-state storage is commonplace.

Our approach to approximate MLC memory exploits the underlying analog medium used to implement digital storage. Analog reads and writes are inherently imprecise, so MLCs must incorporate *guard bands* that account for this imprecision and prevent storage errors. These guard bands lead to tighter tolerances on target values, which in turn limit the achievable write performance. Approximate MLCs reduce or eliminate guard bands to speed up iterative writes at the cost of occasional errors. Figure 13 illustrates this idea.

6.3.1 *Multi-Level Cell Model*

The basis for MLC storage is an underlying analog value (e.g., resistance for PCM or charge for flash memory). We consider this value to be continuous: while the memory quantizes the value to expose digital storage externally, the inter-

nal value is conceptually a real number between 0 and 1.[1] To implement digital storage, the cell has n discrete *levels*, which are internal analog-domain values corresponding to external digital-domain values. As a simplification, we assume that the levels are evenly distributed so that each level is the center of an equally-sized, non-overlapping band of values: the first level is $\frac{1}{2n}$, the second is $\frac{3}{2n}$, and so on. In practice, values can be distributed exponentially, rather than linearly, in a cell's resistance range [23, 145]; in this case, the abstract value space corresponds to the logarithm of the resistance. A cell with $n = 2$ levels is called a single-level cell (SLC) and any design with $n > 2$ levels is a multi-level cell (MLC).

Writes and reads to the analog substrate are imperfect. A write pulse, rather than adjusting the resistance by a precise amount, changes it according to a probability distribution. During reads, material nondeterminism causes the recovered value to differ slightly from the value originally stored and, over time, the stored value can change due to drift [231]. Traditional (fully precise) cells are designed to minimize the likelihood that write imprecision, read noise, or drift cause storage errors in the digital domain. That is, given any digital value, a write followed by a read recovers the same digital value with high probability.

Put more formally, let v be a cell's internal analog value. A write operation for a digital value d first determines l_d, the value level corresponding to d. Ideally, the write operation would set $v = l_d$ precisely. Realistically, it sets v to $w(l_d)$ where w is an error function introducing perturbations from the ideal analog value. Similarly, a read operation recovers a perturbed analog value $r(v)$ and quantizes it to obtain a digital output.

The number of levels, n, and the access error functions, w and r, determine the trade-off space of performance, density, and reliability for the cell.

WRITE ERROR FUNCTION A single programming pulse typically has poor precision due to process variation and nondeterministic material behavior. As a result, MLC designs for both flash and PCM adopt iterative program-and-verify (P&V) mechanisms [155, 203]. In PCM, each P&V iteration adjusts the cell's resistance and then reads it back to check whether the correct value was achieved. The process continues until an acceptable resistance value has been set. To model the latency and error characteristics of iterative writes, we consider the effect of each step to be drawn from a normal distribution. The write mechanism determines the ideal pulse size but applies that pulse with some error added. Figure 14 illustrates one iteration in this process.

Two parameters control the operation of the P&V write algorithm. First, iteration terminates when the stored value is within a threshold distance T from the target value. Setting $T < \frac{1}{2n}$ as in Figure 13 provides *guard bands* between the levels to account for read error. The value of T dictates the probability that a read error will occur. Second, the variance of the normal distribution governing the effect of each pulse is modeled as a constant proportion, P, of the intended

1 At small feature sizes, quantum effects may cause values to appear discrete rather than continuous. We do not consider these effects here.

```
def w(v_t):
  v = 0
  while |v_t − r(v)| > T:
    step = v_t − v
    v += N(step, P·step)
  return v
```

Figure 15: Pseudocode for the write error function, w, in PCM cells. Here, $N(\mu, \sigma^2)$ is a normally distributed random variable with average μ and variance σ^2. The parameter T controls the termination criterion and P reflects the precision of each write pulse.

step size. These parameters determine the average number of iterations required to write the cell.

Figure 15 shows the pseudocode for writes, which resembles the PCM programming feedback control loop of Pantazi et al. [144]. Section 6.5.2 describes our methodology for calibrating the algorithm's parameters to reflect realistic PCM systems.

Each constituent write pulse in a PCM write can either increase or decrease resistance [144, 145, 147]. Flash memory write pulses, in contrast, are unidirectional, so writes must be more conservative to avoid costly RESET operations in the case of overprogramming [201].

READ ERROR FUNCTION Reading from a storage cell is also imprecise. PCM cells are subject to both *noise*, random variation in the stored value, and *drift*, a gradual unidirectional shift [152]. We reuse the model and parameters of Yeo et al. [231]. Namely, the sensed analog value $r(v)$ is related to the written value v as $r(v) = v + \log_{10} t \cdot N(\mu_r, \sigma_r^2)$ where t is the time, in seconds, elapsed since the cell was written. The parameters μ_r and σ_r are the mean and standard deviation of the error effect respectively.

The same error function, with t equal to the duration of a write step, is used to model errors during the verification step of the write process. We use $t = 250\,\text{ns}$ [85, 153] for this case.

READ QUANTIZATION A read operation must determine the digital value corresponding to the analog value $r(v)$. We assume reads based on a successive approximation analog-to-digital converter (ADC), which has been proposed for PCM systems that can vary their level count [155]. The latency for a successive approximation ADC is linear in the number of bits (i.e., $\log n$).

MODEL SIMPLIFICATIONS While this model is more detailed than some recent work, which has used simple closed-form probability distributions to describe program-and-verify writes [85, 153], it necessarily makes some simplifications over the full complexity of the physics underlying PCM.

(a) Concatenation code (b) Striping code

Figure 16: Two codes for storing 16-bit numbers in four 4-bit cells. Each color indicates a different cell. A single-level error leads to a bit flip in the indicated position. In (a), this is the lowest-order bit in the white cell. In (b), the white cell holds the binary value 0111, which is one level away from 1000.

For simplicity, our model does not incorporate differential writes, a technique that would allow a write to begin without an initial RESET pulse [85]. The write algorithm also does not incorporate the detection of hard failures, which is typically accomplished by timing out after a certain number of iterations [203]. Hard failure detection is orthogonal to the approximate MLC technique.

We measure write performance improvement in terms of the number of iterations per write. While some MLC write techniques use different durations for differently sized pulses [23, 141, 144], we expect the pulses to have approximately the same average time in aggregate. Previous work, for example, has assumed that each step takes 250 nanoseconds [85, 153]. Furthermore, since our evaluation focuses on performance and energy, we do not model any potential lifetime benefits afforded by the technique's reduction in write pulses.

Finally, our model assumes for simplicity that the value range has uniform guard band sizes: in terms of our model, the threshold T is constant among levels. Asymmetric guard bands could exploit the fact that drift is unidirectional. This optimization is orthogonal to the approximate MLC technique, which simply decreases the size of guard bands relative to their nominal size.

6.3.2 *Encoding Values to Minimize Error*

MLC systems typically divide the bits from a single cell among different memory pages [203]. Using this technique, some pages consist of high-order bits from many cells while other pages consist entirely of low-order bits. In approximate MLCs, low-order bits are the least reliable. So this traditional strategy would lead to pages with uniformly poor accuracy. Here, we use a different approach in order to represent all approximate values with acceptable accuracy.

If each cell has n levels, then individual cells can each represent $\log n$ bits. If a program needs to store $\log n$-bit numbers, then the error characteristics of a single cell are advantageous: a single-level error—when the cell stores $l - 1$ or $l + 1$ after attempting to write l—corresponds to a integer error of 1 in the stored value.

But we also need to combine multiple cells to store larger numbers. We consider two approaches. Concatenation (Figure 16a) appends the bits from the constituent cells to form each word. Striping (Figure 16b) interleaves the cells so that the highest-order bits of each cell all map to the highest-order bits of the word.

An ideal code would make errors in high bits rare while allowing more errors in the low bits of a word. With the straightforward concatenation code, however, a single-level error can cause a high-order bit flip: the word's $\log n$-th most significant bit is the *least* significant bit in its cell. The striping code mitigates high-bit errors but does not prevent them. In the example shown in Figure 16b, the white cell stores the value 0111, so a single-level error can change its value to 1000. This error causes a bit flip in the word's most significant bit. (Gray coding, which some current MLC systems use [85], does not address this problem: single-level errors are as likely to cause flips in high-order bits as in low-order bits.) We evaluate both approaches in Section 6.6 and find, as expected, that the striping code mitigates errors most effectively.

6.3.2.1 *Defining an Optimal Code*

While the above bit-striping approach works well and is straightforward to implement, it is not necessarily *optimal:* there may exist other coding schemes that further mitigate error. Better codes have the potential to benefit any approximate-computing technique that uses analog and multi-level substrates: not only storage but also networking and communication channels. Future work should explore strategies for deriving error-optimal codes.

As a first step in this direction, this section formalizes the notion of error-minimizing, multi-level codes. The two simple codes discussed above are points in a large space of possible codes. We also define the average error of a code as a way to quantify the code's error-mitigating power.

Codes represent b-bit numbers (i.e., the first 2^b integers) using digits drawn from an alphabet of n symbols. A codeword $w = \langle v_1, v_2, \ldots, v_{b/\log n} \rangle$ is a vector of numbers $0 \leq v < n$ where n is the number of levels per cell. Assuming n is a power of two whose base divides b, there are $n^{b/\log n} = 2^b$ codewords, so a code is bijection between the first 2^b integers and the 2^b codewords.

Let the distance $d(w, w')$ between two codewords be the l_1 norm, or the city block distance between the two vectors. We assume that the analog medium confuses words with a smaller distance between them more often than more distant words. Specifically, the probability that w is written and w' is subsequently recovered is inversely proportional to $d(w, w')$.

A *code* is a function c where $c(i)$ is the codeword that represents the integer i. (The domain of c is the integers $0 \leq i < 2^b$.) The inverse, $c^{-1}(w)$, decodes a vector to the represented integer.

Let $A(w)$ be the random process that introduces these analog storage or transmission errors into a codeword. The overall average error of a code c is:

$$\mathrm{E}\Big[|i - c^{-1}(A(c(i)))|\Big]$$

An optimal code is one that minimizes this expected value when i is drawn from a uniform distribution (or some other sensible distribution).

An exhaustive search for the optimal code by this definition is intractable: there are $2^b!$ possible codes for b-bit numbers. Recent work [81] has used a constraint formulation to search a subset of codes that reorder bits, but more so-

phisticated schemes may exist that nonetheless have practical circuit-level implementations. Future work should develop search strategies for low-error codes in the enormous space of possible mappings.

6.3.3 *Memory Interface*

MLC blocks can be made precise or approximate by adjusting the target threshold of write operations. For this reason, the memory array must know which threshold value to use for each write operation. Rather than storing the precision level as metadata for each block of memory, we encode that information in the operation itself by extending the memory interface to include precision flags as described in Section 6.2. This approach, aside from eliminating metadata space overhead, eliminates the need for a metadata read on the critical path for writes.

Read operations are identical for approximate and precise memory, so the precision flag in read operations goes unused. A different approximate MLC design could adjust the cell density of approximate memory; in this case, the precision flag would control the bit width of the ADC circuitry [155].

OVERHEADS Since no metadata is used to control cells' precision, this scheme carries no space overhead. However, at least one additional bit is necessary in each read and write request on the memory interface to indicate the operation's precision. If multiple threshold values are provided to support varying precision levels, multiple bits will be needed. Additional circuitry may also be necessary to permit a tunable threshold value during cell writes. Our performance evaluation, in Section 6.5, does not quantify these circuit area overheads.

6.4 USING FAILED MEMORY CELLS

PCM, along with flash memory and other upcoming memory technologies, suffers from cell failures during a device's deployment—it "wears out." Thus, techniques for hiding failures from software are critical to providing a useful lifespan for a memory [102]. These techniques typically abandon portions of memory containing uncorrectable failures and use only failure-free blocks [154, 186, 190]. By employing otherwise-unusable failed blocks to store *approximate* data, it is possible to extend the lifetime of an array as long as sufficient intact capacity remains to store the application's *precise* data.

The key idea is to use blocks with exhausted error-correction resources to store approximate data. Previous work on approximate storage in DRAM [113] and SRAM [59] has examined *soft* errors, which occur randomly in time and space. If approximate data is stored in PCM blocks with failed cells, on the other hand, errors will be *persistent*. That is, a value stored in a particular failed block will consistently exhibit bit errors in the same positions. We can exploit the awareness of failure positions to provide more effective error correction via *bit priorities*.

6.4.1 *Prioritized Bit Correction*

In an error model incorporating stuck-at failures, we can use error correction to concentrate failures where they are likely to do the least harm. For example, when storing a floating-point number, a bit error is least significant when it occurs in the low bits of the mantissa and most detrimental when it occurs in the high bits of the exponent or the sign bit. In a uniform-probability error model, errors in each location are equally likely, while a deterministic-failure model affords the opportunity to protect a value's most important bits.

A correction scheme like error-correcting pointers (ECP) [186] marks failed bits in a block. Each block has limited correction resources; for example, when the technique is provisioned to correct two bits per block (ECP_2), a block becomes unusable for precise storage when three bits fail. For approximate storage, we can use ECP to correct the bits that appear in high-order positions within words and leave the lowest-order failed bits uncorrected. As more failures appear in this block, only the least-harmful stuck bits will remain uncorrected.

6.4.2 *Memory Interface*

A memory module supporting failed-block recycling determines which blocks are approximate and which may be used for precise storage. Unlike with the approximate MLC technique (Section 6.3), software has no control over blocks' precision state. To permit safe allocation of approximate and precise data, the memory must inform software of the locations of approximate (i.e., failed) blocks.

When the memory module is new, all blocks are precise. When the first uncorrectable failure occurs in a block, the memory issues an interrupt and indicates the failed block. This is similar to other systems that use page remapping to retire failed segments of memory [83, 237]. The OS adds the block to a pool of approximate blocks. Memory allocators consult this set of approximate blocks when laying out data in the memory. While approximate data can be stored in any block, precise data must be allocated in memory without failures. Eventually, when too many blocks are approximate, the allocator will not be able to find space for all precise data—at this point, the memory module must be replaced.

To provide traditional error correction for precise data, the memory system must be able to detect hard failures after each write [186]. We reuse this existing error detection support; the precision level of the write operation (see Section 6.2) determines the action taken when a failure is detected. When a failure occurs during a precise write, the module either constructs ECP entries for all failed bits if sufficient entries are available or issues an interrupt otherwise. When a failure occurs during an approximate write, no interrupt is issued. The memory silently corrects as many errors as possible and leaves the remainder uncorrected.

To make bit prioritization work, the memory module needs information from the software indicating which bits are most important. Software specifies this using a value size associated with each approximate write as described in Section 6.2. The value size indicates the homogenous byte-width of approximate val-

ues stored in the block. If a block represents part of an array of double-precision floating point numbers, for example, the appropriate value size is 8 bytes. This indicates to the memory that the bits at index i where $i \equiv 0 \bmod 64$ are most important, followed by $1 \bmod 64$, etc. When a block experiences a new failure and the memory module must choose which errors to correct, it masks the bit indices of each failure to obtain the index modulo 64. It corrects the bits with the lowest indices and leaves the remaining failures uncorrected.

This interface for controlling bit prioritization requires blocks to contain homogeneously sized values. In our experience, this is a common case: many of the applications we examined use approximate `double[]` or `float[]` arrays that span many blocks.

OVERHEADS Like the approximate MLC scheme, failed-block recycling requires additional bits for each read and write operation in the memory interface. Messages must contain a precision flag and, to enable bit priority, a value size field. The memory module must incorporate logic to select the highest-priority bits to correct in an approximate block; however, this selection happens rarely because it need only occur when new failures arise. Finally, to correctly allocate new memory, the OS must maintain a pool of failed blocks and avoid using them for precise storage. This block tracking is analogous to the way that flash translation layers (FTLs) remap bad blocks.

6.5 EVALUATION

Approximate storage trades off precision for performance, durability, and density. To understand this trade-off in the context of real-world approximate data, we simulate both of our techniques and examine their effects on the quality of data sets and application outputs. We use application-specific metrics to quantify quality degradation (see Section 1.2.1).

We first describe the main-memory and persistent-data benchmarks used in our evaluation. We then detail the MLC model parameters that dictate performance and error rates of the approximate MLC technique. Finally, we describe the model for wear-out used in our evaluation of the failed-block recycling technique.

6.5.1 *Applications*

We use two types of benchmarks in our evaluation: main-memory applications and persistent data sets. The main-memory applications are programs that mix some approximate data and some precise control data. The persistent-storage benchmarks are static data sets that can be stored 100% approximately.

For the main-memory applications, we adapt the annotated benchmarks from the evaluation of EnerJ in Chapter 3. An in-house simulator based on the one used in EnerJ's evaluation (Section 3.6) intercepts loads and stores to collect access statistics and inject errors. We examine eight of the EnerJ-annotated bench-

marks: jmeint, raytracer, zxing), and the SciMark2 kernels (fft, lu, mc, smm, and sor). We use the same output-quality metrics as in the EnerJ evaluation.

For persistent storage, we examine four new sets of approximate data. The first, sensorlog, consists of a log of mobile-phone sensor readings from an accelerometer, thermometer, photodetector, and hydrometer. The data is used in a decision tree to infer the device's context, so our quality metric is the accuracy of this prediction relative to a fully-precise data set. The second, image, stores a bitmap photograph as an array of integer RGB values. The quality metric is the mean error of the pixel values. The final two data sets, svm and ann, are trained classifiers for handwritten digit recognition based on a support vector machine and a feed-forward neural network. In both cases, the classifiers were trained using standard algorithms on the "pendigits" data set from the UCI Machine Learning Repository [13]. The data set consists of 3498 training samples and 7494 testing samples, each of which comprises 16 features. Then, the classifier parameters (support vectors and neuron weights, respectively) are stored in approximate memory. The SVM uses 3024 support vectors; the NN is configured with a sigmoid activation function, two hidden layers of 128 neurons each, and a one-hot output layer of 10 neurons. We measure the recognition accuracy of each classifier on an unseen test data set relative to the accuracy of the precise classifier (95% for svm and 80% for ann). Unlike the main-memory applications, which consist of a mixture of approximate and precise data, the persistent data sets are entirely approximate.

6.5.2 *MLC Model Parameters*

To assess our approximate MLC technique, we use the model described in Section 6.3.1. The abstract model has a number of parameters that we need to select for the purposes of simulation. To set the parameters, we use values from the literature on MLC PCM configurations. Since our architecture-level model of iterative program-and-verify writes is original, we infer its parameters by calibrating them to match typical write latencies and error rates.

For a baseline (precise) MLC PCM cell, we need a configuration where errors are improbable but not impossible. We choose a conservative baseline raw bit error rate (RBER) of 10^{-8}, which comports with RBERs observed in flash memory today [26, 127].

We first select parameters for the read model in Section 6.3.1, which incorporates the probabilistic effects of read noise and drift. For the parameters μ_r and σ_r, we use typical values from Yeo et al. [231] normalized to our presumed 0.0–1.0 value range. Specifically, for PCM, we choose $\mu_r = 0.0067$ and $\sigma_r = 0.0027$. Since the read model incorporates drift, it is sensitive to the retention time between writes and reads. Retention time can be short in a main-memory deployment and much longer when PCM is used for persistent storage. As an intermediate value, we consider retention for $t = 10^5$ seconds, or slightly more than one day. Note that this retention time is pessimistic for the main-memory case: in our experiments, every read experiences error as if it occurred 10^5 seconds

after the preceding write. In real software, the interval between writes and subsequent reads is typically much lower.

We model a 4-level (2-bit) PCM cell. To calibrate the write model, we start from an average write time of 3 cycles as suggested by Nirschl et al. [141] and a target RBER of 10^{-8}. We need values for the parameters T and P that match these characteristics. We choose our baseline threshold to be 20% of the largest threshold that leads to non-overlapping values (i.e., $T = 0.025$); this leads to about 3 iterations per write. Setting $P = 0.035$ leads to an error probability on the order of 10^{-8} for a retention time of 10^5 seconds.

6.5.3 *Wear-Out Model*

To evaluate the effect of using blocks with failed cells for approximate storage, we simulate single-level PCM. In single-level PCM, bits become stuck independently as their underlying cells fail. With multi-level designs, in contrast, a single cell failure can cause multiple bits to become stuck, so bit failures are not independent. Assuming that the memory assigns bits from a given cell to distinct pages [203] and that wear leveling randomly remaps pages, failures nonetheless *appear* independent in multi-level PCM. So a multi-level failure model would closely resemble our single-level model with an accelerated failure rate.

We evaluate PCM with 2-bit error-correcting pointers (ECP) [186]. While precise configurations of the ECP technique typically use 6-bit correction, approximate storage can extend device lifetime without incurring as much overhead as a fully precise configuration. Approximate blocks also use the bit priority assignment mechanism from Section 6.4.1: where possible, ECP corrections are allocated to higher-order bits within each value in the block.

To understand the occurrence of stuck bits in failed blocks, we need a realistic model for the rate at which cells wear out over time. To this end, we simulate a PCM array for trillions of writes and measure the distribution of cell failures among blocks. The statistical simulator is adapted from Azevedo et al. [12] and assumes an average PCM cell lifetime of 10^8 writes (although the *first* failure occurs much earlier). We use separate workloads to simulate wear in a main-memory setting and in a persistent-storage setting.

MAIN-MEMORY WEAR To model wear in main-memory PCM deployments, we simulate the above suite of main-memory applications and gather statistics about their memory access patterns, including the relative size of each program's approximate vs. precise data and the frequency of writes to each type of memory. We then take the harmonic mean of these statistics to create an aggregate workload consisting of the entire suite. We run a statistical PCM simulation based on these application characteristics, during which all blocks start out precise. When a block experiences its first uncorrectable cell failure, it is moved to the approximate pool. Failed blocks continue to be written and experience additional bit failures because they store approximate data. Periodically, we record the amount of memory that remains precise along with the distribution of failures among the

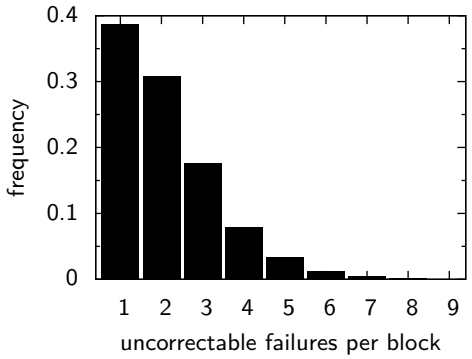


Figure 17: Distribution of uncorrectable cell failures using ECP_2 among 512-bit blocks after the entire memory has been overwritten 3.2×10^7 times under the main-memory wear model. (At this stage, half of the blocks have at least one uncorrectable failure.)

approximate blocks. We simulate each application under these measured failure conditions.

As an example, Figure 17 depicts the error rate distribution for the wear stage at which 50% of the memory's blocks have at least one failure that is uncorrectable using ECP_2—i.e., half the blocks are approximate. In this stage, most of the blocks have only a few uncorrectable failures: 39% of the approximate blocks have exactly one such failure and only 1.7% have six or more.

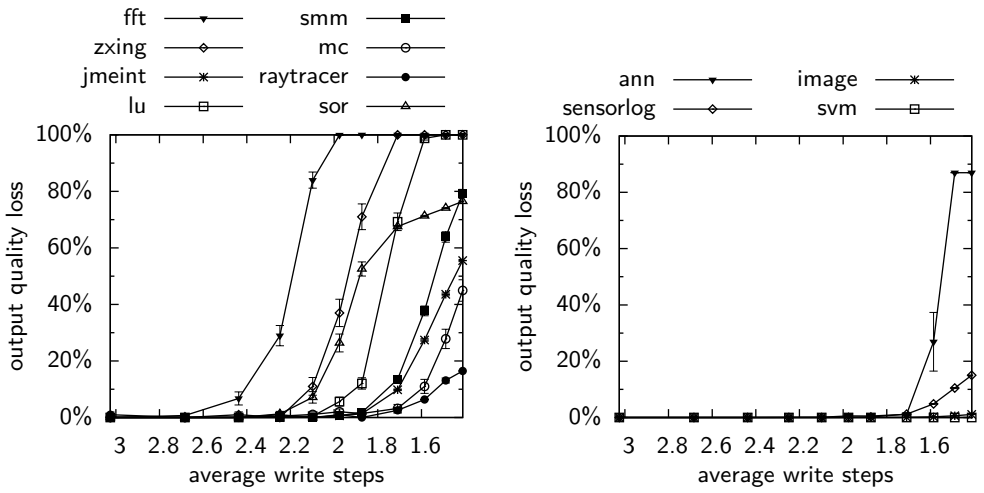
PERSISTENT-STORAGE WEAR For our persistent-storage data sets, all data is approximate. So we simulate writes uniformly across all of memory, both failed and fully-precise. This corresponds to a usage scenario in which the PCM array is entirely dedicated to persistent storage—no hybrid transient/persistent storage is assumed. As with the main-memory wear model, we periodically snapshot the distribution of errors among all blocks and use these to inject bit errors into stored data.

6.6 RESULTS

We evaluate both sets of benchmarks under each of our two approximate storage techniques. We first measure the approximate MLC mechanism.

6.6.1 *Approximate MLC Memory*

In our approximate MLC experiments, we map all approximate data to simulated arrays of two-bit PCM cells. We run each benchmark multiple times with differing threshold (T) parameters. We use T values between 20% and 90% of the maximum threshold (i.e., the threshold that eliminates guard bands altogether). For each threshold, we measure the average number of iterations required to write a random value. This yields an application-independent metric that is directly proportional to write latency (i.e., inversely proportional to performance).



(a) Main memory with approximate MLC.

(b) Persistent data with approximate MLC.

Figure 18: Output degradation for each benchmark using the approximate MLC technique. The horizontal axis shows the average number of iterations per write. The vertical axis is the output quality loss as defined by each application's quality metric. Quality loss is averaged over 100 executions in (a) and 10 in (b); the error bars show the standard error of the mean.

Configurations with fewer iterations per write are faster but cause more errors. So, for each application, the optimal configuration is the one that decreases write iterations the most while sacrificing as little output quality as possible. Faster writes help close PCM's performance gap with DRAM in the main-memory case and improve write bandwidth in the persistent-data case [102, 112].

APPROXIMATE MAIN MEMORY Figure 18a relates write performance to application output quality loss. For configurations with fewer write iterations—to the right-hand side of the plot—performance improves and quality declines. The leftmost point in the plot is the nominal configuration, in which writes take 3.03 iterations on average and errors are rare. Reducing the number of iterations has a direct impact on performance: a 50% reduction in iterations leads to 2× improvement in write speed.

The error for each application stays low for several configurations and then increases sharply when hardware errors become too frequent. The raytracer benchmark exhibits quality loss below 2% up to the configuration with 1.71 iterations per write on average, a 1.77× speedup over the baseline. Even the least tolerant application, fft, sees only 4% quality loss when using an average of 2.44 iterations per write (or 1.24× faster than the baseline). This variance in tolerance suggests that different applications have different optimal MLC configurations. Approximate memories can accommodate these differences by exposing the threshold parameter T for tuning.

To put these speedups in the context of the whole application, we show the fraction of dynamic writes that are to approximate data in Figure 19. Most

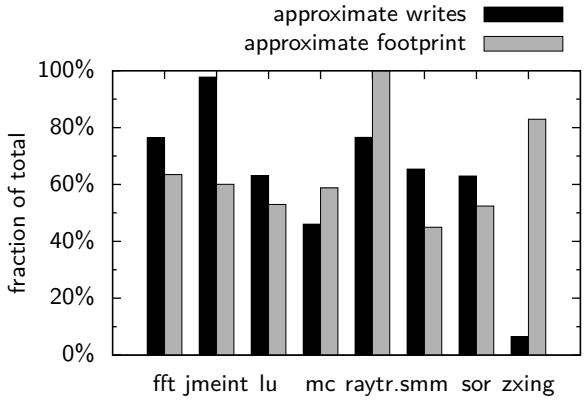


Figure 19: Proportions of approximate writes and approximate data in each main-memory benchmark.

applications use approximate writes for more than half of their stores; jmeint in particular has 98% approximate writes. One application, zxing, has a large amount of "cold" approximate data and benefits less from accelerating approximate writes.

PERSISTENT STORAGE Figure 18b shows the quality degradation for each persistent data set when running on approximate MLC memory. The persistent data sets we examine are more tolerant than the main-memory benchmarks. The sensor logging application, for instance, exhibits only 5% quality degradation in the configuration with 1.59 iterations per write (1.91× faster than the baseline) while the bitmap image has only 1% quality degradation even in the most aggressive configuration we examined, in which writes take 1.41 cycles (2.14× faster than the baseline). The neural network classifier, ann, experiences less than 10% recognition accuracy loss when using 1.77× faster writes; svm, in contrast, saw negligible accuracy loss in every configuration we measured.

Overall, in the configurations with less than 10% quality loss, the benchmarks see 1.7× faster writes to approximate cells over precise cells on average.

This write latency reduction benefits application performance and memory system power efficiency. Since write latency improvements reduce contention and therefore also impact read latency, prior evaluations have found that they can lead to large IPC increases [74, 85]. Since fewer programming pulses are used per write and write pulses make up a large portion of PCM energy, the overall energy efficiency of the memory array is improved.

IMPACT OF ENCODING Section 6.3.2 examines two different strategies for encoding numeric values for storage on approximate MLCs. In the first, the bits from multiple cells are concatenated to form whole words; in the second, each value is "striped" across constituent cells so that the highest bits of the value map to the highest bits of the cells. The results given above use the latter encoding, but we also evaluated the simpler code for comparison.

The striped code leads to better output quality on average. For three intermediate write speeds, using that code reduces the mean output error across all applications from 1.1% to 0.4%, from 3.6% to 3.0%, and from 11.0% to 9.0% with respect to the naive code.

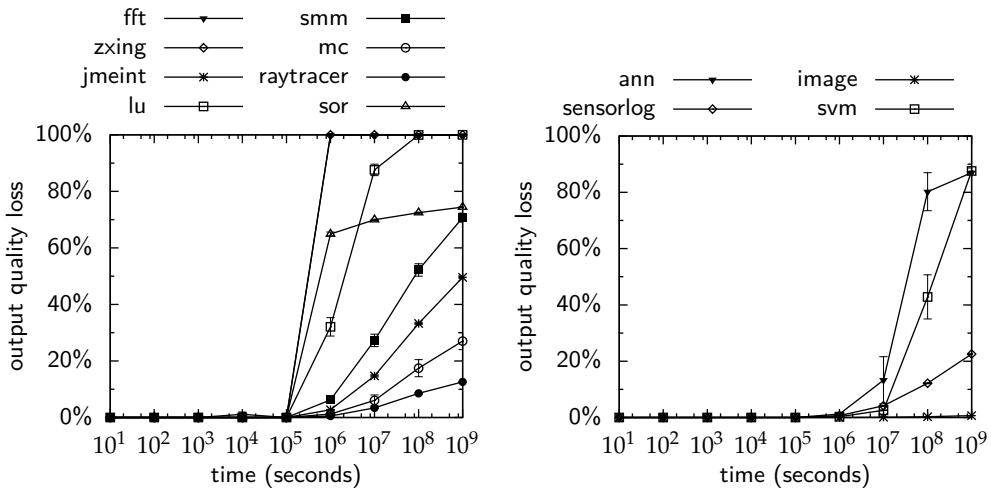
We also performed two-sample t-tests to assess the difference in output quality between the two coding strategies for each of 13 write speed configurations. For nearly every application, the striped code had a statistically significant positive effect on quality more often than a negative one. The only exception is mc, a Monte Carlo simulation, in which the effect of the striped code was inconsistent (positive at some write speeds and negative for others).

While the striped code is imperfect, as discussed in Section 6.3.2, it fares better than the naive code in practice since it lowers the probability of errors in the high-order bits of words.

DENSITY INCREASE We experimented with adding more levels to an approximate MLC. In a precise MLC, increasing cell density requires more precise writes, but approximate MLCs can keep average write time constant. Our experiments show acceptable error rates when six levels are used (and no other parameters are changed). A non-power-of-two MLC requires additional hardware, similar to binary-coded decimal (BCD) circuitry, to implement even the naive code from Section 6.3.2 but can still yield density benefits. For example, a 512-bit block can be stored in $\lceil \frac{512}{\log 6} \rceil = 199$ six-level cells (compared to 256 four-level cells). With the same average number of write iterations (3.03), many of our benchmarks see little error: jmeint, mc, raytracer, smm, and the four persistent-storage benchmarks see error rates between 0.1% and 4.2%. The other benchmarks, fft, lu, sor, and zxing, see high error rates, suggesting that density increase should only be used with certain applications.

IMPACT OF DRIFT Previous work has suggested that straightforward MLC storage in PCM can be untenable over long periods of time [231]. Approximate storage provides an opportunity to reduce the frequency of scrubbing necessary by tolerating occasional retention errors. To study the resilience of approximate MLC storage to drift, we varied the modeled retention time (the interval between write and read) and examined the resulting application-level quality loss. Recall that the results above assume a retention time of 10^5 seconds, or about one day, for every read operation; we examined retention times between 10^1 and 10^9 seconds (about 80 years) for an intermediate approximate MLC configuration using an average of 2.1 cycles per write.

Figure 20 depicts the application output quality for a range of time intervals. For the main-memory applications in Figure 20a, in which typical retention times are likely far less than one day, we see little quality loss (1% or less) for retention times of 10^4 seconds or shorter. As above, these simulations assume the same drift interval for *every* read. In this sense, the results are pessimistic since many reads are to recently written data and therefore incur less error from drift.



(a) Drift for main-memory applications.

(b) Drift impact for persistent data sets.

Figure 20: Application output quality over time using the approximate MLC technique using 2.1 cycles per write. Drift causes errors to increase in proportion to the time since the last write to the PCM cell.

For the persistent-storage benchmarks in Figure 20b, in contrast, longer retention times are the norm. In that setting, quality loss remains under 10% for at least 10^6 seconds and, for all benchmarks except ann, through 10^7 seconds. The most tolerant data set, image, remains below 10% error for 10^9 seconds of drift. The persistent-storage benchmarks tend to be more resilient to drift because the stored data tends to be uniformly error tolerant: every neuron weight or every pixel contributes equally to the quality of the output. This uniformity contrasts with the main-memory applications, where certain "hot" data structures are more critical for quality and therefore tolerate less error.

A longer retention time means scrubbing can be done less frequently. The above results report the quality impact of one retention cycle: the persistent-storage benchmarks, for example, lose less than 10% of their quality when 10^6 seconds, or about 11 days, elapse after they are first written to memory assuming no scrubbing occurs in that time. Eleven more days of drift will compound additional error. While the results suggest that the more error-tolerant applications can tolerate longer scrubbing cycles, we do not measure how error compounds over longer-term storage periods with infrequent scrubbing.

BIT ERROR RATE To add context to the output quality results above, we also measured the effective bit error rate (BER) of approximate MLC storage. The BER is the probability that a bit read from approximate memory is different from the corresponding last bit written. Across the write speeds we examined, error rates range from 3.7×10^{-7} to 8.4% in the most aggressive configuration. To put these rates in perspective, if the bit error rate is p, then a 64-bit block will have at least 2 errors with probability $\sum_{i=2}^{64} B(i, 64, p)$ where B is the binomial distribution. At a moderately aggressive write speed configuration with an average of 1.9 steps, approximate MLC storage has an error rate of 7.2×10^{-4}, so

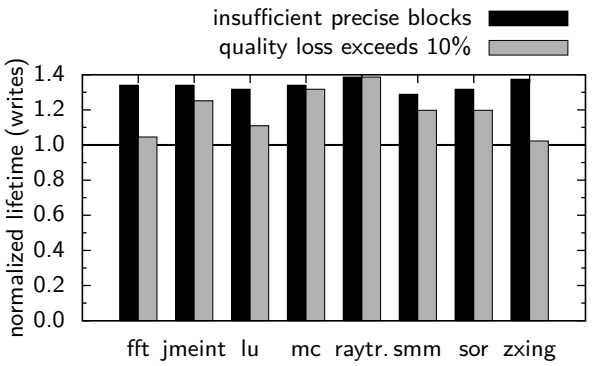


Figure 21: Lifetime extension for each application. Each bar represents the number of writes to the entire array at which the application can no longer run, normalized to the point of array failure in fully-precise mode. The black bar indicates when there is not enough precise memory available. The gray bar shows when the application's output quality degrades more than 10%.

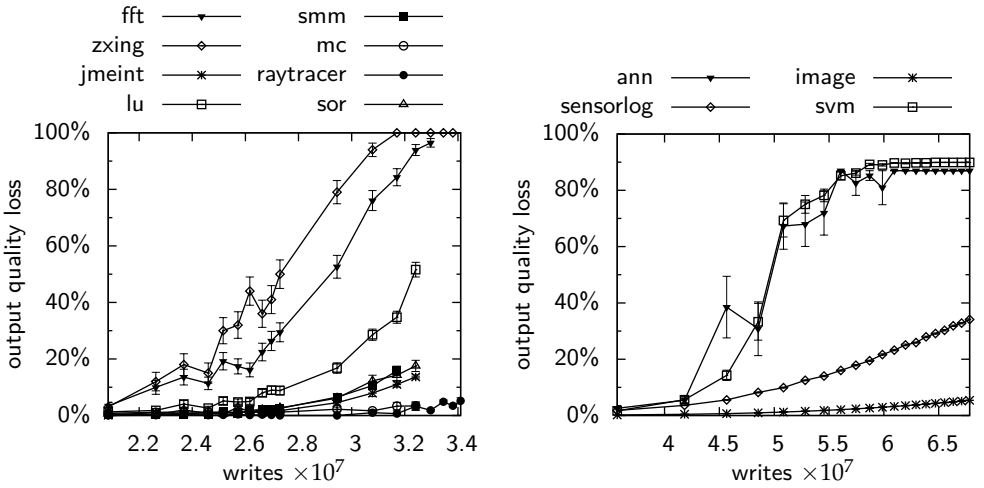
0.1% of 64-bit words have 2 or more errors. This high error rate demonstrates the need for application-level error tolerance: even strong ECC with two-bit correction will not suffice to provide precise storage under such frequent errors.

6.6.2 *Using Failed Blocks*

We evaluate the failed-block recycling technique by simulating benchmarks on PCM arrays in varying stages of wear-out. As the memory device ages and cells fail, some blocks exhaust their error-correction budget. Approximate data is then mapped onto these blocks. Over the array's lifetime, bit errors in approximate memory become more common. Eventually, these errors impact the application to such a degree that the computation quality is no longer acceptable, at which point the memory array must be replaced. We quantify the lifetime extension afforded by this technique, beginning with the main-memory applications.

To quantify lifetime extension, we assume a memory module with a 10% "space margin": 10% of the memory is reserved to allow for some block failures before the array must be replaced. In the baseline precise configuration, the array fails when the fraction of blocks that remain precise (having only correctable failures) drops below 90%. In the approximate configuration, programs continue to run until there is not enough space for their precise data or quality drops below a threshold.

APPROXIMATE MAIN MEMORY Figure 21 depicts the lifetime extension afforded by using failed blocks as approximate storage. For each application, we determine the point in the memory's lifetime (under the wear model described in Section 6.5.3) at which the program can no longer run. We consider two termination conditions: when the amount of precise memory becomes insufficient (i.e., the proportion of approximate memory exceeds the application's propor-



(a) Main memory using failed blocks.

(b) Persistent data sets using failed blocks.

Figure 22: Output quality degradation for each benchmark when using the failed-block recycling technique. The horizontal axis is the number of complete overwrites the array has experienced, indicating the stage of wear-out. The vertical axis is an application-specific error metric.

tion of approximate data) and when the application's output quality degrades more than 10%. Each bar in the figure shows the normalized number of writes to the memory when application failure occurs.

With quality degradation limited to 10%, the benchmarks see lifetime extensions ranging from 2% (zxing) to 39% (raytracer) with a harmonic mean of 18%. With quality unconstrained, the mean lifetime extension is 34%, reflecting the fact that this technique leads to gradually decreasing quality as the memory array ages.

To help explain these results, Figure 22a shows the quality degradation for each application at various points during the memory array's wear-out. The most error-tolerant application, raytracer, sees little quality degradation under all measured wear stages. Some applications are limited by the amount of approximate data they use. Figure 19 shows the proportion of bytes in each application's memory that is approximate (averaged over the execution). Some applications, such as mc, are tolerant to error but only have around 50% approximate data. In other cases, such as zxing and fft, bit errors have a large effect on the computation quality. In fft in particular, we find that a single floating-point intermediate value that becomes `NaN` can contaminate the Fourier transform's entire output. This suggests that the application's precision annotations, which determine which data is stored approximately, may be too aggressive.

PERSISTENT STORAGE Figure 22b shows the quality degradation for each data set at different points during the lifetime of the memory. The memory's intermediate wear-out conditions come from the persistent-storage wear model described in Section 6.5.3. In a fully-precise configuration, the memory fails (exceeds 10% failed blocks) at about 3.4×10^7 overwrites, or at the left-hand

side of the plot. Recall that, in these persistent-storage benchmarks, the data is stored 100% approximately; no precise storage is used.

As with the main-memory storage setting, quality decreases over time as errors become more frequent. But these benchmarks are more tolerant to stuck bits than the main-memory applications. For image, quality loss is below 10% in all wear stages; for sensorlog, it remains below 10% until the array experiences 5.0×10^7 writes, or 42% later than precise array failure. The two machine learning classifiers, ann and svm, each see lifetime extensions of 17%. This tolerance to stuck bits makes the failed-block recycling technique particularly attractive for persistent storage scenarios with large amounts of numeric data.

Overall, across both categories of benchmarks, we see a harmonic mean lifetime extension of 23% (18% for the main-memory benchmarks and 36% for the persistent-storage data sets) when quality loss is limited to 10%. Recent work has demonstrated PCM arrays with a random write bandwidth of 1.5 GB/s [32]; for a 10 GB memory constantly written at this rate, these savings translate to extending the array's lifetime from 5.2 years to 6.5 years.

IMPACT OF BIT PRIORITY The above results use our type-aware prioritized correction mechanism (Section 6.4.1). To evaluate the impact of bit prioritization, we ran a separate set of experiments with this mechanism disabled to model a system that just corrects the errors that occur earliest. We examine the difference in output quality at each wear stage and perform a two-sample t-test to determine whether the difference is statistically significant ($P < 0.01$).

Bit prioritization had a statistically significant positive impact on output quality for all benchmarks except mc. In sensorlog, for example, bit prioritization decreases quality loss from 2.3% to 1.7% in an early stage of wear (the leftmost point in Figure 22b). In fft, the impact is larger: bit prioritization reduces 7.3% quality loss to 3.3% quality loss. As with encoding for approximate MLCs, the exception is mc, whose quality was (statistically significantly) improved in only 4 of the 45 wear stages we measured while it was *negatively* impacted in 6 wear stages. This benchmark is a simple Monte Carlo method and hence may sometimes *benefit* from the entropy added by failed bits. Overall, however, we conclude that bit prioritization has a generally positive effect on storage quality.

IMPACT OF ECP BUDGET The above experiments use a PCM configuration with error-correcting pointers (ECP) [186] configured to correct two stuck bits per 512-bit block at an overhead of 21 extra bits per block. More aggressive error correction improves the endurance of both fully-precise and approximate memory and amplifies the opportunity for priority-aware correction in intermediate wear stages. To quantify the effect of increasing error correction budgets, we also evaluated an ECP_6 configuration (61 extra bits per block).

Moving from ECP_2 to ECP_6 extends the lifetime of a precise memory array by 45% under main-memory wear or 17% under persistent-storage wear. Our results for approximate main-memory storage with ECP_2 provide a portion of these benefits (18% lifetime extension) without incurring any additional correc-

tion overhead. In the persistent-storage case, the lifetime extension for approximate storage (36%) is greater than for increasing the ECP budget.

6.7 DISCUSSION

Approximate storage exposes new efficiency–accuracy trade-offs in a system component that other work on system-level approximation steers around: main memory and persistent storage. As DRAM scaling begins to falter, PCM and other resistive memories will become crucial to satisfying increasing memory needs. The two techniques in this chapter offer one way to work around these new technologies' novel quirks: wear-out and slow writes, especially in multi-level cell configurations.

This work also poses one important unsolved problem (see Section 6.3.2): how should we encode approximate data for approximate channels? The vast body of work on coding theory for error correction tends to assume that we need to recover the data exactly—or, more generally, that every bit in a message is equally important. Encoding data to minimize the *numerical* error in the decoded values remains an important, and unexplored, counterpoint to traditional error correction.

7

AN OPEN-SOURCE APPROXIMATION INFRASTRUCTURE

7.1 INTRODUCTION

Approximate computing includes a diverse spectrum of implementation techniques, spanning both hardware and software: everything from adjusting numerical representations to exploiting analog circuits. Some work relies on programmers for manual reasoning to control approximation's potential effects [60, 113, 164, 194], while other work proposes automated transformation based on code patterns or exhaustive search [14, 176, 177]. 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.

This chapter describes 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 [101], extends the syntax of C and C++ to incorporate an `APPROX` keyword that programmers use to annotate types, as in Chapter 3. ACCEPT's central analysis, *approximatability*, identifies coarse-grained regions of code that can affect only approximate values. Coarse region 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.

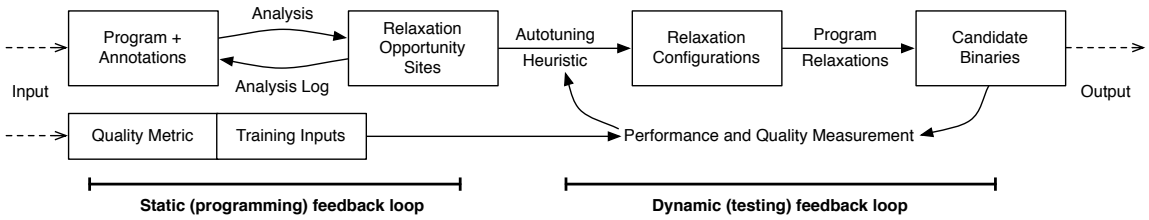


Figure 23: Overview of the ACCEPT compiler workflow.

ACCEPT is an end-to-end framework that makes existing 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×, 4.8×, and 1.5× 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. ACCEPT's combination of static analysis and dynamic measurement alleviates much of the manual labor from the process of applying approximation without sacrificing transparency or control.

The ACCEPT framework is open source and ready for use as research infrastructure. It provides the necessary language and compiler support to prototype and evaluate new strategies for approximation, reducing the need to reinvent these components for each new research evaluation.

7.2 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 23 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 work 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 outsized 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 EnerJ's lightweight annotation system (see Chapter 3). 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 [60, 131, 134, 164, 165, 194, 197].

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.

7.3 ANNOTATION AND PROGRAMMER FEEDBACK

This section describes ACCEPT's annotations and feedback, which help programmers balance safety with approximation. Rather than proving theoretical accuracy guarantees for restricted programming models as in other work [132, 182, 239], ACCEPT's workflow extends mainstream development practices: it com-

bines lightweight safety guarantees, programmer insight, and testing to apply approximation to general code.

7.3.1 *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 from Chapter 3. We originally designed EnerJ to bound the effects of unreliable hardware components that introduce errors at a fine grain; here, we extend the idea to coarse-grained compiler transformations. This way, ACCEPT follows the best-of-both-worlds principle in Section 1.2.5: it combines a fine-grained programming model with more efficient, coarse-grained approximation techniques.

INFORMATION FLOW AND ENDORSEMENT ACCEPT's information-flow type system is directly derived from EnerJ's. The noninterference property from Chapter 3 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.

The annotations consist of an `APPROX` keyword, a type qualifier marking approximate values, and an `ENDORSE` keyword, which casts from an approximate type to its precise equivalent. See Section 3.2 for background on these two constructs.

POINTER TYPES As outlined in Section 3.2.5, covariant reference types can lead to unsoundness. As with object types in EnerJ, therefore, pointer and C++ reference types in ACCEPT are 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 7.4 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. First, 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, the `ACCEPT_FORBID` annotation is useful for marking parts of the program involved in introspection. Section 7.7.4 gives more detail on these experiences.

7.3.2 *Programmer Feedback*

ACCEPT takes inspiration from parallelizing compilers that use a development feedback loop to help guide the programmer toward parallelization opportunities [77, 168]. 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.

7.4 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 an analysis called *approximatability*. This section describes ACCEPT's implementations of several program relaxations drawn from the literature and how approximatability analysis makes them safe. As a framework for approximation, ACCEPT is extensible to relaxations beyond those we describe here.

7.4.1 *Approximatability Analysis*

ACCEPT provides a core program analysis that client optimizations use to ensure safety. 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 whether it is safe to approximate a region of code. Specifically, the approximatability analysis checks, for a region of interest (e.g., a loop body), whether its side effects are exclusively approximate or may include precise data—in other words, whether it is *pure with respect to precise data*. Approximatability is the key criterion for whether a relaxation can apply. In ACCEPT, every relaxation strategy consults the approximatability analysis and optimizes only approximatable code. A region is approximatable if it:

- contains no stores to precise variables that may be read outside the region;
- does not call any functions that are not approximatable; 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 approximatable and asserts otherwise only if it can prove approximatability. Functions whose definitions are not available are conservatively considered not approximatable. This includes standard-library functions, such as `printf`, where input and output make code unsafe to approximate.

For example, this code:

Algorithm 1: Candidate region selection.

Input: function f
Output: set of approximatable 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 ← formRegionBetween(B′, B)
5              if region is approximatable then
6                  R ← R ∪ {region}
7              end
8          end
9      end
10 end
```

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

is approximatable 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 approximatability restriction ensures that code transformations influence only approximate data. Since only the approximate value `a` escapes the approximatable 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. Optimizations that affect only approximatable code uphold ACCEPT's contract with the programmer: that approximation must affect only variables explicitly marked as approximate.

We implement the core approximatability analysis conservatively using SSA definition–use chains and a simple pointer-escape analysis. Section 7.6 gives more implementation details.

7.4.2 *Target Region Selection*

Accelerator-style program transformations work best when they target larger regions of code. To help optimizations identify profitable targets, ACCEPT can enumerate a function's replaceable approximate code regions. A *candidate region* is a set of instructions that is approximatable, 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 7.4.3.3, can enumerate the candidate regions in a program to attempt optimization. Approximatability 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, a variable-accuracy superoptimizer such as the

floating-point extension to STOKE [187] 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 approximatability checks in the worst case—but typically fewer because the LLVM compiler infrastructure pre-computes the dominator and post-dominator trees.

7.4.3 *Safe Approximate Relaxations*

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

7.4.3.1 *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 [194]. 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 approximatable 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.

7.4.3.2 *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 [131, 134, 164, 165]. 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 approximatable. To elide `pthread_barrier_wait()` synchronization, ACCEPT looks for pairs of calls whose intervening code is approximatable, in such cases removing the *first* call (the second call remains to delimit the end of the region).

7.4.3.3 *Neural Acceleration*

Recent work has shown how to accelerate approximate programs with hardware neural networks [16, 40, 204]. *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) [60, 137, 197]. 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) [137]. ACCEPT uses approximate region selection (Section 7.4.2) 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.

7.4.3.4 *Other Client Relaxations*

The three optimizations above demonstrate ACCEPT's breadth as a framework for realizing ideas from approximate-computing research. We have also used ACCEPT to prototype two other optimizations, not described here: 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 [131], float-to-fixed conversion [1], bit-width reduction [173, 210], GPU pattern replacement [176], and alternate-algorithm selection [7, 14].

7.5 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 [14, 27, 59, 60, 133, 180]. 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 [51] 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.

7.6 IMPLEMENTATION

ACCEPT extends the LLVM compiler infrastructure [101] and has three main components: (1) a modified compiler frontend based on Clang [46] 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.

7.6.1 *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 [66] and Java's JSR-308 with its accompanying Checker Framework [56, 148]. 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.

APPROXIMATION-AWARE TYPE SYSTEM The primary language constructs in ACCEPT's EnerJ-inspired 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.

7.6.2 *Analysis and Relaxations*

Approximatability (Section 7.4.1) and region selection (Section 7.4.2) are implemented as LLVM analysis passes. The ACCEPT prototype includes three relaxations, also LLVM passes, that consume the analysis results. The approximatability 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 approximatable. The region-selection analysis offers methods to enumerate approximatable 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 approximatable because it behaves as a store to `p`.

The loop-perforation and synchronization-elision relaxations (Section 7.4) use approximatability 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.

7.6.3 *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.

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

7.6.4 *Neural Acceleration*

We evaluate ACCEPT's approximate region selection using a Neural Processing Unit (NPU) accelerator implemented on an on-chip FPGA (Section 7.4.3.3). The design is based on recent work that implements an NPU based on systolic arrays [60, 137].

7.7 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.

7.7.1 *Applications*

Table 6 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 [40, 60, 133, 180, 204]. Five programs—canneal, fluidanimate, streamcluster, x264, and zynq-blackscholes—are from the PARSEC parallel benchmark suite [19]. 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 [205].

We selected programs for three deployment platforms—a server, a mobile SoC, and a microcontroller—which we describe in detail below. In one case, sobel,

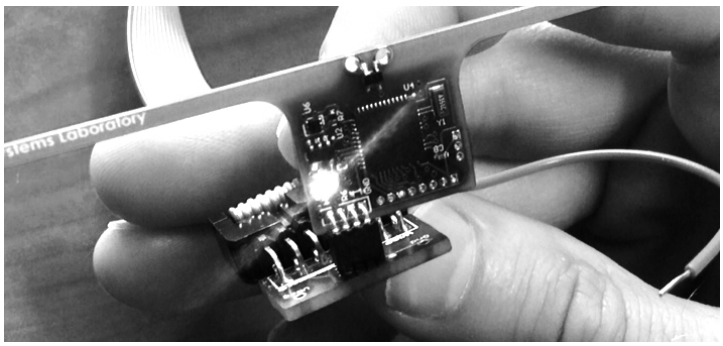


Figure 24: WISP sensing platform [178].

we examine two versions: a conventional implementation for the server and a version ported to the bare-metal (OS-free) environment of the SoC.

To evaluate the applications' output accuracy, we develop application-specific quality metrics as in prior work on approximate computing [14, 59, 60, 133, 180]. Table 6 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 6 shows the source code annotation density. Section 7.7.4 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.

7.7.2 *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 (Section 7.6.4) 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 [137]. Finally, for the embedded msp430-activity workload, we used the WISP [178] device depicted in Figure 24. 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.

Only the Zynq platform supports ACCEPT's neural acceleration optimization. The server and microcontroller benchmarks used the other two optimizations,

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 7: Tuning statistics and resulting optimal configurations for each benchmark.

loop perforation and synchronization elision, while the Zynq experiments explored all three.

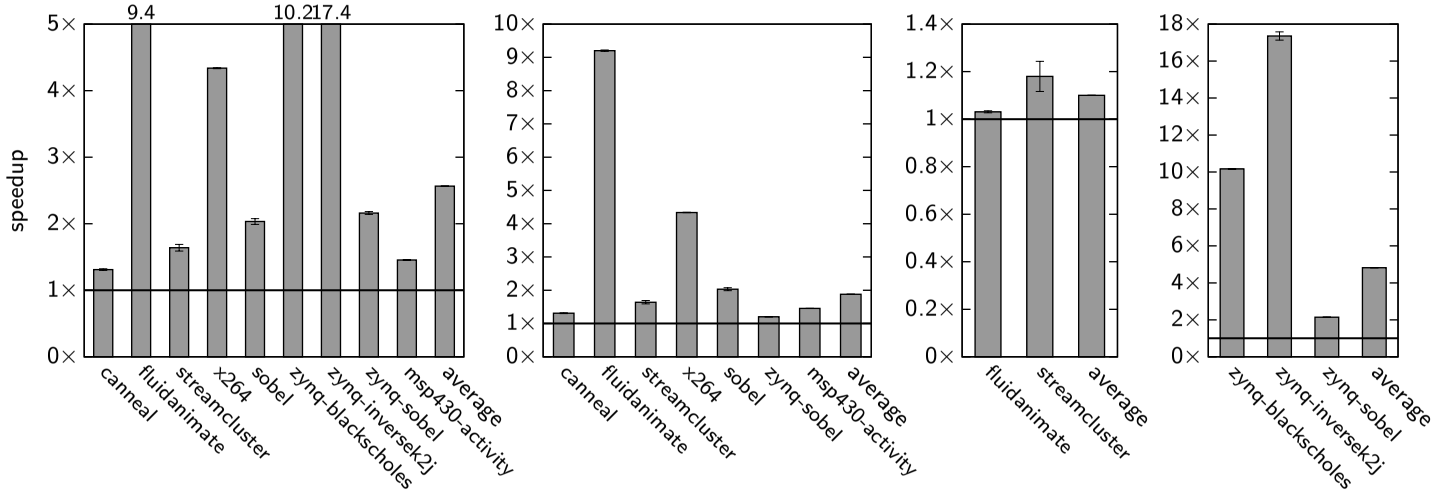
We compiled all applications with LLVM's standard `-O2` optimizations in addition to ACCEPT's program relaxations. We measured performance by reading the system clock before and after a region of interest that excluded the loading of data files from disk and dumping of results. (This region of interest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

7.7.3 *Results*

Figure 25a 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 25 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. Figure 26 shows ACCEPT's Pareto frontier for each benchmark where the frontier contains at least two points. (Configurations are considered "optimal" when no other configuration has both better



(a) Main results (all optimizations) (b) Loop perforation (c) Sync. elision (d) Neural acceleration

Figure 25: Speedup for each application, including all optimizations (a) and each optimization in isolation (b–d).

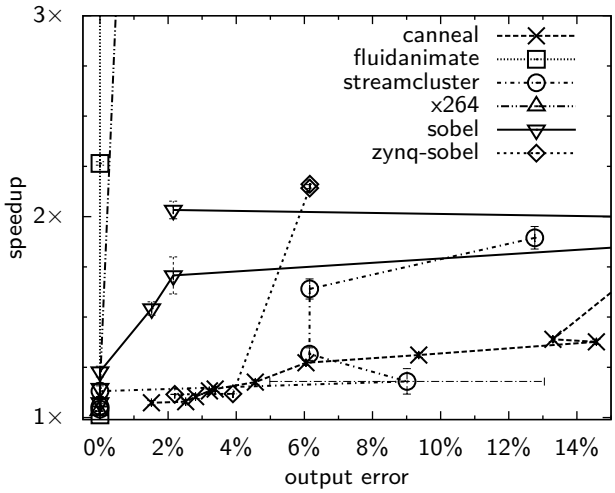


Figure 26: Pareto frontiers for benchmarks with at least two optima.

speedup and better output quality *up to the standard error of the mean*, so some configurations that appear suboptimal are included in the figure due to uncertainty.) Table 7 shows the range of output error rates and speedups in the frontiers.

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 7 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 use on the Zynq system, 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 where 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).

7.7.4 *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.

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 7.3.2), 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 approximatability 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. Both components help alleviate the "manual labor" otherwise necessary to reason about hidden program effects or repeatedly invoke and analyze measurement runs.

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.

SYSTEMS PROGRAMMING The escape hatches from ACCEPT's safety analysis were useful for abstracting 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.

7.8 DISCUSSION

ACCEPT differs from the other projects in this dissertation in its focus on a robust, open-source, end-to-end implementation. The goal is to demonstrate that a common compiler infrastructure can address the common concerns for a wide variety of realistic approximation techniques—in the same way that a classical compiler infrastructure like LLVM provides all the tools that an intrepid compiler hacker needs to build new optimizations. This level of generality required that we solve two common challenges: balancing programmer insight with automation, and bridging the gap between fine-grained annotations and coarse-grained optimizations.

The ACCEPT source code and documentation is available online at:
`http://sampa.cs.washington.edu/accept`

Part IV

CLOSING

8

RETROSPECTIVE

Approximate computing research is still in its early stages. This dissertation re-examines traditional abstractions in hardware and software and argues that they should include a notion of computational quality. It develops five principles for the design of approximation-aware abstractions:

APPLICATION-SPECIFIC RESULT QUALITY In many domains, applications come with correctness constraints that are not binary: there are better outputs and worse outputs. But as with traditional correctness criteria, there is no single, universal "soft" quality criterion. A key principle in this work is that programmers should express *quality metrics* to quantify an output's usefulness on a continuous scale. Quality metrics are essential not only to the design of tools that constrain correctness, but also to the empirical evaluation of any approximation technique.

SAFETY VS. QUALITY The abstractions in this dissertation benefit from decomposing correctness into two complementary concerns: *quality*, the degree of accuracy for approximate values, and *safety*, whether to allow any degree of approximation at all. While this zero-versus-nonzero distinction may at first seem artificial, it decomposes many intractable problems into two smaller problems that can be tractably solved using different tools. EnerJ (Chapter 3) and DECAF (Chapter 4) demonstrate this separation of concerns: information flow types are best suited for safety, and constraint-solving numerical type inference is best suited for quality. Using a single technique for both would be less effective.

HARDWARE–SOFTWARE CO-DESIGN Approximation is a cross-cutting concern. While both hardware and software techniques hold promise, a good rule of thumb is to *never do hardware without software*. Hardware techniques that work opaquely—without incorporating any information at all from the application—are easy to design but doomed to failure. An approximate memory (Chapter 6) that can flip any bit with any probability, for example, ignores the software's complex needs for different levels of reliability for different kinds of data. Researchers should always design hardware techniques with the programming abstraction in mind.

PROGRAMMING WITH PROBABILISTIC REASONING Many of the best proposals for approximate-computing techniques are inherently probabilistic: an

analog circuit [197] or a noisy memory write (Chapter 6), for example, are nondeterministic by nature. Even when approximation strategies themselves are deterministic, correctness criteria can often be best expressed using probabilities: the chance that a randomly selected input has high quality, or the chance that an individual pixel in an image is wrong. In both cases, approximation calls for programming languages to add constructs reflecting probability and statistics. Chapter 4 develops a type-system approach to probabilistic reasoning, and Chapter 5 explores a new way for programmers to express general probabilistic bounds.

GRANULARITY OF APPROXIMATION Approximation techniques work by replacing some accurate part of a program with a cheaper, less accurate counterpart. A critical dimension in these techniques is the *granularity* of components they replace. Approaches that replace individual arithmetic operations [59] can be general and flexible, but their efficiency gains tend to be small. Coarse-grained replacement techniques, such as neural acceleration [60], can be more complex to apply but tend to offer larger gains. The ACCEPT compiler framework in Chapter 7 represents a step toward unifying an intelligible fine-grained programming abstraction with powerful coarse-grained approximation strategies.

These principles should guide the next phase of research on new abstractions for approximation.

9

PROSPECTIVE

The research on approximate computing during this decade has asked more questions than it has answered. To bring approximation mainstream, the community will need to address a swath of open problems.

COMPOSITION Current tools for approximate programmability are stuck in a whole-program paradigm. ACCEPT's compiler analyses and auto-tuner machinery, from Chapter 7, assume that they can observe the entire application at once. Probabilistic assertions, from Chapter 5, fundamentally describe whole-program properties: they constrain a chance that an execution *from program entry* has a certain property. This whole-program perspective on result quality prevents approximate computing from participating in some of the most powerful concepts in programming: local abstractions, separation of concerns, and libraries. A recent exception is Carbin et al.'s Rely language [29], where accuracy is a relationship between module inputs and module outputs. The next stage of research should continue to define what composition means in an approximate context.

EVERYDAY APPROXIMATION Although the buzzword is new, approximate computing is far from a new idea. Approximation is a fundamental in some domains of computer science. Digital signal processing pipelines incorporate accuracy parameters at every stage; work on real-time graphics gets good-enough results more cheaply than an ideal renderer; and there is an entire subfield in theoretical computer science that designs approximation algorithms for intractable problems. All of these approaches are approximations, but they look very different from the kind of system-level approximations in this dissertation. Programming models for approximate computing can learn lessons from these more established disciplines. And the new techniques developed for approximate computing may also be portable in the opposite direction: they could help bring programmability to areas where approximation has traditionally been difficult to reason about.

HIGH-PERFORMANCE COMPUTING & FAULT TOLERANCE Approximate computing is not the same as fault tolerance, but there are clear connections. High-performance computing infrastructures are often large enough that silent failures are a fact of life; and, meanwhile, many HPC applications can tolerate some errors. Approximate computing researchers should build a bridge to do-

main expertise in HPC. Ideally, approximate programming techniques could help express the latent tolerance in HPC systems while constraining the potential for numerical instability and other failure modes.

DEFINING QUALITY One of the principles of this research is that programs have application-specific quality metrics. Determining exactly what constitutes "quality" for a given application, however, can be deceptively difficult. Consider defects in images: how many pixels can be wrong, and by what amount, before the user notices? Are larger areas of slight discoloration better than smaller areas of more intense errors? What makes users care more about the quality of certain photographs than others? These questions are subjective, context sensitive, and poorly defined, but they are critical to determining whether an approximation is successful. For approximate computing to succeed, we need better methodologies for deriving quality metrics. As a first step, we have started preliminary work that applies crowdsourcing to measure human perception of quality. Researchers should also study software engineers' *de facto* processes for assessing output quality in approximate application domains.

CONNECTIONS TO PROBABILISTIC PROGRAMMING Languages for approximate programming usually need to incorporate probabilistic semantics. Recently, the programming languages research community has developed a focus another area that combines programming with probability: *probabilistic programming languages* [18, 33, 69, 93, 94, 150, 225].[1] So far, this direction has assumed a relatively narrow focus: making it easier to express and work with machine-learning models. But the two research areas should cross-pollinate: techniques from one should apply to problems from the other. Researchers should seek fundamental ideas that underly the two sets of programmability challenges.

Even with these outstanding challenges, approximate computing research has an important role to play in the next era of computer system design. As the semiconductor industry exhausts its traditional approaches to scaling performance, and as it becomes more expensive for hardware to enforce reliability, approximation will begin to look less like an academic curiosity. It will become harder to justify preserving abstractions that are oblivious to the resilience in many high-profile applications, and it will become easier to explain the complexity of better abstractions that incorporate approximation.

1 For general background on probabilistic programming, see `probabilistic-programming.org`.

REFERENCES

[1] Tor M. Aamodt and Paul Chow. Compile-time and instruction-set methods for improving floating- to fixed-point conversion accuracy. In *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (May 2008), 26:1–26:27 (cited on page 123).

[2] S. Abdallah, A. Chehab, A. Kayssi, and I.H. Elhajj. TABSH: tag-based stochastic hardware. In *International Conference on Energy Aware Computing Systems & Applications (ICEAC)*, 2013 (cited on page 17).

[3] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: queries with bounded errors and bounded response times on very large data. In *ACM European Conference on Computer Systems (EuroSys)*, 2013 (cited on page 18).

[4] Ismail Akturk, Karen Khatamifard, and Ulya R. Karpuzcu. On quantification of accuracy loss in approximate computing. In *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, 2015 (cited on page 15).

[5] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. In *IEEE Transactions on Computers* 54.7 (2005) (cited on page 16).

[6] Rajeevan Amirtharajah and Anantha P Chandrakasan. A micropower programmable DSP using approximate signal processing based on distributed arithmetic. In *IEEE Journal of Solid-State Circuits* 39.2 (2004), pages 337–347 (cited on page 16).

[7] Jason Ansel, Cy P. Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman P. Amarasinghe. PetaBricks: a language and compiler for algorithmic choice. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2009 (cited on pages 17, 123).

[8] Jason Ansel, Yee Lok Wong, Cy P. Chan, Marek Olszewski, Alan Edelman, and Saman P. Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *International Symposium on Code Generation and Optimization (CGO)*, 2011 (cited on pages 17, 19).

[9] Gary Anthes. Inexact design: beyond fault-tolerance. In *Communications of the ACM* 56.4 (Apr. 2013), pages 18–20 (cited on page 17).

[10] Aslan Askarov and Andrew C. Myers. A semantic framework for declassification and endorsement. In *European Symposium on Programming (ESOP)*, 2010 (cited on page 26).

[11] Lingamneni Avinash, Christian C. Enz, Jean-Luc Nagel, Krishna V. Palem, and Christian Piguet. Energy parsimonious circuit design through probabilistic pruning. In *Design, Automation and Test in Europe (DATE)*, 2011 (cited on page 16).

[12] Rodolfo Jardim de Azevedo, John D. Davis, Karin Strauss, Parikshit Gopalan, Mark Manasse, and Sergey Yekhanin. Zombie: extending memory lifetime by reviving dead blocks. In *International Symposium on Computer Architecture (ISCA)*, 2013 (cited on page 104).

[13] K. Bache and M. Lichman. UCI Machine Learning Repository. 2013. URL: `http://archive.ics.uci.edu/ml` (cited on page 103).

[14] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010 (cited on pages 17, 19, 43, 115, 123, 128).

[15] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012 (cited on page 85).

[16] Bilel Belhadj, Antoine Joubert, Zheng Li, Rodolphe Heliot, and Olivier Temam. Continuous real-world inputs can open up alternative accelerator designs. In *International Symposium on Computer Architecture (ISCA)*, 2013, pages 1–12 (cited on page 123).

[17] Vimal Bhalodia. SCALE DRAM Subsystem Power Analysis. Master's thesis. MIT, 2005 (cited on page 35).

[18] Sooraj Bhat, Johannes Borgström, Andrew D. Gordon, and Claudio Russo. Deriving probability density functions from probabilistic functional programs. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013 (cited on pages 19, 140).

[19] Christian Bienia. Benchmarking Modern Multiprocessors. PhD thesis. Princeton University, Jan. 2011 (cited on page 127).

[20] David Boland and George A. Constantinides. A scalable approach for automated precision analysis. In *ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2012 (cited on page 16).

[21] James Bornholt, Todd Mytkowicz, and Kathryn S. McKinley. Uncertain<T>: a first-order type for uncertain data. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014 (cited on pages 18, 19, 85).

[22] Brett Boston, Adrian Sampson, Dan Grossman, and Luis Ceze. Probability type inference for flexible approximate programming. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015 (cited on pages xi, 7, 14, 179).

[23] S. Braga, A. Sanasi, A. Cabrini, and G. Torelli. Voltage-driven partial-RESET multilevel programming in phase-change memories. In *IEEE Transactions on Electron Devices* 57.10 (2010), pages 2556–2563 (cited on pages 96, 98).

[24] Melvin A. Breuer. Multi-media applications and imprecise computation. In *Euromicro Conference on Digital System Design (DSD)*, 2005 (cited on page 15).

[25] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: a framework for architectural-level power analysis and optimizations. In *International Symposium on Computer Architecture (ISCA)*, 2000 (cited on page 38).

[26] Yu Cai, E.F. Haratsch, O. Mutlu, and Ken Mai. Error patterns in MLC NAND flash memory: measurement, characterization, and analysis. In *Design, Automation and Test in Europe (DATE)*, 2012 (cited on page 103).

[27] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Proving acceptability properties of relaxed nondeterministic approximate programs. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012 (cited on pages 19, 123).

[28] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Verified integrity properties for safe approximate program transformations. In *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*, 2013 (cited on page 19).

[29] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2013 (cited on pages 18, 47–49, 55, 139).

[30] Michael Carbin and Martin Rinard. (Relative) safety properties for relaxed approximate programs. In *Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012 (cited on page 19).

[31] Aaron Carroll and Gernot Heiser. An analysis of power consumption in a smartphone. In *USENIX Annual Technical Conference (ATC)*, 2010 (cited on pages 36, 38).

[32] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: a high-performance storage array architecture for next-generation, non-volatile memories. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010 (cited on page 112).

[33] Arun T. Chaganty, Aditya V. Nori, and Sriram K. Rajamani. Efficiently sampling probabilistic programs via program analysis. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2013 (cited on pages 19, 140).

[34] Lakshmi N. Chakrapani, Bilge E. S. Akgul, Suresh Cheemalavagu, Pinar Korkmaz, Krishna V. Palem, and Balasubramanian Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *Design, Automation and Test in Europe (DATE)*, 2006 (cited on pages 17, 85, 91).

[35] Ik Joon Chang, D. Mohapatra, and K. Roy. A priority-based 6T/8T hybrid SRAM architecture for aggressive voltage scaling in video applications. In *IEEE Transactions on Circuits and Systems for Video Technology* 21.2 (2011), pages 101–112 (cited on page 16).

[36] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2011 (cited on page 20).

[37] Swarat Chaudhuri and Armando Solar-Lezama. Smooth interpretation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2010 (cited on page 20).

[38] Swarat Chaudhuri and Armando Solar-Lezama. Smoothing a program soundly and robustly. In *International Conference on Computer Aided Verification (CAV)*, 2011 (cited on page 20).

[39] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: a benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2009 (cited on page 85).

[40] Tianshi Chen, Yunji Chen, Marc Duranton, Qi Guo, Atif Hashmi, Mikko H. Lipasti, Andrew Nere, Shi Qiu, Michèle Sebag, and Olivier Temam. BenchNN: on the broad potential application scope of hardware neural network accelerators. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2012 (cited on pages 18, 123, 127).

[41] Herman Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. In *The Annals of Mathematical Statistics* 23.4 (1952), pages 493–507 (cited on page 80).

[42] Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Analysis and characterization of inherent application resilience for approximate computing. In *Design Automation Conference (DAC)*, 2013 (cited on page 15).

[43] Vinay K. Chippa, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. StoRM: a stochastic recognition and mining processor. In. 2014 (cited on page 17).

[44] V.K. Chippa, D. Mohapatra, A. Raghunathan, K. Roy, and S.T. Chakradhar. Scalable effort hardware design: exploiting algorithmic resilience for energy efficiency. In *Design Automation Conference (DAC)*, 2010 (cited on page 15).

[45] Adam Chlipala, Leaf Petersen, and Robert Harper. Strict bidirectional type checking. In *ACM SIGPLAN International Workshop on Types in Languages Design and Implementation (TLDI)*, 2005 (cited on page 26).

[46] Clang: a C language family frontend for LLVM. `http://clang.llvm.org` (cited on page 125).

[47] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011 (cited on pages 93, 94).

[48] Michael Cohen, Haitao Steve Zhu, Emgin Ezgi Senem, and Yu David Liu. Energy types. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012 (cited on page 19).

[49] Jason Cong and Karthik Gururaj. Assuring application-level correctness against soft errors. In *IEEE–ACM International Conference on Computer-Aided Design (ICCAD)*, 2011 (cited on page 20).

[50] Keith D Cooper, Mary W Hall, and Ken Kennedy. A methodology for procedure cloning. In *Computer Languages* 19.2 (Apr. 1993), pages 105–117 (cited on page 53).

[51] George B. Dantzig. Discrete-variable extremum problems. In *Operations Research* 5.2 (1957), pages 266–277 (cited on page 125).

[52] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008 (cited on pages 53, 59).

[53] Henry Duwe. Exploiting application level error resilience via deferred execution. Master's thesis. University of Illinois at Urbana-Champaign, 2013 (cited on page 17).

[54] Yong hun Eom and Brian Demsky. Self-stabilizing Java. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2012 (cited on page 20).

[55] D. Ernst, Nam Sung Kim, S. Das, S. Pant, R. Rao, Toan Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2003 (cited on page 35).

[56] Michael D. Ernst. Type Annotations specification (JSR 308). `http://types.cs.washington.edu/jsr308/`. 2008 (cited on pages 24, 59, 125).

[57] Hadi Esmaeilzadeh. Approximate Acceleration for a Post Multicore Era. PhD thesis. University of Washington, 2013 (cited on pages 12, 13).

[58] Hadi Esmaeilzadeh, Kangqi Ni, and Mayur Naik. *Expectation-Oriented Framework for Automating Approximate Programming*. Technical report GT-CS-13-07. Georgia Institute of Technology, 2013. URL: `http://hdl.handle.net/1853/49755` (cited on pages 18, 47).

[59] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012 (cited on pages 7, 8, 12, 33, 38, 48, 51, 57, 61, 62, 64, 85, 91–93, 100, 123, 128, 138).

[60] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012 (cited on pages 7, 8, 12, 17, 18, 49, 85, 115, 117, 123, 127, 128, 138).

[61] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *International Symposium on Computer Architecture (ISCA)*, 2007 (cited on pages 36, 38).

[62] Shuangde Fang, Zidong Du, Yuntan Fang, Yuanjie Huang, Yang Chen, Lieven Eeckhout, Olivier Temam, Huawei Li, Yunji Chen, and Chengyong Wu. Performance portability across heterogeneous SoCs using a generalized library-based approach. In *ACM Transactions on Architecture and Code Optimization (TACO)* 11.2 (June 2014), 21:1–21:25 (cited on page 17).

[63] Yuntan Fang, Huawei Li, and Xiaowei Li. A fault criticality evaluation framework of digital systems for error tolerant video applications. In *Asian Test Symposium (ATS)*, 2011 (cited on page 15).

[64] Yuntan Fang, Huawei Li, and Xiaowei Li. SoftPCM: enhancing energy efficiency and lifetime of phase change memory in video applications via approximate write. In *Asian Test Symposium (ATS)*, 2012 (cited on page 16).

[65] Krisztián Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *International Symposium on Computer Architecture (ISCA)*, 2002 (cited on pages 34, 35).

[66] Jeffrey S. Foster. Type Qualifiers: Lightweight Specifications to Improve Software Quality. PhD thesis. University of California, Berkeley, Dec. 2002 (cited on page 125).

[67] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 1999 (cited on page 27).

[68] Mrinmoy Ghosh and Hsien-Hsin S. Lee. Smart refresh: an enhanced memory controller design for reducing energy in conventional and 3D die-stacked DRAMs. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007 (cited on page 34).

[69] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2008 (cited on pages 8, 19, 71, 140).

[70] Beayna Grigorian and Glenn Reinman. Accelerating divergent applications on SIMD architectures using neural networks. In *IEEE International Conference on Computer Design*, 2014 (cited on pages 17, 18).

[71] Beayna Grigorian and Glenn Reinman. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *NASA–ESA Conference On Adaptive Hardware And Systems (AHS)*, 2014 (cited on page 19).

[72] V. Gupta, D. Mohapatra, Sang Phill Park, A. Raghunathan, and K. Roy. IMPACT: imprecise adders for low-power approximate computing. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2011 (cited on page 16).

[73] Vaibhav Gupta, Debabrata Mohapatra, Anand Raghunathan, and Kaushik Roy. Low-power digital signal processing using approximate adders. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 32.1 (Jan. 2013), pages 124–137 (cited on page 16).

[74] Andrew Hay, Karin Strauss, Timothy Sherwood, Gabriel H. Loh, and Doug Burger. Preventing PCM banks from seizing too much power. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011 (cited on page 107).

[75] Rajamohana Hegde and Naresh R. Shanbhag. Energy-efficient signal processing via algorithmic noise-tolerance. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 1999 (cited on page 16).

[76] Andreas Heinig, Vincent John Mooney, Florian Schmoll, Peter Marwedel, Krishna V. Palem, and Michael Engel. Classification-based improvement of application robustness and quality of service in probabilistic computer systems. In *International Conference on Architecture of Computing Systems (ARCS)*, 2012 (cited on page 15).

[77] S. Hiranandani, K. Kennedy, C.-W. Tseng, and S. Warren. The D editor: a new interactive parallel programming tool. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 1994 (cited on page 119).

[78] Caglar Hizli. Energy Aware Probabilistic Arithmetics. Master's thesis. Eindhoven University of Technology, 2013 (cited on pages 16, 62).

[79] Chen-Han Ho, M. de Kruijf, K. Sankaralingam, B. Rountree, M. Schulz, and B.R. De Supinski. Mechanisms and evaluation of cross-layer fault-tolerance for supercomputing. In *IEEE International Conference on Parallel Processing (ICPP)*, 2012 (cited on page 18).

[80] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin C. Rinard. Dynamic knobs for responsive power-aware computing. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011 (cited on pages 17, 19).

[81] Daniel E. Holcomb and Kevin Fu. QBF-based synthesis of optimal word-splitting in approximate multi-level storage cells. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014 (cited on page 99).

[82] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. In *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23.3 (2001) (cited on pages 29, 165).

[83] Engin Ipek, Jeremy Condit, Edmund B. Nightingale, Doug Burger, and Thomas Moscibroda. Dynamically replicated memory: building reliable systems from nanoscale resistive memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010 (cited on page 101).

[84] Szymon Jakubczak and Dina Katabi. SoftCast: clean-slate scalable wireless video. In *Workshop on Wireless of the Students, by the Students, for the Students (S3)*, 2010 (cited on page 18).

[85] Lei Jiang, Bo Zhao, Youtao Zhang, Jun Yang, and Bruce R. Childers. Improving write operations in MLC phase change memory. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2012 (cited on pages 97–99, 107).

[86] A.B. Kahng, Seokhyeong Kang, R. Kumar, and J. Sartori. Designing a processor from the ground up to allow voltage/reliability tradeoffs. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010 (cited on pages 16, 62).

[87] Andrew B. Kahng and Seokhyeong Kang. Accuracy-configurable adder for approximate arithmetic designs. In *Design Automation Conference (DAC)*, 2012 (cited on page 16).

[88] Georgios Karakonstantis, Debabrata Mohapatra, and Kaushik Roy. Logic and memory design based on unequal error protection for voltage-scalable, robust and adaptive DSP systems. In *Journal of Signal Processing Systems* 68.3 (Sept. 2012), pages 415–431 (cited on page 16).

[89] Ulya R. Karpuzcu, Ismail Akturk, and Nam Sung Kim. Accordion: toward soft near-threshold voltage computing. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2014 (cited on pages 16, 48, 61).

[90] Zvi M. Kedem, Vincent J. Mooney, Kirthi Krishna Muntimadugu, and Krishna V. Palem. An approach to energy-error tradeoffs in approximate ripple carry adders. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2011 (cited on pages 16, 62).

[91] Daya S. Khudia, Babak Zamirai, Mehrzad Samadi, and Scott Mahlke. Rumba: an online quality management system for approximate computing. In *International Symposium on Computer Architecture (ISCA)*, 2015 (cited on page 19).

[92] Daya Shanker Khudia and Scott Mahlke. Harnessing soft computations for low-budget fault tolerance. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014 (cited on page 16).

[93] Oleg Kiselyov and Chung-Chieh Shan. Embedded probabilistic programming. In *IFIP Working Conference on Domain-Specific Languages (DSL)*, 2009 (cited on pages 19, 140).

[94] Daphne Koller, David McAllester, and Avi Pfeffer. Effective Bayesian inference for stochastic programs. In *AAAI Conference on Artificial Intelligence (AAAI)*, 1997 (cited on pages 19, 140).

[95] D. Kozen. Semantics of probabilistic programs. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, Oct. 1979, pages 101–114 (cited on pages 19, 74, 87).

[96] M. de Kruijf and K. Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *Workshop on Silicon Errors in Logic: System Effects (SELSE)*, 2009 (cited on pages 15, 23).

[97] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *International Symposium on Computer Architecture (ISCA)*, 2010 (cited on pages 16, 18, 23, 35, 91).

[98] Animesh Kumar. SRAM Leakage-Power Optimization Framework: a System Level Approach. PhD thesis. University of California at Berkeley, 2008 (cited on page 35).

[99] Animesh Kumar, Jan Rabaey, and Kannan Ramchandran. SRAM supply voltage scaling: a reliability perspective. In *International Symposium on Quality Electronic Design (ISQED)*, 2009 (cited on page 16).

[100] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: verification of probabilistic real-time systems. In *International Conference on Computer Aided Verification (CAV)*, 2011 (cited on page 19).

[101] Chris Lattner and Vikram Adve. LLVM: a compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, 2004 (cited on pages 73, 81, 115, 125).

[102] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *International Symposium on Computer Architecture (ISCA)*, 2009 (cited on pages 92, 93, 95, 100, 106).

[103] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. ERSA: error resilient system architecture for probabilistic applications. In *Design, Automation and Test in Europe (DATE)*, 2010 (cited on pages 17, 23, 91).

[104] A. Legay and B. Delahaye. Statistical model checking: a brief overview. In *Quantitative Models: Expressiveness and Analysis* (2010) (cited on page 19).

[105] Boxun Li, Peng Gu, Yi Shan, Yu Wang, Yiran Chen, and Huazhong Yang. RRAM-based analog approximate computing. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* (2015) (cited on page 18).

[106] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008 (cited on page 16).

[107] Sheng Li, Jung Ho Ahn, R.D. Strong, J.B. Brockman, D.M. Tullsen, and N.P. Jouppi. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2009 (cited on page 38).

[108] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2007. URL: http://dx.doi.org/10.1109/HPCA.2007.346196 (cited on pages 15, 23, 40).

[109] Xuanhua Li and Donald Yeung. Exploiting application-level correctness for low-cost fault tolerance. In *Journal of Instruction-Level Parallelism* (2008). URL: http://www.jilp.org/vol10/v10paper10.pdf (cited on page 15).

[110] Xuanhua Li and Donald Yeung. Exploiting soft computing for increased fault tolerance. In *Workshop on Architectural Support for Gigascale Integration (ASGI)*, 2006. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.78.2997 (cited on pages 15, 23).

[111] Jinghang Liang, Jie Han, and Fabrizio Lombardi. New metrics for the reliability of approximate and probabilistic adders. In *IEEE Transactions on Computers* 99 (2012) (cited on page 16).

[112] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. Optimizing NAND flash-based SSDs via retention relaxation. In *USENIX Conference on File and Storage Technologies (FAST)*, 2012 (cited on pages 16, 106).

[113] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: saving refresh-power in mobile devices through critical data partitioning. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011 (cited on pages 15, 16, 23, 35, 40, 92, 93, 100, 115).

[114] Xiaoxiao Liu, Mengjie Mao, Beiye Liu, Hai Li, Yiran Chen, Boxun Li, Yu Wang, Hao Jiang, Mark Barnell, Qing Wu, and Jianhua Yang. RENO: a high-efficient reconfigurable neuromorphic computing accelerator design. In *Design Automation Conference (DAC)*, 2015 (cited on page 18).

[115] LLVM Project. LLVM Interpreter. http://llvm.org/docs/doxygen/html/classllvm_1_1Interpreter.html. 2013 (cited on page 82).

[116] G. Long, F. T. Chong, D. Franklin, J. Gilbert, and D. Fan. Soft coherence: preliminary experiments with error-tolerant memory consistency in numerical applications. In *Workshop on Chip Multiprocessor Memory Systems and Interconnects (CMP-MSI)*, 2009 (cited on page 17).

[117] Jan Lucas, Mauricio Alvarez Mesa, Michael Andersch, and Ben Juurlink. Sparkk: quality-scalable approximate storage in dram. In *The Memory Forum*, 2014 (cited on page 16).

[118] Chong Luo, Jun Sun, and Feng Wu. Compressive network coding for approximate sensor data gathering. In *IEEE Global Communications Conference (GLOBECOM)*, 2011 (cited on page 18).

[119] Aqeel Mahesri and Vibhore Vardhan. Power consumption breakdown on a modern laptop. In *Workshop on Power-Aware Computer Systems (PACS)*, 2004 (cited on page 36).

[120] Vikash K. Mansinghka, Eric M. Jonas, and Joshua B. Tenenbaum. *Stochastic Digital Circuits for Probabilistic Inference*. Technical report MIT-CSAIL-TR-2008-069. MIT, 2008 (cited on page 17).

[121] Lawrence McAfee and Kunle Olukotun. EMEURO: a framework for generating multi-purpose accelerators via deep learning. In *International Symposium on Code Generation and Optimization (CGO)*, 2015 (cited on page 18).

[122] Frank D. McSherry. Privacy integrated queries: an extensible platform for privacy-preserving data analysis. In *ACM SIGMOD International Conference on Management of Data*, 2009 (cited on page 85).

[123] Jiayuan Meng, Srimat Chakradhar, and Anand Raghunathan. Best-effort parallel execution framework for recognition and mining applications. In *IEEE International Parallel & Distributed Processing Symposium*, 2009 (cited on page 15).

[124] Jiayuan Meng, Anand Raghunathan, Srimat Chakradhar, and Surendra Byna. Exploiting the forgiving nature of applications for scalable parallel execution. In *IEEE International Parallel & Distributed Processing Symposium*, 2010 (cited on page 17).

[125] Jin Miao. Modeling and synthesis of approximate digital circuits. PhD thesis. The University of Texas at Austin, 2014 (cited on page 16).

[126] Jin Miao, Ku He, Andreas Gerstlauer, and Michael Orshansky. Modeling and synthesis of quality-energy optimal approximate adders. In *IEEE–ACM International Conference on Computer-Aided Design (ICCAD)*, 2012 (cited on page 16).

[127] N. Mielke, T. Marquart, Ning Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L.R. Nevill. Bit error rate in NAND flash memories. In *IEEE International Reliability Physics Symposium*, 2008 (cited on page 103).

[128] Joshua San Miguel, Mario Badr, and Natalie Enright Jerger. Load value approximation. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014 (cited on page 17).

[129] T. Minka, J.M. Winn, J.P. Guiver, and D.A. Knowles. Infer.NET 2.5. Microsoft Research Cambridge. `http://research.microsoft.com/infernet`. 2012 (cited on page 71).

[130] Sasa Misailovic, Michael Carbin, Sara Achour, Zichao Qi, and Martin C. Rinard. Chisel: reliability- and accuracy-aware optimization of approximate computational kernels. In *ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2014 (cited on pages 18, 47).

[131] Sasa Misailovic, Deokhwan Kim, and Martin Rinard. *Parallelizing Sequential Programs With Statistical Accuracy Tests*. Technical report MIT-CSAIL-TR-2010-038. MIT, Aug. 2010 (cited on pages 17, 117, 122, 123).

[132] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *International Static Analysis Symposium (SAS)*, 2011 (cited on pages 17, 78, 117).

[133] Sasa Misailovic, Stelios Sidiroglou, Hank Hoffman, and Martin Rinard. Quality of service profiling. In *International Conference on Software Engineering (ICSE)*, 2010 (cited on pages 19, 23, 123, 127, 128).

[134] Sasa Misailovic, Stelios Sidiroglou, and Martin Rinard. Dancing with uncertainty. In *Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012 (cited on pages 17, 117, 122).

[135] Prashanth Mohan, Abhradeep Thakurta, Elaine Shi, Dawn Song, and David Culler. GUPT: privacy preserving data analysis made easy. In *ACM SIGMOD International Conference on Management of Data*, 2012 (cited on page 85).

[136] Debabrata Mohapatra, Vinay K Chippa, Anand Raghunathan, and Kaushik Roy. Design of voltage-scalable meta-functions for approximate computing. In *Design, Automation and Test in Europe (DATE)*, 2011 (cited on page 16).

[137] Thierry Moreau, Mark Wyse, Jacob Nelson, Adrian Sampson, Hadi Esmaeilzadeh, Luis Ceze, and Mark Oskin. SNNAP: approximate computing on programmable SoCs via neural acceleration. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2015 (cited on pages 12, 13, 18, 123, 127, 128).

[138] Andrew C. Myers. JFlow: practical mostly-static information flow control. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999 (cited on page 9).

[139] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *Design, Automation and Test in Europe (DATE)*, 2010 (cited on pages 17, 85, 91).

[140] Karthik Natarajan, Heather Hanson, Stephen W. Keckler, Charles R. Moore, and Doug Burger. Microprocessor pipeline energy analysis. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2003 (cited on page 38).

[141] T. Nirschl, J.B. Phipp, T.D. Happ, G.W. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C.-F. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y.C. Chen, Y. Zhu, R. Bergmann, H.-L. Lung, and C. Lam. Write strategies for 2 and 4-bit multi-level phase-change memory. In *IEEE International Electron Devices Meeting (IEDM)*, 2007 (cited on pages 98, 104).

[142] Krishna Palem and Avinash Lingamneni. What to do about the end of Moore's law, probably! In *Design Automation Conference (DAC)*, 2012 (cited on page 17).

[143] David J. Palframan, Nam Sung Kim, and Mikko H. Lipasti. Precision-aware soft error protection for GPUs. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2014 (cited on page 16).

[144] A. Pantazi, A. Sebastian, N. Papandreou, MJ Breitwisch, C. Lam, H. Pozidis, and E. Eleftheriou. Multilevel phase change memory modeling and experimental characterization. In *European Phase Change and Ovonics Symposium (EPCOS)*, 2009 (cited on pages 97, 98).

[145] N. Papandreou, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, H. Pozidis, and E. Eleftheriou. Multilevel phase-change memory. In *IEEE International Conference on Electronics Circuits and Systems (ICECS)*, 2010 (cited on pages 96, 97).

[146] N. Papandreou, H. Pozidis, T. Mittelholzer, G.F. Close, M. Breitwisch, C. Lam, and E. Eleftheriou. Drift-tolerant multilevel phase-change memory. In *IEEE International Memory Workshop (IMW)*, 2011 (cited on page 94).

[147] N. Papandreou, H. Pozidis, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, and E. Eleftheriou. Programming algorithms for multilevel phase-change memory. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011 (cited on page 97).

[148] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2008 (cited on pages 36, 125).

[149] Sungwoo Park, Frank Pfenning, and Sebastian Thrun. A probabilistic language based upon sampling functions. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2005 (cited on pages 71, 85).

[150] Avi Pfeffer. *A General Importance Sampling Algorithm for Probabilistic Programs*. Technical report TR-12-07. Harvard University, 2007 (cited on pages 19, 140).

[151] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: synthesis-aided compiler for low-power spatial architectures. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014 (cited on page 52).

[152] H. Pozidis, N. Papandreou, A. Sebastian, T. Mittelholzer, M. BrightSky, C. Lam, and E. Eleftheriou. A framework for reliability assessment in multilevel phase-change memory. In *IEEE International Memory Workshop (IMW)*, 2012 (cited on page 97).

[153] M. K. Qureshi, M. M. Franceschini, and L. A. Lastras-Montano. Improving read performance of phase change memories via write cancellation and write pausing. In *International Symposium on High-Performance Computer Architecture (HPCA)*, 2010 (cited on pages 97, 98).

[154] Moinuddin K. Qureshi. Pay-as-you-go: low-overhead hard-error correction for phase change memories. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011 (cited on page 100).

[155] Moinuddin K. Qureshi, Michele M. Franceschini, Luis A. Lastras-Montaño, and John P. Karidis. Morphable memory system: a robust architecture for exploiting multi-level phase change memories. In *International Symposium on Computer Architecture (ISCA)*, 2010 (cited on pages 96, 97, 100).

[156] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *International Symposium on Computer Architecture (ISCA)*, 2009 (cited on pages 92, 93).

[157] A. Rahimi, A. Marongiu, R.K. Gupta, and L. Benini. A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters. In *IEEE–ACM–IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2013 (cited on page 16).

[158] Amir Rahmati, Matthew Hicks, Daniel E. Holcomb, and Kevin Fu. Probable cause: the deanonymizing effects of approximate DRAM. In *International Symposium on Computer Architecture (ISCA)*, 2015 (cited on page 16).

[159] Norman Ramsey and Avi Pfeffer. Stochastic lambda calculus and monads of probability distributions. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002 (cited on pages 19, 71).

[160] Ashish Ranjan, Arnab Raha, Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. ASLAN: synthesis of approximate sequential circuits. In *Design, Automation and Test in Europe (DATE)*, 2014 (cited on page 16).

[161] Ashish Ranjan, Swagath Venkataramani, Xuanyao Fong, Kaushik Roy, and Anand Raghunathan. Approximate storage for energy efficient spintronic memories. In *Design Automation Conference (DAC)*, 2015 (cited on page 16).

[162] Benjamin Recht, Christopher Re, Stephen J. Wright, and Feng Niu. Hogwild: a lock-free approach to parallelizing stochastic gradient descent. In *Conference on Neural Information Processing Systems (NIPS)*, 2011 (cited on page 17).

[163] Jason Reed and Benjamin C. Pierce. Distance makes the types grow stronger: a calculus for differential privacy. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2010 (cited on page 85).

[164] Lakshminarayanan Renganarayanan, Vijayalakshmi Srinivasan, Ravi Nair, and Daniel Prener. Programming with relaxed synchronization. In *Workshop on Relaxing Synchronization for Multicore and Manycore Scalability (RACES)*, 2012 (cited on pages 17, 115, 117, 122).

[165] Martin Rinard. Parallel synchronization-free approximate data structure construction. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2013 (cited on pages 117, 122).

[166] Martin Rinard, Henry Hoffmann, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computing. In *Onward!*, 2010 (cited on page 23).

[167] Michael F. Ringenburg. Dynamic Analyses of Result Quality in Energy-Aware Approximate Programs. PhD thesis. University of Washington, 2014 (cited on page 13).

[168] Michael F. Ringenburg and Sung-Eun Choi. Optimizing loop-level parallelism in Cray XMT applications. In *Cray User Group Proceedings*, May 2009 (cited on page 119).

[169] Michael F. Ringenburg, Adrian Sampson, Isaac Ackerman, Luis Ceze, and Dan Grossman. Monitoring and debugging the quality of results in approximate programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015 (cited on pages 13, 19, 48).

[170] Indrajit Roy, Srinath T. V. Setty, Ann Kilzer, Vitaly Shmatikov, and Emmett Witchel. Airavat: security and privacy for MapReduce. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2010 (cited on page 85).

[171] Pooja Roy, Rajarshi Ray, Chundong Wang, and Weng Fai Wong. ASAC: automatic sensitivity analysis for approximate computing. In *ACM SIGPLAN–SIGBED Conference on Languages, Compilers, Tools and Theory for Embedded Systems (LCTES)*, 2014 (cited on page 20).

[172] Sourya Roy, Tyler Clemons, S M Faisal, Ke Liu, Nikos Hardavellas, and Srinivasan Parthasarathy. *Elastic Fidelity: Trading-off Computational Accuracy for Energy Reduction*. Technical report NWU-EECS-11-02. Northwestern University, 2011 (cited on page 15).

[173] Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, David H. Bailey, Costin Iancu, and David Hough. Precimonious: tuning assistant for floating-point precision. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2013 (cited on pages 18, 123).

[174] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. In *IEEE Journal on Selected Areas in Communications, special issue on Formal Methods for Security* 21.1 (2003) (cited on pages 9, 23).

[175] Mastooreh Salajegheh, Yue Wang, Kevin Fu, Anxiao Jiang, and Erik Learned-Miller. Exploiting half-wits: smarter storage for low-power devices. In *USENIX Conference on File and Storage Technologies (FAST)*, 2011 (cited on page 16).

[176] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. Paraprox: pattern-based approximation for data parallel applications. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014 (cited on pages 8, 17, 47, 115, 123).

[177] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: self-tuning approximation for graphics engines. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013 (cited on pages 17, 115).

[178] Alanson P. Sample, Daniel J. Yeager, Pauline S. Powledge, Alexander V. Mamishev, and Joshua R. Smith. Design of an RFID-based battery-free programmable sensing platform. In *IEEE Transactions on Instrumentation and Measurement* 57.11 (Nov. 2008), pages 2608–2615 (cited on page 128).

[179] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. *EnerJ: Approximate Data Types for Safe and General Low-Power Computation — Full Proofs*. Technical report UW-CSE-10-12-01. University of Washington, 2011 (cited on pages 14, 165).

[180] Adrian Sampson, Werner Dietl, Emily Fortuna, Danushen Gnanapragasam, Luis Ceze, and Dan Grossman. EnerJ: approximate data types for safe and general low-power computation. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2011 (cited on pages xi, 14, 47, 59, 67, 91–93, 123, 127, 128).

[181] Adrian Sampson, Jacob Nelson, Karin Strauss, and Luis Ceze. Approximate storage in solid-state memories. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013 (cited on pages xi, 7, 14).

[182] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Expressing and verifying probabilistic assertions. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014 (cited on pages xi, 7, 14, 117).

[183] Adrian Sampson, Pavel Panchekha, Todd Mytkowicz, Kathryn S. McKinley, Dan Grossman, and Luis Ceze. Probabilistic Assertions: Extended Semantics and Proof. ACM Digital Library auxiliary materials accompanying the paper. `http://dx.doi.org/10.1145/2594291.2594294`. 2014 (cited on pages 14, 189).

[184] Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2013 (cited on pages 19, 71, 77).

[185] John Sartori and Rakesh Kumar. Branch and data herding: reducing control and memory divergence for error-tolerant GPU applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2012 (cited on page 17).

[186] Stuart Schechter, Gabriel H. Loh, Karin Strauss, and Doug Burger. Use ECP, not ECC, for hard failures in resistive memories. In *International Symposium on Computer Architecture (ISCA)*, 2010 (cited on pages 100, 101, 104, 112).

[187] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. In *ACM Conference on Programming Language Design and Implementation (PLDI)*, 2014 (cited on pages 18, 122).

[188] Eric Schulte, Jonathan Dorn, Stephen Harding, Stephanie Forrest, and Westley Weimer. Post-compiler software optimization for reducing energy. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014 (cited on page 18).

[189] Sayandeep Sen, Syed Gilani, Shreesha Srinath, Stephen Schmitt, and Suman Banerjee. Design and implementation of an "approximate" communication system for wireless media applications. In *ACM SIGCOMM*, 2010 (cited on page 18).

[190] Nak Hee Seong, Dong Hyuk Woo, V. Srinivasan, J.A. Rivers, and H.-H.S. Lee. SAFER: stuck-at-fault error recovery for memories. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2010 (cited on page 100).

[191] Muhammad Shafique, Waqas Ahmad, Rehan Hafiz, and Jörg Henkel. A low latency generic accuracy configurable adder. In *Design Automation Conference (DAC)*, 2015 (cited on page 16).

[192] Q. Shi, H. Hoffmann, and O. Khan. A HW-SW multicore architecture to tradeoff program accuracy and resilience overheads. In *Computer Architecture Letters* (2014) (cited on page 16).

[193] Majid Shoushtari, Abbas BanaiyanMofrad, and Nikil Dutt. Exploiting partially-forgetful memories for approximate computing. In *IEEE Embedded Systems Letters* 7.1 (Mar. 2015), pages 19–22 (cited on page 16).

[194] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin C. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2011 (cited on pages 17, 23, 115, 117, 122).

[195] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi. A numerical optimization-based methodology for application robustification: transforming applications for error tolerance. In *IEEE–IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010 (cited on page 20).

[196] Jacob Sorber, Alexander Kostadinov, Matthew Garber, Matthew Brennan, Mark D. Corner, and Emery D. Berger. Eon: a language and runtime system for perpetual systems. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2007 (cited on page 19).

[197] Renée St. Amant, Amir Yazdanbakhsh, Jongse Park, Bradley Thwaites, Hadi Esmaeilzadeh, Arjang Hassibi, Luis Ceze, and Doug Burger. General-purpose code acceleration with limited-precision analog computation. In *International Symposium on Computer Architecture (ISCA)*, 2014 (cited on pages 18, 117, 123, 138).

[198] Phillip Stanley-Marbell. Encoding efficiency of digital number representations under deviation constraints. In *Information Theory Workshop (ITW)*, 2009 (cited on page 16).

[199] Phillip Stanley-Marbell and Diana Marculescu. A programming model and language implementation for concurrent failureprone hardware. In *Workshop on Programming Models for Ubiquitous Parallelism (PMUP)*, 2006. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.121.9864 (cited on page 18).

[200] Phillip Stanley-Marbell and Martin Rinard. Lax: driver interfaces for approximate sensor device access. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015 (cited on page 18).

[201] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, Jung-Hyuk Choi, Jang-Rae Kim, and Hyung-Kyu Lim. A 3.3 V 32 Mb NAND flash memory with incremental step pulse programming scheme. In *IEEE Journal of Solid-State Circuits* 30.11 (1995), pages 1149–1156 (cited on page 97).

[202] Ayswarya Sundaram, Ameen Aakel, Derek Lockhart, Darshan Thaker, and Diana Franklin. Efficient fault tolerance in multi-media applications through selective instruction replication. In *Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies*, 2008 (cited on page 16).

[203] K. Takeuchi, T. Tanaka, and T. Tanzawa. A multipage cell architecture for high-speed programming multilevel NAND flash memories. In *IEEE Journal of Solid-State Circuits* 33.8 (1998), pages 1228–1238 (cited on pages 96, 98, 104).

[204] Olivier Temam. A defect-tolerant accelerator for emerging high-performance applications. In *International Symposium on Computer Architecture (ISCA)*, 2012 (cited on pages 18, 123, 127).

[205] Texas Instruments, Inc. MSP430 Ultra-Low Power Microcontrollers. http://www.ti.com/msp430 (cited on page 127).

[206] Darshan D. Thaker, Diana Franklin, John Oliver, Susmit Biswas, Derek Lockhart, Tzvetan S. Metodi, and Frederic T. Chong. Characterization of error-tolerant applications when protecting control data. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2006 (cited on page 15).

[207] Anna Thomas and Karthik Pattabiraman. Llfi: an intermediate code level fault injector for soft computing applications. In *Workshop on Silicon Errors in Logic: System Effects (SELSE)*, 2013 (cited on page 15).

[208] Bradley Thwaites, Gennady Pekhimenko, Amir Yazdanbakhsh, Jongse Park, Girish Mururu, Hadi Esmaeilzadeh, Onur Mutlu, and Todd Mowry. Rollback-free value prediction with approximate loads. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2014 (cited on page 17).

[209] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *Dynamic Languages Symposium (DLS)*, 2006 (cited on page 54).

[210] Jonathan Ying Fai Tong, David Nagle, and Rob. A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. In *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 8.3 (2000) (cited on pages 16, 35, 123).

[211] Hung-Wei Tseng, Laura M. Grupp, and Steven Swanson. Underpowering NAND flash: profits and perils. In *Design Automation Conference (DAC)*, 2013 (cited on page 16).

[212] G. Tziantzioulis, A. M. Gok, S. M. Faisal, N. Hardavellas, S. Ogrenci-Memik, and S. Parthasarathy. b-HiVE: a bit-level history-based error model with value correlation for voltage-scaled integer and floating point units. In *Design Automation Conference (DAC)*, (cited on page 16).

[213] Swagath Venkataramani, Vinay K. Chippa, Srimat T. Chakradhar, Kaushik Roy, and Anand Raghunathan. Quality programmable vector processors for approximate computing. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2013 (cited on pages 33, 47, 48, 51, 57, 61, 64, 67).

[214] Swagath Venkataramani, Anand Raghunathan, Jie Liu, and Mohammed Shoaib. Scalable-effort classifiers for energy-efficient machine learning. In *Design Automation Conference (DAC)*, 2015 (cited on page 17).

[215] Swagath Venkataramani, Kaushik Roy, and Anand Raghunathan. Substitute-and-simplify: a unified design paradigm for approximate and quality configurable circuits. In *Design, Automation and Test in Europe (DATE)*, 2013 (cited on page 16).

[216] Swagath Venkataramani, Amit Sabne, Vivek Kozhikkottu, Kaushik Roy, and Anand Raghunathan. SALSA: systematic logic synthesis of approximate circuits. In *Design Automation Conference (DAC)*, 2012 (cited on page 16).

[217] Rangharajan Venkatesan, Amit Agarwal, Kaushik Roy, and Anand Raghunathan. MACACO: modeling and analysis of circuits for approximate computing. In *IEEE–ACM International Conference on Computer-Aided Design (ICCAD)*, 2011 (cited on page 16).

[218] Ajay K. Verma, Philip Brisk, and Paolo Ienne. Variable latency speculative addition: a new paradigm for arithmetic circuit design. In *Design, Automation and Test in Europe (DATE)*, 2008 (cited on page 16).

[219] Benjamin Vigoda, David Reynolds, Jeffrey Bernstein, Theophane Weber, and Bill Bradley. Low power logic for statistical inference. In *International Symposium on Low Power Electronics and Design (ISLPED)*, 2010 (cited on page 17).

[220] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011 (cited on page 93).

[221] Abraham Wald. Sequential tests of statistical hypotheses. In *The Annals of Mathematical Statistics* 16.2 (1945), pages 117–186 (cited on page 81).

[222] Lucas Wanner and Mani Srivastava. ViRUS: virtual function replacement under stress. In *USENIX Workshop on Power-Aware Computing and Systems (HotPower)*, 2014 (cited on page 17).

[223] M. Weber, M. Putic, Hang Zhang, J. Lach, and Jiawei Huang. Balancing adder for error tolerant applications. In *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2013 (cited on pages 16, 48, 61).

[224] Edwin Westbrook and Swarat Chaudhuri. *A Semantics for Approximate Program Transformations*. Technical report Preprint: `arXiv:1304.5531`. 2013 (cited on page 19).

[225] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2011 (cited on pages 19, 140, 192).

[226] Vicky Wong and Mark Horowitz. Soft error resilience of probabilistic inference applications. In *Workshop on Silicon Errors in Logic: System Effects (SELSE)*, 2006 (cited on pages 15, 23, 40).

[227] A. Yazdanbakhsh, D. Mahajan, B. Thwaites, Jongse Park, A. Nagendrakumar, S. Sethuraman, K. Ramkrishnan, N. Ravindran, R. Jariwala, A. Rahimi, H. Esmaeilzadeh, and K. Bazargan. Axilog: language support for approximate hardware design. In *Design, Automation and Test in Europe (DATE)*, 2015 (cited on page 16).

[228] Rong Ye, Ting Wang, Feng Yuan, Rakesh Kumar, and Qiang Xu. On reconfiguration-oriented approximate adder design and its application. In *IEEE–ACM International Conference on Computer-Aided Design (ICCAD)*, 2013 (cited on page 16).

[229] Thomas Y. Yeh, Petros Faloutsos, Milos Ercegovac, Sanjay J. Patel, and Glen Reinman. The art of deception: adaptive precision reduction for area efficient physics acceleration. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007 (cited on page 16).

[230] Thomas Y. Yeh, Glenn Reinman, Sanjay J. Patel, and Petros Faloutsos. Fool me twice: exploring and exploiting error tolerance in physics-based animation. In *ACM Transactions on Graphics* 29.1 (Dec. 2009) (cited on page 15).

[231] Sungkap Yeo, Nak Hee Seong, and Hsien-Hsin S. Lee. Can multi-level cell PCM be reliable and usable? Analyzing the impact of resistance drift. In *Workshop on Duplicating, Deconstructing and Debunking (WDDD)*, 2012 (cited on pages 96, 97, 103, 108).

[232] Yavuz Yetim, Sharad Malik, and Margaret Martonosi. CommGuard: mitigating communication errors in error-prone parallel execution. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2015 (cited on page 17).

[233] Yavuz Yetim, Margaret Martonosi, and Sharad Malik. Extracting useful computation from error-prone processors for streaming applications. In *Design, Automation and Test in Europe (DATE)*, 2013 (cited on page 17).

[234] Håkan L. S. Younes and Reid G. Simmons. Statistical probabilistic model checking with a focus on time-bounded properties. In *Information and Computation* 204.9 (2006), pages 1368–1409 (cited on page 81).

[235] Håkan L.S. Younes. Error control for probabilistic model checking. In *Verification, Model Checking, and Abstract Interpretation* (2006), pages 142–156 (cited on page 80).

[236] Qian Zhang, Feng Yuan, Rong Ye, and Qiang Xu. ApproxIt: an approximate computing framework for iterative methods. In *Design Automation Conference (DAC)*, 2014 (cited on page 19).

[237] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. In *International Symposium on Computer Architecture (ISCA)*, 2009 (cited on pages 92, 93, 101).

[238] Ning Zhu, Wang Ling Goh, and Kiat Seng Yeo. An enhanced low-power high-speed adder for error-tolerant application. In *International Symposium on Integrated Circuits (ISIC)*, 2009 (cited on page 16).

[239] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin C. Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL)*, 2012 (cited on pages 17, 117).

Part V

APPENDIX: SEMANTICS AND THEOREMS

A

ENERJ: NONINTERFERENCE PROOF

A.1 TYPE SYSTEM

This appendix gives the full formalism for EnerJ, the programming language for approximate computing from Chapter 3. It is based on the EnerJ paper's accompanying technical report [179].

This section introduces the core type system, which is made up of type qualifiers that extend Featherweight Java [82]. Section A.2 describes the big-step operational semantics that define the language's runtime system. Section A.3 proves a number of properties about the language, the most important of which is *non-interference* (intuitively, that the precise part of the program is unaffected by the approximate part).

A.1.1 *Ordering*

We introduce a strict ordering on the language's type qualifiers:

$\boxed{q <:_{\mathrm{q}} q'}$ ordering of precision qualifiers

$$\frac{q \neq \texttt{top}}{q <:_{\mathrm{q}} \texttt{lost}} \qquad \frac{}{q <:_{\mathrm{q}} \texttt{top}} \qquad \frac{}{q <:_{\mathrm{q}} q}$$

Subclassing is standard:

$\boxed{C \sqsubseteq C'}$ subclassing

$$\frac{\texttt{class}\ Cid\ \texttt{extends}\ C'\ \texttt{\{ \_ \_ \}} \in Prg}{Cid \sqsubseteq C'} \qquad \frac{\texttt{class}\ C \ldots \in Prg}{C \sqsubseteq C}$$

$$\frac{C \sqsubseteq C_1 \qquad C_1 \sqsubseteq C'}{C \sqsubseteq C'}$$

Subtyping combines these two and adds a special case for primitives:

$\boxed{T <:\ T'}$ subtyping

$$\frac{q <:_{\mathrm{q}} q' \qquad C \sqsubseteq C'}{q\ C <:\ q'\ C'} \qquad \frac{q <:_{\mathrm{q}} q'}{q\ P <:\ q'\ P} \qquad \frac{}{\texttt{precise}\ P <:\ \texttt{approx}\ P}$$

We use the method ordering to express that we can replace a call of the sub-method by a call to the super-method, i.e. for our static method binding:

$\boxed{ms <:\ ms'}$ invocations of method ms can safely be replaced by calls to ms'

$$\frac{T' <: T \qquad \overline{T_k}^k <: \overline{T'_k}^k}{T\ m(\overline{T_k\ pid}^k)\ \texttt{precise} <:\ T'\ m(\overline{T'_k\ pid}^k)\ \texttt{approx}}$$

A.1.2 *Adaptation*

The `context` qualifier depends on the context and we need to adapt it, when the receiver changes, i.e. for field accesses and method calls.

We need to be careful and decide whether we can represent the new qualifier. If not, we use `lost`.

$\boxed{q \rhd q' = q''}$ combining two precision qualifiers

$$\frac{q'{=}\texttt{context} \wedge (q \in \{\texttt{approx}, \texttt{precise}, \texttt{context}\})}{q \rhd q' = q}$$

$$\frac{q'{=}\texttt{context} \wedge (q \in \{\texttt{top}, \texttt{lost}\})}{q \rhd q' = \texttt{lost}}$$

$$\frac{q' \neq \texttt{context}}{q \rhd q' = q'}$$

To combine whole types, we adapt the qualifiers:

$\boxed{q \rhd T = T'}$ precision qualifier - type combination

$$\frac{q \rhd q' = q''}{q \rhd q'\ C = q''\ C} \qquad \frac{q \rhd q' = q''}{q \rhd q'\ P = q''\ P}$$

The same logic follows for methods:

$\boxed{q \rhd ms = ms'}$ precision qualifier - method signature combination

$$\frac{q \rhd T = T' \qquad q \rhd \overline{T_k}^k = \overline{T'_k}^k}{q \rhd T\ m(\overline{T_k\ pid}^k)\ q' = T'\ m(\overline{T'_k\ pid}^k)\ q'}$$

A.1.3 *Look-up Functions*

The declared type of a field can be looked-up in the class declaration:

$\boxed{\text{FType}(C,f) = T}$ look up field f in class C

$$\frac{\texttt{class}\ Cid\ \texttt{extends}\ \_\ \{\ \_\ T\ f\ ;\ \_\ \_\ \}\ \in Prg}{\text{FType}(Cid,f) = T}$$

For a qualified class type, we also need to adapt the type:

$\boxed{\text{FType}(qC,f) = T}$ look up field f in reference type qC

$$\frac{\text{FType}(C,f) = T_1 \qquad q \rhd T_1 = T}{\text{FType}(q\ C,f) = T}$$

Note that subsumption in the type rule will be used to get to the correct class that declares the field. Methods work similarly.

$\boxed{\mathrm{MSig}(C, m, q) = ms}$ look up signature of method m in class C

$$\frac{\texttt{class}\ Cid\ \texttt{extends}\ \_\ \{\ \_\ \_\ ms\ \{\ e\ \}\ \_\ \}\in Prg \qquad \mathrm{MName}(ms)=m\ \wedge\ \mathrm{MQual}(ms)=q}{\mathrm{MSig}(Cid, m, q) = ms}$$

$\boxed{\mathrm{MSig}(qC, m) = ms}$ look up signature of method m in reference type qC

$$\frac{\mathrm{MSig}(C, m, q) = ms \qquad q \rhd ms = ms'}{\mathrm{MSig}(q\,C, m) = ms'}$$

A.1.4 *Well-formedness*

A well-formed expression:

$\boxed{{}^{s}\Gamma \vdash e : T}$ expression typing

$$\frac{{}^{s}\Gamma \vdash e : T_1 \qquad T_1 <: T}{{}^{s}\Gamma \vdash e : T} \qquad \frac{qC\ \mathsf{OK}}{{}^{s}\Gamma \vdash \texttt{null} : qC} \qquad \frac{}{{}^{s}\Gamma \vdash \mathcal{L} : \texttt{precise}\ P} \qquad \frac{{}^{s}\Gamma(x) = T}{{}^{s}\Gamma \vdash x : T}$$

$$\frac{q\,C\ \mathsf{OK} \qquad q \in \{\texttt{precise}, \texttt{approx}, \texttt{context}\}}{{}^{s}\Gamma \vdash \texttt{new}\ q\,C\texttt{()} : T} \qquad \frac{{}^{s}\Gamma \vdash e_0 : q\,C \qquad \mathrm{FType}(q\,C, f) = T}{{}^{s}\Gamma \vdash e_0.f : T}$$

$$\frac{{}^{s}\Gamma \vdash e_0 : q\,C \qquad \mathrm{FType}(q\,C, f) = T \qquad \texttt{lost} \notin T \qquad {}^{s}\Gamma \vdash e_1 : T}{{}^{s}\Gamma \vdash e_0.f\ :=\ e_1 : T}$$

$$\frac{{}^{s}\Gamma \vdash e_0 : q\,C \qquad q \in \{\texttt{precise}, \texttt{context}, \texttt{top}\} \qquad \mathrm{MSig}(\texttt{precise}\ C, m) = T\ m(\overline{T_i\ pid}^{\,i})\ \texttt{precise} \qquad \texttt{lost} \notin \overline{T_i}^{\,i} \qquad {}^{s}\Gamma \vdash \overline{e_i}^{\,i} : \overline{T_i}^{\,i}}{{}^{s}\Gamma \vdash e_0.m(\overline{e_i}^{\,i}) : T}$$

$$\frac{{}^{s}\Gamma \vdash e_0 : \texttt{approx}\ C \qquad \mathrm{MSig}(\texttt{approx}\ C, m) = T\ m(\overline{T_i\ pid}^{\,i})\ \texttt{approx} \qquad \texttt{lost} \notin \overline{T_i}^{\,i} \qquad {}^{s}\Gamma \vdash \overline{e_i}^{\,i} : \overline{T_i}^{\,i}}{{}^{s}\Gamma \vdash e_0.m(\overline{e_i}^{\,i}) : T}$$

$$\frac{{}^{s}\Gamma \vdash e_0 : \texttt{approx}\ C \qquad \mathrm{MSig}(\texttt{approx}\ C, m) = \mathit{None} \qquad \mathrm{MSig}(\texttt{precise}\ C, m) = T\ m(\overline{T_i\ pid}^{\,i})\ \texttt{precise} \qquad \texttt{lost} \notin \overline{T_i}^{\,i} \qquad {}^{s}\Gamma \vdash \overline{e_i}^{\,i} : \overline{T_i}^{\,i}}{{}^{s}\Gamma \vdash e_0.m(\overline{e_i}^{\,i}) : T}$$

$$\frac{{}^{s}\Gamma \vdash e : \_ \qquad q\,C\ \mathsf{OK}}{{}^{s}\Gamma \vdash (q\,C)\ e : T}$$

$$\frac{{}^{s}\Gamma \vdash e_0 : q\,P \qquad {}^{s}\Gamma \vdash e_1 : q\,P}{{}^{s}\Gamma \vdash e_0 \oplus e_1 : q\,P} \qquad \frac{{}^{s}\Gamma \vdash e_0 : \texttt{precise}\ P \qquad {}^{s}\Gamma \vdash e_1 : T \qquad {}^{s}\Gamma \vdash e_2 : T}{{}^{s}\Gamma \vdash \texttt{if}(e_0)\ \{e_1\}\ \texttt{else}\ \{e_2\} : T}$$

Note how `lost` is used to forbid invalid field updates and method calls.
Well-formed types:

$\boxed{T\ \mathsf{OK}}$ well-formed type

$$\frac{\texttt{class}\ C \ldots \in Prg}{q\,C\ \mathsf{OK}} \qquad \frac{}{q\,P\ \mathsf{OK}}$$

Well-formed classes just propagate the checks and ensure the superclass is valid:

$\boxed{Cls\ \mathsf{OK}}$ well-formed class declaration

$$\frac{{}^{s}\Gamma = \{\texttt{this} \mapsto \texttt{context}\ Cid\} \quad {}^{s}\Gamma \vdash \overline{fd}\ \mathsf{OK} \quad {}^{s}\Gamma, Cid \vdash \overline{md}\ \mathsf{OK} \quad \texttt{class}\ C \ldots \in Prg}{\texttt{class}\ Cid\ \texttt{extends}\ C\ \{\ \overline{fd}\ \overline{md}\ \}\ \mathsf{OK}} \qquad \frac{}{\texttt{class Object \{\}}\ \mathsf{OK}}$$

Fields just check their types:

$\boxed{{}^{s}\Gamma \vdash T\,f;\ \mathsf{OK}}$ well-formed field declaration

$$\frac{T\ \mathsf{OK}}{{}^{s}\Gamma \vdash T\,f;\ \mathsf{OK}}$$

Methods check their type, the body expression, overriding, and the method qualifier:

$\boxed{{}^{s}\Gamma, C \vdash md\ \mathsf{OK}}$ well-formed method declaration

$$\frac{\begin{array}{l}{}^{s}\Gamma = \{\texttt{this} \mapsto \texttt{context}\ C\} \\ {}^{s}\Gamma' = \left\{\texttt{this} \mapsto \texttt{context}\ C,\ \overline{pid \mapsto T_i}^{\,i}\right\} \\ T,\ \overline{T_i}^{\,i}\ \mathsf{OK} \qquad {}^{s}\Gamma' \vdash e : T \qquad C \vdash m\ \mathsf{OK} \\ q \in \{\texttt{precise}, \texttt{approx}\}\end{array}}{{}^{s}\Gamma, C \vdash T\ m(\overline{T_i\ pid}^{\,i})\ q\ \{\ e\ \}\ \mathsf{OK}}$$

Overriding checks for all supertypes C' that a helper judgment holds:

$\boxed{C \vdash m\ \mathsf{OK}}$ method overriding OK

$$\frac{C \sqsubseteq C' \Longrightarrow C, C' \vdash m\ \mathsf{OK}}{C \vdash m\ \mathsf{OK}}$$

This helper judgment ensures that if both methods are of the same precision, the signatures are equal. For a precise method we allow an approximate version that has relaxed types:

$\boxed{C, C' \vdash m\ \mathsf{OK}}$ method overriding OK auxiliary

$$\frac{\begin{array}{l}\mathrm{MSig}(C, m, \texttt{precise}) = ms_0 \ \wedge\ \mathrm{MSig}(C', m, \texttt{precise}) = ms_0' \ \wedge\ (ms_0' = None \ \vee\ ms_0 = ms_0') \\ \mathrm{MSig}(C, m, \texttt{approx}) = ms_1 \ \wedge\ \mathrm{MSig}(C', m, \texttt{approx}) = ms_1' \ \wedge\ (ms_1' = None \ \vee\ ms_1 = ms_1') \\ \mathrm{MSig}(C, m, \texttt{precise}) = ms_2 \ \wedge\ \mathrm{MSig}(C', m, \texttt{approx}) = ms_2' \ \wedge\ (ms_2' = None \ \vee\ ms_2 <: ms_2')\end{array}}{C, C' \vdash m\ \mathsf{OK}}$$

An environment simply checks all types:

$\boxed{{}^{s}\Gamma\ \mathsf{OK}}$ well-formed static environment

$$\frac{{}^{s}\Gamma = \left\{\texttt{this} \mapsto q\ C,\ \overline{pid \mapsto T_i}^{\,i}\right\} \quad q\ C,\ \overline{T_i}^{\,i}\ \mathsf{OK}}{{}^{s}\Gamma\ \mathsf{OK}}$$

Finally, a program checks the contained classes, the main expression and type, and ensures that the subtyping hierarchy is acyclic:

$\boxed{\vdash Prg\ \mathsf{OK}}$ well-formed program

$$\frac{\begin{array}{l} Prg{=}\overline{Cls_i}^{\,i},\ C,\ e \\ \quad \overline{Cls_i\ \mathsf{OK}}^{\,i} \qquad\qquad \mathtt{context}\ C\ \mathsf{OK} \\ \{\mathtt{this} \mapsto \mathtt{context}\ C\} \vdash e : \_ \\ \forall C', C''.\ ((C' \sqsubseteq C'' \ \wedge\ C'' \sqsubseteq C') \Longrightarrow C'{=}C'') \end{array}}{\vdash Prg\ \mathsf{OK}}$$

A.2 RUNTIME SYSTEM

A.2.1 *Helper Functions*

$\boxed{h + o\ =\ (h', \iota)}$ add object o to heap h resulting in heap h' and fresh address ι

$$\frac{\iota \notin \mathrm{dom}(h) \qquad h'{=}h \oplus (\iota \mapsto o)}{h + o\ =\ (h', \iota)}$$

$\boxed{h[\iota.f\ :=\ v]\ {=}h'}$ field update in heap

$$\frac{\begin{array}{l} v{=}\mathtt{null}_a\ \vee\ (v{=}\iota'\ \wedge\ \iota' \in \mathrm{dom}(h)) \\ h(\iota) = \left(T, \overline{fv}\right) \qquad f \in \mathrm{dom}\left(\overline{fv}\right) \qquad \overline{fv}'{=}\overline{fv}[f \mapsto v] \\ h'{=}h \oplus \left(\iota \mapsto \left(T, \overline{fv}'\right)\right) \end{array}}{h[\iota.f\ :=\ v]\ {=}h'}$$

$$\frac{\begin{array}{rl} h(\iota) = \left(T, \overline{fv}\right) & \overline{fv}(f) = (q', {}^r\mathcal{L}') \\ \overline{fv}'{=}\overline{fv}[f \mapsto (q', {}^r\mathcal{L})] & h'{=}h \oplus \left(\iota \mapsto \left(T, \overline{fv}'\right)\right) \end{array}}{h[\iota.f\ :=\ (q, {}^r\mathcal{L})]\ {=}h'}$$

A.2.2 *Runtime Typing*

In the runtime system we only have `precise` and `approx`. The `context` qualifier is substituted by the correct concrete qualifiers. The `top` and `lost` qualifiers are not needed at runtime.

This function replaces `context` qualifier by the correct qualifier from the environment:

$\boxed{\mathrm{sTrT}(h, \iota, T)\ {=}T'}$ convert type T to its runtime equivalent T'

$$\frac{q{=}\mathtt{context} \Longrightarrow q'{=}\mathrm{TQual}(h(\iota)\!\downarrow_1) \qquad q{\neq}\mathtt{context} \Longrightarrow q'{=}q}{\mathrm{sTrT}(h, \iota, q\ C)\ {=}q'\ C} \qquad \frac{q{=}\mathtt{context} \Longrightarrow q'{=}\mathrm{TQual}(h(\iota)\!\downarrow_1) \qquad q{\neq}\mathtt{context} \Longrightarrow q'{=}q}{\mathrm{sTrT}(h, \iota, q\ P)\ {=}q'\ P}$$

We can assign a type to a value, relative to a current object ι. For a reference type, we look up the concrete type in the heap, determine the runtime representation of the static type, and ensure that the latter is a subtype of the former. The null value can be assigned an arbitrary type. And for primitive values we ensure that the runtime version of the static type is a supertype of the concrete type.

$\boxed{h, \iota \vdash v : T}$ type T assignable to value v

$$\frac{\mathrm{sTrT}(h, \iota_0, q\ C) = q'\ C \qquad h(\iota)\downarrow_1 = T_1 \qquad T_1 <: q'\ C}{h, \iota_0 \vdash \iota : q\ C} \qquad \frac{}{h, \iota_0 \vdash \mathtt{null}_a : q\ C} \qquad \frac{\mathrm{sTrT}(h, \iota_0, q'\ P) = q''\ P \qquad {}^r\mathcal{L} \in P \qquad q\ P <: q''\ P}{h, \iota_0 \vdash (q, {}^r\mathcal{L}) : q'\ P}$$

A.2.3 *Look-up Functions*

Look-up a field of an object at a given address. Note that subtyping allows us to go to the class that declares the field:

$\boxed{\mathrm{FType}(h, \iota, f) = T}$ look up type of field in heap

$$\frac{h, \iota \vdash \iota : q\ C \qquad \mathrm{FType}(q\ C, f) = T}{\mathrm{FType}(h, \iota, f) = T}$$

Look-up the method signature of a method at a given address. Subtyping again allows us to go to any one of the possible multiple definitions of the methods. In a well-formed class, all these methods are equal:

$\boxed{\mathrm{MSig}(h, \iota, m) = ms}$ look up method signature of method m at ι

$$\frac{h, \iota \vdash \iota : q\ C \qquad \mathrm{MSig}(q\ C, m) = ms}{\mathrm{MSig}(h, \iota, m) = ms}$$

For the method body, we need the most concrete implementation. This first function looks for a method with the given name and qualifier in the given class and in sequence in all super classes:

$\boxed{\mathrm{MBody}(C, m, q) = e}$ look up most-concrete body of m, q in class C or a superclass

$$\frac{\mathtt{class}\ Cid\ \mathtt{extends}\ \_\ \{\ \_\ \_\ ms\ \{\ e\ \}\ \_\ \} \in Prg \qquad \mathrm{MName}(ms) = m \ \wedge\ \mathrm{MQual}(ms) = q}{\mathrm{MBody}(Cid, m, q) = e}$$

$$\frac{\mathtt{class}\ Cid\ \mathtt{extends}\ C_1\ \{\ \_\ \overline{ms_n\ \{\ e_n\ \}}^{\,n}\ \} \in Prg \qquad \overline{\mathrm{MName}(ms_n) \neq m}^{\,n} \qquad \mathrm{MBody}(C_1, m, q) = e}{\mathrm{MBody}(Cid, m, q) = e}$$

To look up the most concrete implementation for a method at a given address, we have three cases to consider. If it's a precise method, look it up. If it's an approximate method, try to find an approximate method. If you are looking for an approximate method, but couldn't find one, try to look for a precise methods:

$\boxed{\mathrm{MBody}(h, \iota, m) = e}$ look up most-concrete body of method m at ι

$$\frac{h(\iota)\downarrow_1 = \mathtt{precise}\ C \qquad \mathrm{MBody}(C, m, \mathtt{precise}) = e}{\mathrm{MBody}(h, \iota, m) = e}$$

$$\frac{h(\iota)\downarrow_1 = \mathtt{approx}\ C \qquad \mathrm{MBody}(C, m, \mathtt{approx}) = e}{\mathrm{MBody}(h, \iota, m) = e}$$

$$\frac{h(\iota)\downarrow_1 = \mathtt{approx}\ C \qquad \mathrm{MBody}(C, m, \mathtt{approx}) = \mathit{None} \qquad \mathrm{MBody}(C, m, \mathtt{precise}) = e}{\mathrm{MBody}(h, \iota, m) = e}$$

Get the field values corresponding to a given reference type. For fields of reference type, just use the null value. For fields of a primitive type, we need to look up the declared type of the field in order to determine the correct qualifier for the value.

$\boxed{\text{FVsInit}(qC) = \overline{fv}}$ initialize the fields for reference type qC

$$\frac{q \in \{\texttt{precise}, \texttt{approx}\} \quad \forall f \in \text{refFields}(C) \,.\, \overline{fv}(f) = \texttt{null}_a \quad \forall f \in \text{primFields}(C) \,.\, (\text{FType}(q\,C, f) = q'\,P \wedge \overline{fv}(f) = (q', 0))}{\text{FVsInit}(q\,C) = \overline{fv}}$$

A.2.4 *Semantics*

The standard semantics of our programming language:

$\boxed{{}^r T \vdash h, e \rightsquigarrow h', v}$ big-step operational semantics

$$\frac{}{{}^r T \vdash h, \texttt{null} \rightsquigarrow h, \texttt{null}_a} \qquad \frac{}{{}^r T \vdash h, \mathcal{L} \rightsquigarrow h, (\texttt{precise}, {}^r\mathcal{L})} \qquad \frac{{}^r T(x) = v}{{}^r T \vdash h, x \rightsquigarrow h, v}$$

$$\frac{\text{sTrT}(h, {}^r T(\texttt{this}), q\,C) = q'\,C \quad \text{FVsInit}(q'\,C) = \overline{fv} \quad h + \left(q'\,C, \overline{fv}\right) = (h', \iota)}{{}^r T \vdash h, \texttt{new}\, q\, C\texttt{()} \rightsquigarrow h', \iota} \qquad \frac{{}^r T \vdash h, e_0 \rightsquigarrow h', \iota_0 \quad h'(\iota_0.f) = v}{{}^r T \vdash h, e_0.f \rightsquigarrow h', v}$$

$$\frac{{}^r T \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \quad {}^r T \vdash h_0, e_1 \rightsquigarrow h_1, v \quad h_1[\iota_0.f := v] = h'}{{}^r T \vdash h, e_0.f := e_1 \rightsquigarrow h', v}$$

$$\frac{{}^r T \vdash h, e_0 \rightsquigarrow h_0, \iota_0 \quad {}^r T \vdash h_0, \overline{e_i}^{\,i} \rightsquigarrow h_1, \overline{v_i}^{\,i} \quad \text{MBody}(h_0, \iota_0, m) = e \quad \text{MSig}(h_0, \iota_0, m) = \_\, m(\overline{\_\, pid}^{\,i})\, q \quad {}^r T' = \left\{\texttt{precise}; \texttt{this} \mapsto \iota_0, \overline{pid \mapsto v_i}^{\,i}\right\} \quad {}^r T' \vdash h_1, e \rightsquigarrow h', v}{{}^r T \vdash h, e_0.m(\overline{e_i}^{\,i}) \rightsquigarrow h', v}$$

$$\frac{{}^r T \vdash h, e \rightsquigarrow h', v \quad h', {}^r T(\texttt{this}) \vdash v : q\,C}{{}^r T \vdash h, (q\,C)\, e \rightsquigarrow h', v} \qquad \frac{{}^r T \vdash h, e_0 \rightsquigarrow h_0, (q, {}^r\mathcal{L}_0) \quad {}^r T \vdash h_0, e_1 \rightsquigarrow h', (q, {}^r\mathcal{L}_1)}{{}^r T \vdash h, e_0 \oplus e_1 \rightsquigarrow h', (q, {}^r\mathcal{L}_0 \oplus {}^r\mathcal{L}_1)}$$

$$\frac{{}^r T \vdash h, e_0 \rightsquigarrow h_0, (q, {}^r\mathcal{L}) \quad {}^r\mathcal{L} \neq 0 \quad {}^r T \vdash h_0, e_1 \rightsquigarrow h', v}{{}^r T \vdash h, \texttt{if}(e_0)\,\texttt{\{}e_1\texttt{\}}\,\texttt{else}\,\texttt{\{}e_2\texttt{\}} \rightsquigarrow h', v} \qquad \frac{{}^r T \vdash h, e_0 \rightsquigarrow h_0, (q, 0) \quad {}^r T \vdash h_0, e_2 \rightsquigarrow h', v}{{}^r T \vdash h, \texttt{if}(e_0)\,\texttt{\{}e_1\texttt{\}}\,\texttt{else}\,\texttt{\{}e_2\texttt{\}} \rightsquigarrow h', v}$$

$$\frac{{}^r T \vdash h, e \rightsquigarrow h', v \quad h' \cong \tilde{h}' \quad v \cong \tilde{v}}{{}^r T \vdash h, e \rightsquigarrow \tilde{h}', \tilde{v}}$$

A program is executed by instantiating the main class and then evaluating the main expression in a suitable heap and environment:

$\boxed{\vdash Prg \rightsquigarrow h, v}$ big-step operational semantics of a program

$$\frac{\begin{array}{l}\text{FVsInit}(\texttt{precise}\,C) = \overline{fv} \\ \emptyset + \left(\texttt{precise}\,C, \overline{fv}\right) = (h_0, \iota_0) \\ {}^{r}T_0 = \{\texttt{precise}; \texttt{this} \mapsto \iota_0\} \qquad {}^{r}T_0 \vdash h_0, e \rightsquigarrow h, v\end{array}}{\vdash \overline{Cls},\ C,\ e \rightsquigarrow h, v}$$

We provide a checked version of the semantics that ensures that we do not have an interference between approximate and precise parts:

$\boxed{{}^{r}T \vdash h, e \rightsquigarrow_c h', v}$ checked big-step operational semantics

$$\frac{{}^{r}T \vdash h, \texttt{null} \rightsquigarrow h, \texttt{null}_a}{{}^{r}T \vdash h, \texttt{null} \rightsquigarrow_c h, \texttt{null}_a} \qquad \frac{{}^{r}T \vdash h, \mathcal{L} \rightsquigarrow h, (\texttt{precise}, {}^{r}\mathcal{L})}{{}^{r}T \vdash h, \mathcal{L} \rightsquigarrow_c h, (\texttt{precise}, {}^{r}\mathcal{L})}$$

$$\frac{{}^{r}T \vdash h, x \rightsquigarrow h, v}{{}^{r}T \vdash h, x \rightsquigarrow_c h, v} \qquad \frac{{}^{r}T \vdash h, \texttt{new}\,q\,C\texttt{()} \rightsquigarrow h', \iota}{{}^{r}T \vdash h, \texttt{new}\,q\,C\texttt{()} \rightsquigarrow_c h', \iota}$$

$$\frac{\begin{array}{l}{}^{r}T \vdash h, e_0 \rightsquigarrow_c h', \iota_0 \\ {}^{r}T \vdash h, e_0.f \rightsquigarrow h', v\end{array}}{{}^{r}T \vdash h, e_0.f \rightsquigarrow_c h', v} \qquad \frac{\begin{array}{l}{}^{r}T \vdash h, e_0 \rightsquigarrow_c h_0, \iota_0 \qquad h(\iota_0)\downarrow_1 = q\,C \\ {}^{r}T\downarrow_1 = q' \qquad (q = q' \ \vee\ q' = \texttt{precise}) \\ {}^{r}T \vdash h_0, e_1 \rightsquigarrow_c h_1, v \\ {}^{r}T \vdash h, e_0.f := e_1 \rightsquigarrow h', v\end{array}}{{}^{r}T \vdash h, e_0.f := e_1 \rightsquigarrow_c h', v}$$

$$\frac{\begin{array}{l}{}^{r}T \vdash h, e_0 \rightsquigarrow_c h_0, \iota_0 \qquad {}^{r}T \vdash h_0, \overline{e_i}^{\,i} \rightsquigarrow_c h_1, \overline{v_i}^{\,i} \\ \text{MBody}(h_0, \iota_0, m) = e \qquad \text{MSig}(h_0, \iota_0, m) = \_\,m(\overline{\_\,pid}^{\,i})\,q \\ {}^{r}T' = \left\{\texttt{precise}; \texttt{this} \mapsto \iota_0,\ \overline{pid \mapsto v_i}^{\,i}\right\} \\ {}^{r}T' \vdash h_1, e \rightsquigarrow_c h', v \\ {}^{r}T \vdash h, e_0.m(\overline{e_i}^{\,i}) \rightsquigarrow h', v\end{array}}{{}^{r}T \vdash h, e_0.m(\overline{e_i}^{\,i}) \rightsquigarrow_c h', v}$$

$$\frac{\begin{array}{l}{}^{r}T \vdash h, e \rightsquigarrow_c h', v \\ {}^{r}T \vdash h, (q\,C)\ e \rightsquigarrow h', v\end{array}}{{}^{r}T \vdash h, (q\,C)\ e \rightsquigarrow_c h', v} \qquad \frac{\begin{array}{l}{}^{r}T \vdash h, e_0 \rightsquigarrow_c h_0, (q, {}^{r}\mathcal{L}_0) \\ {}^{r}T \vdash h_0, e_1 \rightsquigarrow_c h', (q, {}^{r}\mathcal{L}_1) \\ {}^{r}T \vdash h, e_0 \oplus e_1 \rightsquigarrow h', (q, {}^{r}\mathcal{L}_0 \oplus {}^{r}\mathcal{L}_1)\end{array}}{{}^{r}T \vdash h, e_0 \oplus e_1 \rightsquigarrow_c h', (q, {}^{r}\mathcal{L}_0 \oplus {}^{r}\mathcal{L}_1)}$$

$$\frac{\begin{array}{l}{}^{r}T \vdash h, e_0 \rightsquigarrow_c h_0, (q, {}^{r}\mathcal{L}) \qquad {}^{r}\mathcal{L} \neq 0 \\ {}^{r}T' = {}^{r}T(q) \qquad {}^{r}T' \vdash h_0, e_1 \rightsquigarrow_c h', v \\ {}^{r}T \vdash h, \texttt{if}(e_0)\,\{e_1\}\,\texttt{else}\,\{e_2\} \rightsquigarrow h', v\end{array}}{{}^{r}T \vdash h, \texttt{if}(e_0)\,\{e_1\}\,\texttt{else}\,\{e_2\} \rightsquigarrow_c h', v}$$

$$\frac{\begin{array}{l}{}^{r}T \vdash h, e_0 \rightsquigarrow_c h_0, (q, {}^{r}\mathcal{L}) \qquad {}^{r}\mathcal{L} = 0 \\ {}^{r}T' = {}^{r}T(q) \qquad {}^{r}T' \vdash h_0, e_2 \rightsquigarrow_c h', v \\ {}^{r}T \vdash h, \texttt{if}(e_0)\,\{e_1\}\,\texttt{else}\,\{e_2\} \rightsquigarrow h', v\end{array}}{{}^{r}T \vdash h, \texttt{if}(e_0)\,\{e_1\}\,\texttt{else}\,\{e_2\} \rightsquigarrow_c h', v}$$

A.2.5 *Well-formedness*

A heap is well formed if all field values are correctly typed and all types are valid:

$\boxed{h\ \mathsf{OK}}$ well-formed heap

$$\frac{\begin{array}{l}\forall \iota \in \text{dom}(h), f \in h(\iota)\downarrow_2 . (\text{FType}(h, \iota, f) = T \ \wedge\ h, \iota \vdash h(\iota.f) : T) \\ \forall \iota \in \text{dom}(h) . (h(\iota)\downarrow_1\ \mathsf{OK} \ \wedge\ \text{TQual}(h(\iota)\downarrow_1) \in \{\texttt{precise}, \texttt{approx}\})\end{array}}{h\ \mathsf{OK}}$$

This final judgment ensures that the heap and runtime environment correspond to a static environment. It makes sure that all pieces match up:

$$\frac{\begin{array}{l}{}^{r}T = \left\{\texttt{precise}; \texttt{this} \mapsto \iota, \overline{pid \mapsto v_i}^{\,i}\right\} \\ {}^{s}T = \left\{\texttt{this} \mapsto \texttt{context}\, C, \overline{pid \mapsto T_i}^{\,i}\right\} \\ \quad h\ \mathsf{OK} \qquad\qquad {}^{s}T\ \mathsf{OK} \\ h, \iota \vdash \iota : \texttt{context}\, C \\ h, \iota \vdash \overline{v_i}^{\,i} : \overline{T_i}^{\,i}\end{array}}{\boxed{h, {}^{r}T : {}^{s}T\ \mathsf{OK}}}\ \text{runtime and static environments correspond}$$

A.3 PROOFS

The principal goal of formalizing EnerJ is to prove a *non-interference* property (Theorem 4). The other properties listed in this section support that proof.

A.3.1 *Type Safety*

Theorem 2 (Type Safety)**.**

$$\left.\begin{array}{ll} 1. & \vdash Prg\ \ OK \\ 2. & h, {}^{r}T : {}^{s}T\ \ OK \\ 3. & {}^{s}T \vdash e : T \\ 4. & {}^{r}T \vdash h, e \rightsquigarrow h', v \end{array}\right\} \Longrightarrow \left\{\begin{array}{ll} I. & h', {}^{r}T : {}^{s}T\ \ OK \\ II. & h', {}^{r}T(\texttt{this}) \vdash v : T \end{array}\right.$$

We prove this by rule induction on the operational semantics.

Case 1: e**=**`null`

The heap is not modified so I. trivially holds.

The null literal statically gets assigned an arbitrary reference type. The null value can be assigned an arbitrary reference type.

Case 2: $e = \mathcal{L}$

The heap is not modified so I. trivially holds.

A primitive literal statically gets assigned type precise or a supertype. The evaluation of a literal gives a precise value which can be assigned any primitive type.

Case 3: $e = x$

The heap is not modified so I. trivially holds.

We know that 2. that the environments correspond and therefore that the static type of the variable can be assigned to the value of the variable.

Case 4: e**=**`new` $q C$`()`

For I. we only have to show that the newly created object is valid. The initialization with the null or zero values ensures that all fields are correctly typed.

The type of the new object is the result of sTrT on the static type.

Case 5: $e = e_0.f$

The heap is not modified so $I.$ trivially holds.

We know from 2. that the heap is well formed. In particular, we know that the values stored for fields are subtypes of the field types.

We perform induction on e_0 and then use Lemma 1 to adapt the declared field, which is checked by the well-formed heap, to the adapted field type T.

Case 6: $e{=}e_0.f := e_1$

We perform induction on e_0 and e_1. We know from 3. that the static type of e_1 is a subtype of the adapted field type. We use Lemma 2 to adapt the type to the declaring class to re-establish that the heap is well formed.

Case 7: $e{=}e_0.m(\overline{e})$

A combination of cases 6 and 7.

Case 8: $e{=}(q\,C)\ e$

By induction we know that the heap is still well formed.

4. performs a runtime check to ensure that the value has the correct type.

Case 9: $e{=}e_0 \oplus e_1$

By induction we know that the heap is still well formed.

The type matches trivially.

Case 10: $e{=}\texttt{if}(e_0)\ \texttt{\{}e_1\texttt{\}}\ \texttt{else}\ \texttt{\{}e_2\texttt{\}}$

By induction we know that the heap is still well formed.

The type matches by induction. □

A.3.2 *Equivalence of Checked Semantics*

We prove that an execution under the unchecked operational semantics has an equivalent execution under the checked semantics.

Theorem 3 (Equivalence of Checked Semantics)**.**

$$\left.\begin{array}{ll} 1. & \vdash Prg\ \ OK \\ 2. & h, {}^{r}T : {}^{s}T\ \ OK \\ 3. & {}^{s}T \vdash e : T \\ 4. & {}^{r}T \vdash h, e \rightsquigarrow h', v \end{array}\right\} \Longrightarrow\ I.\quad {}^{r}T \vdash h, e \rightsquigarrow_c h', v$$

We prove this by rule induction on the operational semantics.

The checked operational semantics is only different from the unchecked semantics for the field write, method call, and conditional cases. The other cases trivially hold.

Case 1: $e{=}\texttt{if}(e_0)\ \texttt{\{}e_1\texttt{\}}\ \texttt{else}\ \texttt{\{}e_2\texttt{\}}$

We know from 3. that the static type of the condition is always precise. Therefore, ${}^{r}T'$ is well formed and we can apply the induction hypothesis on e_1 and e_2.

Case 2: $e{=}e_0.m(\overline{e})$

From the proof of type safety we know that the values in ${}^{r}T'$ are well formed. We are using `precise` as the approximate environment. Therefore, ${}^{r}T'$ is well formed and we can apply the induction hypothesis on e.

Case 3: e=$e_0.f := e_1$

We know from 2. that q'=`precise`. Therefore, the additional check passes. □

A.3.3 *Non-Interference*

To express a non-interference property, we first define a relation $\cong$ on values, heaps, and environments. Intuitively, $\cong$ denotes an equality that disregards approximate values. The relation holds only for values, heaps, and environments with identical types.

Where v and $\tilde{v}$ are primitive values, $v \cong \tilde{v}$ iff the values have the same type qP and either $q =$ `approx` or $v = \tilde{v}$. For objects, $\iota \cong \tilde{\iota}$ iff $\iota = \tilde{\iota}$. For heaps, $h \cong \tilde{h}$ iff the two heaps contain the same set of addresses ι and, for each such ι and each respective field f, $h(\iota.f) \cong \tilde{h}(\iota.f)$. Similarly, for environments, ${}^{r}T \cong {}^{\tilde{r}}\tilde{T}$ iff ${}^{r}T(\texttt{this}) \cong {}^{\tilde{r}}\tilde{T}(\texttt{this})$ and, for every parameter identifier pid, ${}^{r}T(pid) \cong {}^{\tilde{r}}\tilde{T}(pid)$.

We can now state our desired non-interference property.

Theorem 4 (Non-Interference)**.**

$$\left.\begin{array}{ll} 1. & \vdash Prg\ \texttt{OK} \wedge \vdash h, {}^{r}T : {}^{s}T \\ 2. & {}^{s}T \vdash e : T \\ 3. & {}^{r}T \vdash h, e \rightsquigarrow h', v \\ 4. & h \cong \tilde{h} \wedge {}^{r}T \cong {}^{\tilde{r}}\tilde{T} \\ 5. & \vdash \tilde{h}, {}^{\tilde{r}}\tilde{T} : {}^{s}T \end{array}\right\} \Longrightarrow \left\{\begin{array}{lll} I. & {}^{\tilde{r}}\tilde{T} \vdash \tilde{h}, e \rightarrow \tilde{h}', \tilde{v} \\ II. & h' \cong \tilde{h}' \\ III. & v \cong \tilde{v} \end{array}\right.$$

The non-interference property follows from the definition of the checked semantics, which are shown to hold in Theorem 3 given premises 1, 2, and 3. That is, via Theorem 3, we know that ${}^{r}T \vdash h, e \rightsquigarrow_c h', v$. The proof proceeds by rule induction on the *checked* semantics.

Case 1: e=`null`

The heap is unmodified, so $h = h'$ and $\tilde{h}' = \tilde{h}$. Because $h \cong \tilde{h}$, trivially $h' \cong \tilde{h}'$ (satisfying II.).

Both $v =$ `null` and $\tilde{v} =$ `null`, so III. also holds.

Case 2: e=$\mathcal{L}$

As above, the heap is unmodified and $v = \tilde{v}$ because literals are assigned precise types.

Case 3: e=x

Again, the heap is unmodified. If x has precise type, then $v = \tilde{v}$ and III. holds. Otherwise, both v and $\tilde{v}$ have approximate type so $v \cong \tilde{v}$ vacuously. (That is, $v \cong \tilde{v}$ holds for any such pair of values when their type is approximate.)

Case 4: e=`new` qC`()`

In this case, a new object o is created with address v and $h' = h \oplus (v \mapsto o)$. Because v has a reference type and $\tilde{v}$ has the same type, $v \cong \tilde{v}$. Furthermore, $\tilde{h'} = h \oplus (\tilde{v} \mapsto o)$, so $h \cong \tilde{h}$.

Case 5: e=$e_0.f$

The heap is unmodified in field lookup, so $II.$ holds by induction. Also by induction, e_0 resolves to the same address ι under h as under $\tilde{h}$ due to premise 4. If $h(\iota.f)$ has approximate type, then $III.$ holds vacuously; otherwise $v = \tilde{v}$.

Case 6: e=$e_0.f := e_1$

Apply induction to both subexpressions (e_0 and e_1). Under either heap h or $\tilde{h}$, the first expression e_0 resolves to the same object o. By type safety, e_1 resolves to a value with a dynamic type compatible with the static type of o's field f.

If the value is approximate, then the field must have approximate type and the conclusions hold vacuously. If the value is precise, then induction implies that the value produced by e_1 must be $v = \tilde{v}$, satisfying $III.$ Similarly, the heap update to h is identical to the one to $\tilde{h}$, so $\tilde{h} \cong \tilde{h'}$.

Case 7: e=$e_0.m(\bar{e})$

As in Case 5, let e_0 map to o in both h and $\tilde{h}$. The same method body is therefore looked up by MBody and, by induction on the evaluation of the method body, the conclusions all hold.

Case 8: e=$(qC)\ e$

Induction applies directly; the expression changes neither the output heap nor the value produced.

Case 9: e=$e_0 \oplus e_1$

The expression does not change the heap. If the type of $e_0 \oplus e_1$ is approximate, then $III.$ hold vacuously. If it is precise, then both $e0$ and $e1$ also have precise type, and, via induction, each expression produces the same literal under h and ${}^rT$ as under $\tilde{h}$ and ${}^r\tilde{T}$. Therefore, $v = \tilde{v}$, satisfying $III.$

Case 10: e=`if`(e_0) `{`e_1`} else {`e_2`}`

By type safety, e_0 resolves to a value with precise type. Therefore, by induction, the expression produces the same value under heap h and environment ${}^rT$ as under the equivalent structures $\tilde{h}$ and ${}^r\tilde{T}$. The rule applied for ${}^rT \vdash h, e \rightsquigarrow h', v$ (either COS_COND_T or COS_COND_F) also applies for ${}^r\tilde{T} \vdash \tilde{h}, e \rightarrow \tilde{h'}, \tilde{v}$ because the value in the condition is the same in either case. That is, either e_1 is evaluated in bot settings or else e_2 is; induction applies in either case. □

A.3.4 *Adaptation from a Viewpoint*

Lemma 1 (Adaptation from a Viewpoint)**.**

$$\left.\begin{array}{ll} 1. & h, \iota_0 \vdash \iota : q\ C \\ 2. & h, \iota \vdash v : T \end{array}\right\} \Longrightarrow \begin{array}{c} \exists T'.\ q \rhd T = T' \wedge \\ h, \iota_0 \vdash v : T' \end{array}$$

This lemma justifies the type rule and the method result in .

Case analysis of T:

Case 1: T**=**$q'\ C'$ **or** T**=**$q'\ P$ **where** $q' \in \{\texttt{precise}, \texttt{approx}, \texttt{top}\}$

In this case we have that $T'=T$ and the viewpoint is irrelevant.

Case 2: T**=**`context` C' **or** T**=**`context` P

Case 2a: $q \in \{\texttt{precise}, \texttt{approx}\}$

We have that $T'=q\ C'$ or $T'=q\ P$, respectively.

2. uses the precision of ι to substitute `context`. 1. gives us the type for ι. Together, they give us the type of v relative to ι_0.

Case 2b: $q \in \{\texttt{lost}, \texttt{top}\}$

We have that $T'=$`lost` C' or $T'=$`lost` P, respectively.

Such a T' is a valid type for any value. □

A.3.5 *Adaptation to a Viewpoint*

Lemma 2 (Adaptation to a Viewpoint)**.**

$$\left.\begin{array}{ll} 1. & h, \iota_0 \vdash \iota : q\ C \\ 2. & q \rhd T = T' \\ 3. & \texttt{lost} \notin T' \\ 4. & h, \iota_0 \vdash v : T' \end{array}\right\} \Longrightarrow\ h, \iota \vdash v : T$$

This lemma justifies the type rule and the requirements for the types of the parameters in .

Case analysis of T:

Case 1: T**=**$q'\ C'$ **or** T**=**$q'\ P$ **where** $q' \in \{\texttt{precise}, \texttt{approx}, \texttt{top}\}$

In this case we have that $T'=T$ and the viewpoint is irrelevant.

Case 2: T**=**`context` C' **or** T**=**`context` P

We have that $T'=q\ C'$ or $T'=q\ P$, respectively. 3. forbids lost from occurring.

1. gives us the precision for ι and 4. for v, both relative to ι_0. From 2. and 3. we get the conclusion. □

B

PROBABILITY TYPES: SOUNDNESS PROOF

This appendix expands on the formalism for DECAF, the probability-types language in Chapter 4. We present the full syntax, static semantics, and dynamic semantics for the core PROB language. We prove a *soundness* theorem that embodies the probability type system's fundamental accuracy guarantee. This appendix corresponds to the appendix for the main DECAF paper in OOPSLA 2015 [22].

B.1 SYNTAX

We formalize a core of PROB without inference. The syntax for statements, expressions, and types is:

$$
\begin{aligned}
s &\equiv T\, v := e \mid v := e \mid s\,;s \mid \textbf{if}\; e\; s\; s \mid \textbf{while}\; e\; s \mid \textbf{skip} \\
e &\equiv c \mid v \mid e \oplus_p e \mid \mathsf{endorse}(p, e) \mid \mathsf{check}(p, e) \mid \mathsf{track}(p, e) \\
\oplus &\equiv + \mid - \mid \times \mid \div \\
T &\equiv q\, \tau \\
q &\equiv \texttt{@Approx}(p) \mid \texttt{@Dyn} \\
\tau &\equiv \mathsf{int} \mid \mathsf{float} \\
v &\in \text{variables},\ c \in \text{constants},\ p \in [0.0, 1.0]
\end{aligned}
$$

For the purpose of the static and dynamic semantics, we also define values V, heaps H, dynamic probability maps D, true probability maps S, and static contexts Γ:

$$
\begin{aligned}
V &\equiv c \mid \Box \\
H &\equiv \cdot \mid H, v \mapsto V \\
D &\equiv \cdot \mid D, v \mapsto p \\
S &\equiv \cdot \mid S, v \mapsto p \\
\Gamma &\equiv \cdot \mid \Gamma, v \mapsto T
\end{aligned}
$$

We define $H(v)$, $D(v)$, $S(v)$, and $\Gamma(v)$ to denote variable lookup in these maps.

B.2 TYPING

The type system defines the static semantics for the core language. We first give typing judgments first for expressions and then for statements.

B.2.1 *Operator Typing*

We introduce a helper "function" that determines the unqualified result type of a binary arithmetic operator.

$\boxed{\text{optype}(\tau_1, \tau_2) = \tau_3}$

$$\text{optype}(\tau, \tau) = \tau \qquad \text{optype}(\mathsf{int}, \mathsf{float}) = \mathsf{float} \qquad \text{optype}(\mathsf{float}, \mathsf{int}) = \mathsf{float}$$

Now we can give the types of the binary operator expressions themselves. There are two cases: one for statically-typed operators and one for dynamic tracking. The operands may not mix static and dynamic qualifiers (recall that the compiler inserts track casts to introduce dynamic tracking when necessary).

$\boxed{\Gamma \vdash e : T}$

OP-STATIC-TYPES

$$\frac{\Gamma \vdash e_1 : \mathtt{@Approx}(p_1)\ \tau_1 \qquad \Gamma \vdash e_2 : \mathtt{@Approx}(p_2)\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2) \qquad p' = p_1 \cdot p_2 \cdot p_{\text{op}}}{\Gamma \vdash e_1 \oplus_{p_{\text{op}}} e_2 : \mathtt{@Approx}(p')\ \tau_3}$$

OP-DYN-TYPES

$$\frac{\Gamma \vdash e_1 : \mathtt{@Dyn}\ \tau_1 \qquad \Gamma \vdash e_2 : \mathtt{@Dyn}\ \tau_2 \qquad \tau_3 = \text{optype}(\tau_1, \tau_2)}{\Gamma \vdash e_1 \oplus_p e_2 : \mathtt{@Dyn}\ \tau_3}$$

In the static case, the output probability is the product of the probabilities for the left-hand operand, right-hand operand, and the operator itself. Section 4.3 gives the probabilistic intuition behind this rule.

B.2.2 *Other Expressions*

The rules for constants and variables are straightforward. Literals are given the precise ($p = 1.0$) type.

CONST-INT-TYPES

$$\frac{c \text{ is an integer}}{\Gamma \vdash c : \mathtt{@Approx}(1.0)\ \mathsf{int}}$$

CONST-FLOAT-TYPES

$$\frac{c \text{ is not an integer}}{\Gamma \vdash c : \mathtt{@Approx}(1.0)\ \mathsf{float}}$$

VAR-TYPES

$$\frac{T = \Gamma(v)}{\Gamma \vdash v : T}$$

Endorsements, both checked and unchecked, produce the explicitly requested type. (Note that check is sound but endorse is potentially unsound: our main soundness theorem, at the end of this appendix, will exclude the latter from the language.) Similarly, track casts produce a dynamically-tracked type given a statically-tracked counterpart.

ENDORSE-TYPES

$$\frac{\Gamma \vdash e : q\ \tau}{\Gamma \vdash \mathsf{endorse}(p, e) : \mathtt{@Approx}(p)\ \tau}$$

CHECK-TYPES

$$\frac{\Gamma \vdash e : \mathtt{@Dyn}\ \tau}{\Gamma \vdash \mathsf{check}(p, e) : \mathtt{@Approx}(p)\ \tau}$$

TRACK-TYPES

$$\frac{\Gamma \vdash e : \mathtt{@Approx}(p')\ \tau \qquad p \leq p'}{\Gamma \vdash \mathsf{track}(p, e) : \mathtt{@Dyn}\ \tau}$$

B.2.3 *Qualifiers and Subtyping*

A simple subtyping relation, introduced in Section 4.3, makes high-probability types subtypes of their low-probability counterparts.

$\boxed{T_1 \prec T_2}$

$$\textsc{subtyping}\ \frac{p \geq p'}{\texttt{@Approx}(p)\ \tau \prec \texttt{@Approx}(p')\ \tau}$$

Subtyping uses a standard subsumption rule.

$$\textsc{subsumption}\ \frac{T_1 \prec T_2 \qquad \Gamma \vdash e : T_1}{\Gamma \vdash e : T_2}$$

B.2.4 *Statement Typing*

Our typing judgment for statements builds up the context Γ.

$\boxed{\Gamma_1 \vdash s : \Gamma_2}$

$$\textsc{skip-types}\ \frac{}{\Gamma \vdash \mathsf{skip} : \Gamma} \qquad \textsc{seq-types}\ \frac{\Gamma_1 \vdash s_1 : \Gamma_2 \qquad \Gamma_2 \vdash s_2 : \Gamma_3}{\Gamma_1 \vdash s_1; s_2 : \Gamma_3} \qquad \textsc{decl-types}\ \frac{\Gamma \vdash e : T \qquad v \notin \Gamma}{\Gamma \vdash T\ v := e : \Gamma, v : T}$$

$$\textsc{mutate-types}\ \frac{\Gamma \vdash e : T \qquad \Gamma(v) = T}{\Gamma \vdash v := e : \Gamma}$$

$$\textsc{if-types}\ \frac{\Gamma \vdash e : \texttt{@Approx}(1.0)\ \tau \qquad \Gamma \vdash s_1 : \Gamma_1 \qquad \Gamma \vdash s_2 : \Gamma_2}{\Gamma \vdash \textbf{if}\ e\ s_1\ s_2 : \Gamma}$$

$$\textsc{while-types}\ \frac{\Gamma \vdash e : \texttt{@Approx}(1.0)\ \tau \qquad \Gamma \vdash s : \Gamma'}{\Gamma \vdash \textbf{while}\ e\ s : \Gamma}$$

The conditions in if and while statements are required to have the precise type ($p = 1.0$).

B.3 OPERATIONAL SEMANTICS

We use a large-step operational semantics for expressions and small-step semantics for statements. Both are nondeterministic: values produced by approximate operators can produce either an error value $\square$ or a concrete number.

B.3.1 *Expression Semantics*

There are two judgments for expressions: one for statically typed expressions and one where dynamic tracking is used. The former, $H; D; S; e \Downarrow_p V$, indi-

cates that the expression e produces a value V, which is either a constant c or the error value $\square$, and p is the probability that $V \neq \square$. The latter judgment, $H; D; S; e \Downarrow_p V, p_d$, models dynamically-tracked expression evaluation. In addition to a value V, it also produces a computed probability value p_d reflecting the compiler's conservative bound on the reliability of e's value. That is, p is the "true" probability that $V \neq \square$ whereas p_d is the dynamically computed conservative bound for p.

In these judgments, H is the heap mapping variables to values and D is the dynamic probability map for `@Dyn`-typed variables maintained by the compiler. The S probability map is used for our type soundness proof: it maintains the actual probability that a variable is correct.

CONSTANTS Literals are always tracked statically.

$$\textsc{const}\ \frac{}{H; D; S; c \Downarrow_{1.0} c}$$

VARIABLES Variable lookup is dynamically tracked when the variable is present in the tracking map D. The probability $S(v)$ is the chance that the variable does not hold $\square$.

$$\textsc{var}\ \frac{v \notin D}{H; D; S; v \Downarrow_{S(v)} H(v)} \qquad \textsc{var-dyn}\ \frac{v \in D}{H; D; S; v \Downarrow_{S(v)} H(v), D(v)}$$

ENDORSEMENTS Unchecked (unsound) endorsements apply only to statically-tracked values and do not affect the correctness probability.

$$\textsc{endorse}\ \frac{H; D; S; e \Downarrow_p V}{H; D; S; \mathsf{endorse}(p_e, e) \Downarrow_p V}$$

CHECKED ENDORSEMENTS Checked endorsements apply to dynamically-tracked values and produce statically-tracked values. The tracked probability must meet or exceed the check's required probability; otherwise, evaluation gets stuck. (Our implementation throws an exception.)

$$\textsc{check}\ \frac{H; D; S; e \Downarrow_p V, p_1 \qquad p_1 \geq p_2}{H; D; S; \mathsf{check}(p_2, e) \Downarrow_p V}$$

TRACKING The static-to-dynamic cast expression allows statically-typed values to be combined with dynamically-tracked ones. The tracked probability field for the value is initialized to match the explicit probability in the expression.

$$\textsc{track}\ \frac{H; D; S; e \Downarrow_p V}{H; D; S; \mathsf{track}(p_d, e) \Downarrow_p V, p_d}$$

OPERATORS Binary operators can be either statically tracked or dynamically tracked. In each case, either operand can be the error value or a constant. When either operand is $\square$, the result is $\square$. When both operands are non-errors, the operator itself can (nondeterministically) produce either $\square$ or a correct result. The correctness probability, however, is the same for all three rules: intuitively, the probability itself is deterministic even though the semantics overall are nondeterministic.

In these rules, $c_1 \oplus c_2$ without a probability subscript denotes the appropriate binary operation on integer or floating-point values. The statically-tracked cases are:

OP

$$\frac{H; D; S; e_1 \Downarrow_{p_1} c_1 \qquad H; D; S; e_2 \Downarrow_{p_2} c_2 \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p c_1 \oplus c_2}$$

OP-OPERATOR-INCORRECT

$$\frac{H; D; S; e_1 \Downarrow_{p_1} c_1 \qquad H; D; S; e_2 \Downarrow_{p_2} c_2 \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square}$$

OP-OPERANDS-INCORRECT

$$\frac{H; D; S; e_1 \Downarrow_{p_1} \square \text{ or } H; D; S; e_2 \Downarrow_{p_2} \square \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square}$$

The dynamic-tracking rules are similar, with the additional propagation of the conservative probability field.

OP-DYN

$$\frac{H; D; S; e_1 \Downarrow_{p_1} c_1, p_{d1} \qquad H; D; S; e_2 \Downarrow_{p_2} c_2, p_{d2} \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p c_1 \oplus c_2,\ p_{d1} \cdot p_{d2} \cdot p_{\text{op}}}$$

OP-DYN-OPERATOR-INCORRECT

$$\frac{H; D; S; e_1 \Downarrow_{p_1} c_1, p_{d1} \qquad H; D; S; e_2 \Downarrow_{p_2} c_2, p_{d2} \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square,\ p_{d1} \cdot p_{d2} \cdot p_{\text{op}}}$$

OP-DYN-OPERANDS-INCORRECT

$$\frac{H; D; S; e_1 \Downarrow_{p_1} \square, p_{d1} \text{ or } H; D; S; e_2 \Downarrow_{p_2} \square, p_{d2} \qquad p = p_1 \cdot p_2 \cdot p_{\text{op}}}{H; D; S; e_1 \oplus_{p_{\text{op}}} e_2 \Downarrow_p \square,\ p_{d1} \cdot p_{d2} \cdot p_{\text{op}}}$$

B.3.2 *Statement Semantics*

The small-step judgment for statements is $H; D; S; s \longrightarrow H'; D'; S'; s'$.

ASSIGNMENT The rules for assignment (initializing a fresh variable) take advantage of nondeterminism in the evaluation of expressions to nondeterministically update the heap with either a constant or the error value, $\square$.

$$\boxed{H; D; s \longrightarrow H'; D'; s'}$$

ASSIGN

$$\frac{H; D; S; e \Downarrow_p V}{H; D; S; \texttt{@Approx}(p')\ \tau\ v := e \longrightarrow H, v \mapsto V; D; S, v \mapsto p; \textbf{skip}}$$

ASSIGN-DYN

$$\frac{H; D; S; e \Downarrow_p V, p_d}{H; D; S; \texttt{@Dyn}\ \tau\ v := e \longrightarrow H, v \mapsto V; D, v \mapsto p_d; S, v \mapsto p; \textbf{skip}}$$

Mutation works like assignment, but existing variables are overwritten in the heap.

MUTATE

$$\frac{H; D; S; e \Downarrow_p V}{H; D; S; v := e \longrightarrow H, v \mapsto V; D; S, v \mapsto p; \textbf{skip}}$$

MUTATE-DYN

$$\frac{H; D; e \Downarrow_p V, p_d}{H; D; v := e \longrightarrow H, v \mapsto V; D, v \mapsto p_d; S, v \mapsto p; \textbf{skip}}$$

SEQUENCING Sequencing is standard and deterministic.

SEQ-SKIP

$$\frac{}{H; D; S; \textbf{skip}; s \longrightarrow H; D; S; s}$$

SEQ

$$\frac{H; D; S; s_1 \longrightarrow H'; D'; S'; s_1'}{H; D; S; s_1; s_2 \longrightarrow H'; D'; S'; s_1'; s_2}$$

IF AND WHILE The type system requires conditions in if and while control flow decisions to be deterministic ($p = 1.0$).

IF-TRUE

$$\frac{H; D; S; e \Downarrow_{1.0} c \qquad c \neq 0}{H; D; S; \textbf{if}\ e\ s_1\ s_2 \longrightarrow H; D; S; s_1}$$

IF-FALSE

$$\frac{H; D; S; e \Downarrow_{1.0} c \qquad c = 0}{H; D; S; \textbf{if}\ e\ s_1\ s_2 \longrightarrow H; D; S; s_2}$$

WHILE

$$\frac{}{H; D; S; \textbf{while}\ e\ s \longrightarrow H; D; S; \textbf{if}\ e\ (s; \textbf{while}\ e\ s)\ \textbf{skip}}$$

B.4 THEOREMS

The purpose of the formalism is to express a soundness theorem that shows that PROB's probability types act as lower bounds on programs' run-time probabilities. We also sketch the proof of a theorem stating that the bookkeeping probability map, S, is eraseable: it is used only for the purpose of our soundness theorem and does not affect the heap.

B.4.1 *Soundness*

The soundness theorem for the language states that the probability types are lower bounds on the run-time correctness probabilities. Specifically, both the

static types @Approx(p) and the dynamically tracked probabilities in D are lower bounds for the corresponding probabilities in S.

To state the soundness theorem, we first define well-formed dynamic states. We write $\vdash D, S : \Gamma$ to denote that the dynamic probability field map D and the actual probability map S are *well-formed* in the static context Γ.

Definition 2 (Well-Formed)**.** $\vdash D, S : \Gamma$ *iff for all* $v \in \Gamma$,

- *If* $\Gamma(v) = \texttt{@Approx}(p)\ \tau$, *then* $p \leq S(v)$ *or* $v \notin S$.
- *If* $\Gamma(v) = \texttt{@Dyn}\ \tau$, *then* $D(v) \leq S(v)$ *or* $v \notin S$.

We can now state and prove the soundness theorem. We first give the main theorem and then two preservation lemmas, one for expressions and one for statements.

Theorem 5 (Soundness)**.** *For all programs* s *with no* endorse *expressions, for all* $n \in \mathbb{N}$ *where* $\cdot;\cdot;\cdot;s \longrightarrow^n H;D;S;s'$, *if* $\cdot \vdash s : \Gamma$, *then* $\vdash D, S : \Gamma$.

Proof. Induct on the number of small steps, n. When $n = 0$, both conditions hold trivially since $v \notin \cdot$ for all v.

For the inductive case, we assume that:

$$\cdot;\cdot;\cdot;s \longrightarrow^n H_1;D_1;S_1;s_1$$

and:

$$H_1;D_1;S_1;s_1 \longrightarrow H_2;D_2;S_2;s_2$$

and that $\vdash D_1, S_1 : \Gamma$. We need to show that $\vdash D_2, S_2 : \Gamma$ also. The Statement Preservation lemma, below, applies and meets this goal. □

The first lemma is a preservation property for expressions. We will use this lemma to prove a corresponding preservation lemma for statements, which in turn applies to prove the main theorem.

Lemma 3 (Expression Preservation)**.** *For all expressions* e *with no* endorse *expressions where* $\Gamma \vdash e : T$ *and where* $\vdash D, S : \Gamma$,

- *If* $T = \texttt{@Approx}(p)\ \tau$, *and* $H;D;S;e \Downarrow_{p'} V$, *then* $p \leq p'$.
- *If* $T = \texttt{@Dyn}\ \tau$, *and* $H;D;S;e \Downarrow_{p'} V, p$, *then* $p \leq p'$.

Proof. Induct on the typing judgment for expressions, $\Gamma \vdash e : T$.

CASE OP-STATIC-TYPES Here, $e = e_1 \oplus_{p_{op}} e_2$ and $T = \texttt{@Approx}(p)\ \tau$. We also have types for the operands: $\Gamma \vdash e_1 : \texttt{@Approx}(p_1)\ \tau_1$ and $\Gamma \vdash e_2 : \texttt{@Approx}(p_2)\ \tau_2$.

By inversion on $H;D;S;e \Downarrow_{p'} V$ (in any of the three operator cases OP, OP-OPERATOR-INCORRECT, or OP-OPERANDS-INCORRECT), $p' = p'_1 \cdot p'_2 \cdot p_{\text{op}}$ where $H;D;S;e_1 \Downarrow_{p'_1} V_1$ and $H;D;S;e_2 \Downarrow_{p'_2} V_2$.

By applying the induction hypothesis to e_1 and e_2, we have $p_1 \leq p'_1$ and $p_2 \leq p'_2$. Therefore, $p_1 \cdot p_2 \cdot p_{\text{op}} \leq p'_1 \cdot p'_2 \cdot p_{\text{op}}$ and, by substitution, $p \leq p'$.

CASE OP-DYN-TYPES The case for dynamically-tracked expressions is similar. Here, $e = e_1 \oplus_{p_{op}} e_2$ and $T = \texttt{@Dyn}\ \tau$, and the operand types are $\Gamma \vdash e_1 : \texttt{@Dyn}\ \tau_1$ and $\Gamma \vdash e_2 : \texttt{@Dyn}\ \tau_2$.

By inversion on $H; D; S; e \Downarrow_{p'} V, p$ (in any of the cases OP-DYN, OP-DYN-OPERATOR-INCORRECT, or OP-DYN-OPERANDS-INCORRECT), $p' = p'_1 \cdot p'_2 \cdot p_{\text{op}}$, $p = p_{d1} \cdot p_{d2} \cdot p_{\text{op}}$ where $H; D; S; e_1 \Downarrow_{p'_1} V_1, p_{d1}$ and $H; D; S; e_2 \Downarrow_{p'_2} V_2, p_{d2}$.

By applying the induction hypothesis to e_1 and e_2, we have $p_{d1} \leq p'_1$ and $p_{d2} \leq p'_2$. Therefore, $p_{d1} \cdot p_{d2} \cdot p_{\text{op}} \leq p'_1 \cdot p'_2 \cdot p_{\text{op}}$ and, by substitution, $p \leq p'$.

CASES CONST-INT-TYPES AND CONST-FLOAT-TYPES Here, we have that $\Gamma \vdash e : \texttt{@Approx}(p)\ \tau$ where $\tau \in \{\text{int}, \text{float}\}$ and $p = 1.0$.

By inversion on $H; D; S; e \Downarrow_{p'} V$ we get $p' = 1.0$.

Because $1.0 \leq 1.0$, we have $p \leq p'$.

CASE VAR-TYPES Here, $e = v, \Gamma \vdash v : T$. Destructing T yields two subcases.

- Case $T = \texttt{@Approx}(p)\ \tau$: By inversion on $H; D; S; e \Downarrow_{p'} V$ we have $p' = S(V)$.

  The definition of well-formedness gives us $p \leq S(V)$.

  By substitution, $p \leq p'$.

- Case $T = \texttt{@Dyn}\ \tau$: By inversion on $H; D; S; e \Downarrow_{p'} V, p$, we have $p' = S(V)$ and $p = D(V)$.

  Well-formedness gives us $D(V) \leq S(V)$.

  By substitution, $p \leq p'$.

CASE ENDORSE-TYPES The expression e may not contain endorse expressions so the claim hold vacuously.

CASE CHECK-TYPES Here, $e = \mathsf{check}(p, e_c)$.

By inversion on $H; D; S; e \Downarrow_{p'} V$, we have $H; D; S; e_c \Downarrow_{p'} V, p''$, and $p \leq p''$.

By applying the induction hypothesis to $H; D; S; e_c \Downarrow_{p'} V, p''$, we get $p'' \leq p'$.

By transitivity of inequalities, $p \leq p'$.

CASE TRACK-TYPES Here, $e = \mathsf{track}(p_t, e_t)$, $\Gamma \vdash e_t : \texttt{@Approx}(p'')$, and $p \leq p''$.

By inversion on $H; D; S; e \Downarrow_{p'} V, p$, we get $H; D; S; e_t \Downarrow_{p'} V$.

By applying the induction hypothesis to $H; D; S; e_t \Downarrow_{p'} V$, we get $p'' \leq p'$.

By transitivity of inequalities, $p \leq p'$.

CASE SUBSUMPTION The case where $T = \texttt{@Approx}(p)\ \tau$ applies. There is one rule for subtyping, so we have $\Gamma \vdash e : \texttt{@Approx}(p_s)\ \tau$ where $p_s \geq p$. By induction, $p_s \leq p'$, so $p \leq p'$. □

Finally, we use this preservation lemma for expressions to prove a preservation lemma for statements, completing the main soundness proof.

Lemma 4 (Statement Preservation). *For all programs s with no* endorse *expressions, if $\Gamma \vdash s : \Gamma'$, and $\vdash D, S : \Gamma$, and $H; D; S \longrightarrow H'; D'; S'$, then $\vdash D', S' : \Gamma'$.*

Proof. We induct on the derivation of the statement typing judgment, $\Gamma \vdash s : \Gamma'$.

CASES SKIP-TYPES, IF-TYPES, AND WHILE-TYPES In these cases, $\Gamma = \Gamma'$, $D = D'$, and $S = S'$, so preservation holds trivially.

CASE SEQ-TYPES Here, $s = s_1; s_2$ and the typing judgments for the two component statements are $\Gamma \vdash s_1 : \Gamma_2$ and $\Gamma_2 \vdash s_2 : \Gamma'$. If $s_1 = \textbf{skip}$, then the case is trivial. Otherwise, by inversion on the small step, $H; D; S; s_1 \longrightarrow H'; D'; S'; s_1'$ and, by the induction hypothesis, $\vdash D_1', S_1' : \Gamma$.

CASE DECL-TYPES The statement s is $Tv := e$ where $\Gamma \vdash e : T$ and $\Gamma' = \Gamma, v : T$. We consider two cases: either $T = \texttt{@Approx}(p)\ \tau$ or $T = \texttt{@Dyn}\ \tau$. In either case, the expression preservation lemma applies.

In the first case, $H; D; S; e \Downarrow_{p'} V$ where $p \leq p'$ via expression preservation and, by inversion, $S' = S, v \mapsto p$ and $D' = D$. Since $S'(v) = p \leq p'$, the well-formedness property $\vdash D, S : \Gamma'$ continues to hold.

In the second case $H; D; S; e \Downarrow_{p'} V, p_d$ where $p_d \leq p'$. By inversion, $S' = S, v \mapsto p$ and $D' = D, v \mapsto p_d$. Since $D'(v) = p_d \leq p'$, we again have $\vdash D, S : \Gamma'$.

CASE MUTATE-TYPES The case where s is $v := e$ proceeds similarly to the above case for declarations. □

B.4.2 *Erasure of Probability Bookkeeping*

We state (and sketch a proof for) an *erasure* property that shows that the "true" probabilities in our semantics, called S, do not affect execution. This property emphasizes that S is bookkeeping for the purpose of stating our soundness result—it corresponds to no run-time data. Intuitively, the theorem states that the steps taken in our dynamic semantics are insensitive to S: that S has no effect on which H', D', or s' can be produced.

In this statement, $\mathrm{Dom}(S)$ denotes the set of variables in the mapping S.

Theorem 6 (Bookkeeping Erasure). *If $H; D; S_1; s \longrightarrow^n H'; D'; S_1'; s'$, then for any probability map S_2 for which $\mathrm{Dom}(S_1) = \mathrm{Dom}(S_2)$, there exists another map S_2' such that $H; D; S_2; s \longrightarrow^n H'; D'; S_2'; s'$.*

Proof sketch. The intuition for the erasure property is that no rule in the semantics uses $S(v)$ for anything other than producing a probability in the $\Downarrow_p$ judgment, and that those probabilities are only ever stored back into S.

The proof proceeds by inducting on the number of steps, n. The base case ($n = 0$) is trivial; for the inductive case, the goal is to show that a single step pre-

serves H', D', and s' when the left-hand probability map S is replaced. Two lemmas show that replacing S with S' in the expression judgments leads to the same result value V and, in the dynamically-tracked case, the same tracking probability p_d. Finally, structural induction on the small-step statement judgment shows that, in every rule, the expression probability affects only S itself.

C

PROBABILISTIC ASSERTIONS: EQUIVALENCE PROOF

This appendix expands on the semantics for probabilistic assertions, in Chapter 5, and gives the full proof of the associated theorem. It is based on the digital material accompanying the paper on probabilistic assertions in PLDI 2014 [183].

C.1 SEMANTICS

This section formalizes a simple probabilistic imperative language, PROBCORE, and MAYHAP's distribution extraction process. We describe PROBCORE's syntax, a *concrete semantics* for nondeterministic run-time execution, and a *symbolic semantics* for distribution extraction. Executing a PROBCORE program under the symbolic semantics produces a Bayesian network for a passert statement. We prove this extracted distribution is equivalent to the original program under the concrete semantics, demonstrating the soundness of MAYHAP's core analysis.

C.1.1 *Core Language*

PROBCORE is an imperative language with assignment, conditionals, and loops. Programs use probabilistic behavior by sampling from a distribution and storing the result, written $v \leftarrow D$. Without loss of generality, a program is a sequence of statements followed by a single passert, since we may verify a passert at any program point by examining the program prefix leading up to the passert.

Figure 27 defines PROBCORE's syntax for programs denoted P, which consist of conditionals C, expressions E, and statements S. For example, we write the location obfuscator from earlier as:

$$
\begin{aligned}
&P \equiv S \,;; \textbf{passert}\, C \\
&C \equiv E < E \mid E = E \mid C \wedge C \mid C \vee C \mid \neg C \\
&E \equiv E + E \mid E * E \mid E \div E \mid R \mid V \\
&S \equiv V := E \mid V \leftarrow D \mid S \,;\, S \mid \textbf{skip} \mid \textbf{if}\, C\, S\, S \mid \textbf{while}\, C\, S \\
&R \in \mathrm{R},\ V \in \text{Variables},\ D \in \text{Distributions}
\end{aligned}
$$

Figure 27: Syntax of PROBCORE.

```
locationX ← Longitude; locationY ← Latitude;
noiseX ← Gauss[0, 1]; noiseY ← Gauss[0, 1];
newX := locationX + noiseX; newY = locationY + noiseY;
dSquared := ((locationX - newX) * (locationY - newY))
  + ((locationY - newY) * (locationY - newY));;
passert dSquared < 100
```

We draw the `Longitude` and `Latitude` inputs from opaque distributions and noise from `Gauss[0, 1]`. The entirety of `Gauss[0, 1]` is an opaque label; `0` and `1` are not expressions in our simple language.

c.1.2 *Concrete Semantics*

The concrete semantics for PROBCORE reflect a straightforward execution in which each sampling statement $V \leftarrow D$ draws a new value. To represent distributions and sampling, we define distributions as functions from a sufficiently large set of *draws* $\mathcal{S}$. The draws are similar to the seed of a pseudorandom number generator: a sequence Σ of draws dictates the probabilistic behavior of PROBCORE programs.

We define a large-step judgment $(H, e) \Downarrow_c v$ for expressions and conditions and a small-step semantics $(\Sigma, H, s) \rightarrow_c (\Sigma', H', s')$ for statements. In the small-step semantics, the heap H consists of the variable-value bindings (queried with $H(v)$) and Σ is the sequence of draws (destructed with $\sigma : \Sigma'$). The result of executing a program is a Boolean declaring whether or not the condition in the passert was satisfied at the end of this particular execution.

The rules for most expressions and statements are standard. The rules for addition and assignment are representative:

$$\textsc{plus}\ \frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 + e_2) \Downarrow_c v_1 + v_2}$$

$$\textsc{assign}\ \frac{(H, e) \Downarrow_c x}{(\Sigma, H, v := e) \rightarrow_c (\Sigma, (v \mapsto x) : H, \mathbf{skip})}$$

Figure 28 gives the full set of rules for the concrete semantics. The rule for the sampling statement, $V \leftarrow D$, consumes a draw σ from the head of the sequence Σ. It uses the draw to compute the sample, $d(\sigma)$.

$$\textsc{sample}\ \frac{\Sigma = \sigma : \Sigma'}{(\Sigma, H, v \leftarrow d) \rightarrow_c (\Sigma', (v \mapsto d(\sigma)) : H, \mathbf{skip})}$$

The result of an execution under the concrete semantics is the result of the passert condition after evaluating the program body. We use the standard definition of $\rightarrow_c^*$ as the reflexive, transitive closure of the small step judgment:

$$\textsc{passert}\ \frac{(\Sigma, H_0, s) \rightarrow_c^* (\Sigma', H', \mathbf{skip}) \qquad (H', c) \Downarrow_c b}{(\Sigma, H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_c b}$$

PLUS
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 + e_2) \Downarrow_c v_1 + v_2}$$

MULT
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 * e_2) \Downarrow_c v_1 v_2}$$

DIVD
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 \div e_2) \Downarrow_c v_1 / v_2}$$

REAL
$$\frac{}{(H, r) \Downarrow_c r}$$

VARB
$$\frac{}{(H, v) \Downarrow_c H(v)}$$

LT
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 < e_2) \Downarrow_c v_1 < v_2}$$

EQ
$$\frac{(H, e_1) \Downarrow_c v_1 \qquad (H, e_2) \Downarrow_c v_2}{(H, e_1 = e_2) \Downarrow_c v_1 = v_2}$$

AND
$$\frac{(H, c_1) \Downarrow_c b_1 \qquad (H, c_2) \Downarrow_c b_2}{(H, c_1 \wedge c_2) \Downarrow_c b_1 \wedge b_2}$$

OR
$$\frac{(H, c_1) \Downarrow_c b_1 \qquad (H, c_2) \Downarrow_c b_2}{(H, c_1 \vee c_2) \Downarrow_c b_1 \vee b_2}$$

NEG
$$\frac{(H, c) \Downarrow_c b}{(H, \neg c) \Downarrow_c \neg b}$$

ASSIGN
$$\frac{(H, e) \Downarrow_c x}{(\Sigma, H, v := e) \to_c (\Sigma, (v \mapsto x) : H, \mathbf{skip})}$$

SAMPLE
$$\frac{\Sigma = \sigma : \Sigma'}{(\Sigma, H, v \leftarrow d) \to_c (\Sigma', (v \mapsto d(\sigma)) : H, \mathbf{skip})}$$

PROGN
$$\frac{(\Sigma, H, s_1) \to_c (\Sigma', H', s_1')}{(\Sigma, H, s_1; s_2) \to_c (\Sigma', H', s_1'; s_2)}$$

PROG1
$$\frac{}{(\Sigma, H, \mathbf{skip}; s_2) \to_c (\Sigma, H, s_2)}$$

WHEN
$$\frac{(H, c) \Downarrow_c \mathbf{true}}{(\Sigma, H, \mathbf{if}\ c\ s_1\ s_2) \to_c (\Sigma, H, s_1)}$$

UNLESS
$$\frac{(H, c) \Downarrow_c \mathbf{false}}{(\Sigma, H, \mathbf{if}\ c\ s_1\ s_2) \to_c (\Sigma, H, s_2)}$$

WHILE
$$\frac{}{(\Sigma, H, \mathbf{while}\ c\ s) \to_c (\Sigma, H, \mathbf{if}\ c\ (s; \mathbf{while}\ c\ s)\ \mathbf{skip})}$$

PASSERT
$$\frac{(\Sigma, H_0, s) \to_c^* (\Sigma', H', \mathbf{skip}) \qquad (H', c) \Downarrow_c b}{(\Sigma, H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_c b}$$

Figure 28: The concrete semantics. We use a big-step operational semantics for conditions and expressions, and a small-step operational semantics for statements and programs. Both use a heap H, which stores variable-value bindings. The small-step operational semantics uses a stream Σ of draws.

C.1.3 *Symbolic Semantics*

While the concrete semantics above describe PROBCORE program execution, the symbolic semantics in this section describe MAYHAP's distribution extraction. Values in the symbolic semantics are expression trees that represent Bayesian networks. The result of a symbolic execution is the expression tree corresponding to the passert condition, as opposed to a Boolean.

The language for expression trees includes conditions denoted C_o, real-valued expressions E_o, constants, and distributions:

$$C_o \equiv E_o < E_o \mid E_o = E_o \mid C_o \wedge C_o \mid C_o \vee C_o \mid \neg C_o$$
$$E_o \equiv E_o + E_o \mid E_o * E_o \mid E_o \div E_o \mid R \mid \langle D, E_o \rangle \mid \textbf{if}\ C_o\ E_o\ E_o$$
$$R \in \mathrm{R},\ D \in \text{Distributions}$$

Instead of the stream of draws Σ used in the concrete semantics, the symbolic semantics tracks a stream offset and the distribution D for every sample. Different branches of an **if** statement can sample a different number of times, so the stream offset may depend on a conditional; thus, the stream offset in $\langle d, n \rangle$ is an expression in E_o and not a simple natural number. The symbolic semantics does not evaluate distributions, so the draws themselves are not required. Expression trees do not contain variables because distribution extraction eliminates them.

The symbolic semantics again has big-step rules $\Downarrow_s$ for expressions and conditions and small-step rules $\rightarrow_s$ for statements. Instead of real numbers, however, expressions evaluate to expression trees in E_o and the heap H maps variables to expression trees. For example, the rules for addition and assignment are:

PLUS
$$\frac{(H, e_1) \Downarrow_s \{x_1\} \qquad (H, e_2) \Downarrow_s \{x_2\}}{(H, e_1 + e_2) \Downarrow_s \{x_1 + x_2\}}$$

ASSIGN
$$\frac{(H, e) \Downarrow_s \{x\}}{(n, H, v := e) \rightarrow_s (n, (v \mapsto \{x\}) : H, \textbf{skip})}$$

The syntax $\{x\}$ represents an expression in E_o, with the brackets intended to suggest quotation or suspended evaluation. Figure 29 lists the full set of rules.

The rule for samples produces an expression tree that captures the distribution and the current stream offset:

SAMPLE
$$\frac{}{(n, H, v \leftarrow d) \rightarrow_s (n + 1, (v \mapsto \{\langle d, n \rangle\}) : H, \textbf{skip})}$$

Each sample statement increments the stream offset, uniquely identifying a sample expression tree. This enumeration is crucial. For example, enumerating samples distinguishes the statement $x \leftarrow d;\ y := x + x$ from a similar program using two samples: $x_1 \leftarrow d;\ x_2 \leftarrow d;\ y := x_1 + x_2$. This approach to numbering samples resembles *naming* in Wingate et al. [225].

The symbolic semantics must consider both sides of an **if** statement. For each **if** statement, we need to merge updates from both branches and form conditional expression trees for conflicting updates. We introduce a function merge,

PLUS
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 + e_2) \Downarrow_s \{x_1 + x_2\}}$$

MULT
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 * e_2) \Downarrow_s \{x_1 * x_2\}}$$

DIVD
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 \div e_2) \Downarrow_s \{x_1 \div x_2\}}$$

REAL
$$\frac{}{(H, r) \Downarrow_s \{r\}}$$

VARB
$$\frac{}{(H, v) \Downarrow_s H(v)}$$

LT
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 < e_2) \Downarrow_s \{x_1 < x_2\}}$$

EQ
$$\frac{(H, e_1) \Downarrow_s x_1 \qquad (H, e_2) \Downarrow_s x_2}{(H, e_1 = e_2) \Downarrow_s \{x_1 = x_2\}}$$

AND
$$\frac{(H, c_1) \Downarrow_s x_1 \qquad (H, c_2) \Downarrow_s x_2}{(H, c_1 \wedge c_2) \Downarrow_s \{x_1 \wedge x_2\}}$$

OR
$$\frac{(H, c_1) \Downarrow_s x_1 \qquad (H, c_2) \Downarrow_s x_2}{(H, c_1 \vee c_2) \Downarrow_s \{x_1 \vee x_2\}}$$

NEG
$$\frac{(H, c) \Downarrow_s x}{(H, \neg c) \Downarrow_s \{\neg x\}}$$

ASSIGN
$$\frac{(H, e) \Downarrow_s \{x\}}{(n, H, v := e) \rightarrow_s (n, (v \mapsto \{x\}) : H, \mathbf{skip})}$$

SAMPLE
$$\frac{}{(\{n\}, H, v \leftarrow d) \rightarrow_s (\{n+1\}, (v \mapsto \{\langle d, n\rangle\}) : H, \mathbf{skip})}$$

PROGN
$$\frac{(n, H, s_1) \rightarrow_s (n', H', s_1')}{(n, H, s_1; s_2) \rightarrow_s (n', H', s_1' \,;\, s_2)}$$

PROG1
$$\frac{}{(n, H, \mathbf{skip}; s_2) \rightarrow_s (n, H, s_2)}$$

IF
$$\frac{\begin{array}{c}(H, c) \Downarrow_s \{x\} \\ (n, H, b_t) \rightarrow_s^* (m_t, H_t, \mathbf{skip}) \qquad (n, H, b_f) \rightarrow_s^* (m_f, H_f, \mathbf{skip})\end{array}}{(n, H, \mathbf{if}\ c\ b_t\ b_f) \rightarrow_s (\{\mathbf{if}\ x\ m_t\ m_f\})\mathtt{merge}(H_t, H_f, \{x\}), \mathbf{skip})}$$

WHILE
$$\frac{}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{if}\ c\ (\mathbf{while}\ c\ s))}$$

WHILE0
$$\frac{(H, c) \Downarrow_s \{x\} \qquad \forall \Sigma, (\Sigma, \{x\}) \Downarrow_o \mathbf{false}}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{skip})}$$

PASSERT
$$\frac{(0, H_0, s) \rightarrow_s^* (n, H', \mathbf{skip}) \qquad (H', c) \Downarrow_s \{x\}}{(H_0, s \,;;\, \mathsf{passert}\ c) \Downarrow_s \{x\}}$$

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a \neq b}{\mathtt{merge}(H_t, H_f, \{x\})(v) = \{\mathbf{if}\ x\ a\ b\}}$$

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a = b}{\mathtt{merge}(H_t, H_f, \{x\})(v) = a}$$

Figure 29: The symbolic semantics produce an expression tree. We use a big-step style for conditions and expressions, and small-step style for statements. Each big step has the form $(H, e) \Downarrow_s \{s_e\}$ or $(H, c) \Downarrow_s \{s_c\}$, where $e \in E$, $c \in C$, and $s_e \in E_o$, and $s_c \in C_o$. H maps variables to expressions in E_o.

which takes two heaps resulting from two branches of an **if** along with the condition and produces a new combined heap. Each variable that does not match across the two input heaps becomes an $\{\mathbf{if}\ c\ a\ b\}$ expression tree in the output heap. The definition of `merge` is straightforward and its post-conditions are:

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a \neq b}{\texttt{merge}(H_t, H_f, \{x\})(v) = \{\mathbf{if}\ x\ a\ b\}}$$

$$\frac{H_t(v) = a \qquad H_f(v) = b \qquad a = b}{\texttt{merge}(H_t, H_f, \{x\})(v) = a}$$

Using the `merge` function, we write the rule for **if** statements:

$$\text{IF}\ \frac{(H, c) \Downarrow_s \{x\} \qquad (H, b_t) \rightarrow_s^* (H_t, \mathbf{skip}) \qquad (H, b_f) \rightarrow_s^* (H_f, \mathbf{skip})}{(n, H, \mathbf{if}\ c\ b_t\ b_f) \rightarrow_s (n, \texttt{merge}(H_t, H_f, \{x\}), \mathbf{skip})}$$

Our symbolic semantics assumes terminating **while** loops. Symbolic execution of potentially-unbounded loops is a well-known problem and, accordingly, our formalism only handles loops with non-probabilistic conditions. A simple but insufficient rule for **while** is:

$$\text{WHILE}\ \frac{}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{if}\ c\ (\mathbf{while}\ c\ s))}$$

This rule generates infinite expression trees and prevents the analysis from terminating. We would like our analysis to exit a loop if it can prove that the loop condition is false—specifically, when the condition does not depend on any probability distributions. To capture this property, we add the following rule:

$$\text{WHILE0}\ \frac{(H, c) \Downarrow_s \{x\} \qquad \forall \Sigma, (\Sigma, \{x\}) \Downarrow_o \mathbf{false}}{(n, H, \mathbf{while}\ c\ s) \rightarrow (n, H, \mathbf{skip})}$$

Here, the judgment $(\Sigma, \{x\}) \Downarrow_o v$ denotes evaluation of the expression tree $\{x\}$ under the draw sequence Σ. This rule applies when MAYHAP proves that an expression tree evaluates to **false** independent of the random draws. In our implementation, MAYHAP proves simple cases, when an expression tree contains no samples, and uses black-box sampling otherwise. Section 5.3 describes a more precise analysis that bounds path probabilities, but we leave its formalization to future work.

We can now define the symbolic evaluation of programs:

$$\text{PASSERT}\ \frac{(0, H_0, s) \rightarrow_s^* (n, H', \mathbf{skip}) \qquad (H', c) \Downarrow_s \{x\}}{(H_0, s\ ;;\ \mathsf{passert}\ c) \Downarrow_s \{x\}}$$

To evaluate the resulting expression tree requires a sequence of draws Σ but no heap. Figure 30 shows the full set of rules. As an example, the rules for addition and sampling are representative:

$$\text{PLUS}\ \frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 + e_2) \Downarrow_o v_1 + v_2} \qquad \text{SAMPLE}\ \frac{}{(\Sigma, \langle d, k \rangle) \Downarrow_o d(\sigma_k)}$$

$$\frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 + e_2) \Downarrow_o v_1 + v_2} \qquad \frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 * e_2) \Downarrow_o v_1 * v_2}$$

$$\frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 \div e_2) \Downarrow_o v_1 \div v_2} \qquad \frac{}{(\Sigma, r) \Downarrow_o r} \qquad \frac{(\Sigma, n) \Downarrow_o k}{(\Sigma, \langle d, n \rangle) \Downarrow_o d(\sigma_k)}$$

$$\frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 < e_2) \Downarrow_o v_1 < v_2} \qquad \frac{(\Sigma, e_1) \Downarrow_o v_1 \qquad (\Sigma, e_2) \Downarrow_o v_2}{(\Sigma, e_1 = e_2) \Downarrow_o v_1 = v_2}$$

$$\frac{(\Sigma, c_1) \Downarrow_o b_1 \qquad (\Sigma, c_2) \Downarrow_o b_2}{(\Sigma, c_1 \wedge c_2) \Downarrow_o b_1 \wedge b_2} \qquad \frac{(\Sigma, c_1) \Downarrow_o b_1 \qquad (\Sigma, c_2) \Downarrow_o b_2}{(\Sigma, c_1 \vee c_2) \Downarrow_o b_1 \vee b_2}$$

$$\frac{(\Sigma, c) \Downarrow_o b}{(\Sigma, \neg c) \Downarrow_o \neg b} \qquad \frac{(\Sigma, c) \Downarrow_o \textbf{true} \qquad (\Sigma, e_1) \Downarrow_o v}{(\Sigma, \textbf{if}\ c\ e_1\ e_2) \Downarrow_o v}$$

$$\frac{(\Sigma, c) \Downarrow_o \textbf{false} \qquad (\Sigma, e_2) \Downarrow_o v}{(\Sigma, \textbf{if}\ c\ e_1\ e_2) \Downarrow_o v}$$

Figure 30: The semantics for our simple expression language. Σ is a stream of draws, and σ_k is the k-th element of Σ.

C.2 THEOREM AND PROOF

Theorem 7. *Let $(0, H_0, p) \Downarrow_s \{x\}$, where x is a finite program. Then $(\Sigma, H_0, p) \Downarrow_c b$ if and only if $(\Sigma, x) \Downarrow_o b$.*

Intuitively, this theorem is true because the distribution extraction $\Downarrow_s$ is just a call-by-need lazy evaluation, and $\Downarrow_o$ is the projection of $\Downarrow_c$ over this lazy evaluation. We prove the theorem formally here.

The proof of this theorem proceeds by structural induction on p. First, a few lemmas establish corresponding properties for conditionals, expressions, then statements, and finally programs.

Lemma 5. *For $e \in E$, let $(H_s, e) \Downarrow_s \{x\}$, and suppose that for every variable a, $(\Sigma, H_s(a)) \Downarrow_e H_c(a)$. Then $(H_c, e) \Downarrow_c v$ if and only if $(\Sigma, x) \Downarrow_o v$.*

Proof. The proof is by induction on e. The condition on H_s and H_c is necessary because H_s maps variables to expressions in E_o, while H_c maps variables to real numbers. Note that Σ is unbound; this is because, while Σ is necessary for sampling distributions in E_o, expressions in E do not involve sampling. We examine each of five cases individually.

$e_1 + e_2$ Let $(H_s, e_1) \Downarrow_s \{x_1\}$ and $(H_s, e_2) \Downarrow_s \{x_2\}$. Also let $(H_c, e_1) \Downarrow_c v_1$ and $(H_c, e_2) \Downarrow_c v_2$, so that $(H_c, e_1 + e_2) \Downarrow_c v_1 + v_2 = v$. By the definition of $\Downarrow_s$, $(H_s, e_1 + e_2) \Downarrow_s \{x_1 + x_2\}$, and by induction $(\Sigma, x_1) \Downarrow_o v_1$ and

$(\Sigma, x_2) \Downarrow_o v_2$. Then by the definition of $\Downarrow_o$, $(\Sigma, x) = (\Sigma, x_1 + x_2) \Downarrow_o v_1 + v_2 = v$. Thus this case is established.

r $(H_s, r) \Downarrow_s \{r\}$ and $(\Sigma, r) \Downarrow_c r$; on the other hand, $(H_c, r) \Downarrow_c r$. Thus this case is established.

v $(H_s, v) \Downarrow_s H_s(v)$, while $(H_c, v) \Downarrow_c H_c(v)$. But by hypothesis, we have that $(\Sigma, H_s(v)) \Downarrow_e H_c(a)$, so this case is established.

The cases for $e_1 * e_2$, $e_1 \div e_2$, and $e_1 \div e_2$ are all analogous to the addition expression, $e_1 + e_2$.

These are all the cases present in the definition of E, so the lemma is complete. □

Lemma 6. *For $c \in C$, let $(H_s, c) \Downarrow_s \{x\}$, and suppose that for every variable a, $(\Sigma, H_s(a)) \Downarrow_e H_c(a)$. Then $(H_c, c) \Downarrow_c b$ if and only if $(\Sigma, x) \Downarrow_o b$.*

Proof. We again use induction, on c. We examine each of five cases individually.

$e_1 < e_2$ By the definition of $\Downarrow_s$, $\{x\} = \{x_1 + x_2\}$. Let $(H_c, e_1) \Downarrow_c v_1$ and $(H_c, e_2) \Downarrow_c v_2$, so that $b = [v_1 < v_2]$. By lemma 5, $(\Sigma, x_1) \Downarrow_o v_1$ and $(\Sigma, x_2) \Downarrow_o v_2$, so $(\Sigma, x) \Downarrow_o [v_1 < v_2] = b$. Thus this case is established.

$e_1 = e_2$ This case is analogous to $e_1 < e_2$.

$c_1 \wedge c_2$ Let $(H_s, c_1) \Downarrow_s \{x_1\}$ and $(H_s, c_2) \Downarrow_s \{x_2\}$. Also let $(H_c, c_1) \Downarrow_c b_1$ and $(H_c, c_2) \Downarrow_c b_2$, so that $(H_c, c_1 \wedge c_2) \Downarrow_c b_1 \wedge b_2 = v$. By the definition of $\Downarrow_s$, $(H_s, c_1 \wedge c_2) \Downarrow_s \{x_1 \wedge x_2\}$, and by induction $(\Sigma, x_1) \Downarrow_o b_1$ and $(\Sigma, x_2) \Downarrow_o b_2$. Then by the definition of $\Downarrow_o$, $(\Sigma, x) = (\Sigma, x_1 \wedge x_2) \Downarrow_o b_1 + b_2 = b$. Thus this case is established.

$c_1 \vee c_2$ This case is analogous to $c_1 \wedge c_2$.

$\neg c_1$ Let $(H_s, c_1) \Downarrow_s \{x_1\}$ and $(H_c, c_1) \Downarrow_c b_1$, so that $(H_c, \neg c_1) \Downarrow_c \neg b_1$. By the definition of $\Downarrow_s$, $(H_s, \neg c_1) \Downarrow_s \{\neg x_1\}$, and by induction $(\Sigma, x_1) \Downarrow_o b_1$, so that $(\Sigma, x) \Downarrow_o \neg b_1 = b$. Thus this case is established.

These are all the cases present in the definition of C, so the lemma is complete. □

We now prove a lemma which establishes equivalence for statements that do not contain **while** loops.

Lemma 7. *Let $(n, H_s, s) \rightarrow_s (m, H'_s, s')$, where s contains no **while** statements. Also suppose that $(\Sigma, n) \Downarrow_o l$ and $(\Sigma, m) \Downarrow_o l + k$. Furthermore let H_c be such that $(\Sigma, H_s(v)) \Downarrow_o H_c(v)$ for all variables v. Then $(\Sigma, H_c, s) \rightarrow_c^* (\Sigma', H'_c, s')$, where $\Sigma = \sigma_1 : \sigma_2 : \cdots : \sigma_k : \Sigma'$. Furthermore, $(\Sigma, H'_s(v)) \Downarrow_o H'_c(v)$ for all v.*

Proof. A few difficulties arise when attempting a naive induction:

- While $\Downarrow_c$ and $\rightarrow_c$ consume an element of Σ, $\Downarrow_s$ and $\rightarrow_s$ simply increment an offset. Our induction must show that this offset is correctly handled.

- While $\rightarrow_c$ only evaluates one side of an **if** statement, $\rightarrow_s$ evaluates both. Proving that this is sound requires proving that the "merge" function correctly unifies the two branches.

- Non-terminating while loops, especially those involving sampling, are difficult to handle in the induction. The statement of the lemma guarantees that the while loop must terminate (since $\rightarrow_s^*$ requires a finite number of steps), but the possibility for while loops to not terminate still complicates the proof.

The first problem is avoided by the statement of the lemma: we require that the symbolic semantics increment the sequence offset by exactly as many elements as the concrete semantics consumes. The second problem requires a careful analysis of the "merge" function. This is also why we assume a single step in $\rightarrow_s$ but a number of steps in $\rightarrow_c^*$. Finally, the last problem is avoided by a nested induction over the number of times the **while** loop is unrolled. Since we assume the symbolic semantics terminate, the loop must at some point unroll fully, so the induction is founded.

As mentioned, we induct over the number of steps taken by $\rightarrow_s^*$. At each step, we assume that the future steps will satisfy the statement of the lemma. We consider each case individually.

$v := e$ Assume that $(H_c, e) \Downarrow_c x_c$, so that $(\Sigma, H_c, v := e) \rightarrow_c (\Sigma, (v \mapsto x) : H_c, \mathbf{skip})$. Furthermore, suppose $(H_s, e) \Downarrow_s \{x_s\}$, so that $(n, H_s, v := e) \rightarrow_s (n, (v \mapsto x_s) : H_s, \mathbf{skip})$. By lemma 5, $(\Sigma, x_s) \Downarrow_o x_s$. But then, for all variables v, we have $(\Sigma, ((v \mapsto x_s) : H_s)(v')) \Downarrow_o ((v \mapsto x_c) : H_c)(v')$ for all v'. If we set $\Sigma' = \Sigma$ and $k = 0$, we find that in this case our theorem is proven.

$v \leftarrow d$ Let $\Sigma = \sigma : \Sigma'$. Then $(\Sigma, H_c, v \leftarrow d) \rightarrow_c (\Sigma', (v \mapsto d(\sigma)) : H_c, \mathbf{skip})$ On the other hand, in the symbolic semantics, $(\{n\}, H_s, v \leftarrow d) \rightarrow_s (\{n+1\}, (v \mapsto \langle d, n\rangle) : H_s, \mathbf{skip})$.

We can see that if $(\Sigma, \{n\}) \Downarrow_o l$, then $(\Sigma, \{n+1\}) \Downarrow_o l + 1$, forcing $k = 1$. Indeed, $\Sigma = \sigma_1 : \Sigma'$. Furthermore, since $(\Sigma, \langle d, n\rangle) \Downarrow_o d(\sigma_1) = d(\sigma)$, we know that for all v', $(\Sigma, ((v \mapsto \langle d, n\rangle) : H_s)(v')) \Downarrow_o ((v \mapsto d(\sigma)) : H_c)(v')$. So this case is established.

skip Since there are no symbolic steps for **skip**, the lemma is vacuously true.

$s_1; s_2$ This statement has two cases: where s_1 is **skip**, and where it is not. If s_1 is **skip**, the case is trivial, so suppose s_1 is not **skip**. Furthermore, let $(n, H_s, s_1) \rightarrow_s (m', H_s', s_1')$. By induction, we also have $(\Sigma, H_c, s_1) \rightarrow_c^* (\Sigma'', H_c', s_1')$, with the expected properties relating Σ' and k, and H_c' and H_s'. But then since:

$$(n, H_s, s_1; s_2) \rightarrow_s (m', H_s', s_1'; s_2)$$

and

$$(\Sigma, H_c, s_1; s_2) \rightarrow_c^* (\Sigma'', H_c', s_1'; s_2)$$

this case is established with $m = m'$ and $\Sigma' = \Sigma''$. (We omit the lemma that $s_1 \to^* s_1'$ implies $s_1; s_2 \to^* s_1'; s_2$, with the expected behavior of the other parameters.)

if $c\ s_1\ s_2$ Per Lemma 6, we know that if $(H_c, c) \Downarrow_c b$, and $(H_s, c) \Downarrow_s \{x_s\}$, then $(\Sigma, x_s) \Downarrow_o b$. Now consider two sub-cases: b is true, and b is false. If b is true, then for all expressions y_t and y_f, the expression **if** $x_s\ y_t\ y_f$ must evaluate to the same result as y_t; otherwise if b is false, to the same result as y_f.

Now, depending on b, either:

$$(\Sigma, H_c, \textbf{if}\ c\ s_1\ s_2) \to (\Sigma', H_c', s_1)$$

or:

$$(\Sigma, H_c, \textbf{if}\ c\ s_1\ s_2) \to (\Sigma', H_c', s_2)$$

We know that:

$$(n, H_s, s_1) \to_s^* (m_t, H_{st}, \textbf{skip})$$

and

$$(n, H_s, s_2) \to_s^* (m_f, H_{sf}, \textbf{skip})$$

But then by induction, we know that $(\Sigma, H_c, s_1) \to_c^* (\Sigma_t, H_{ct}, \textbf{skip})$ or $(\Sigma, H_c, s_2) \to_c^* (\Sigma_f, H_{cf}, \textbf{skip})$, where the relationship of Σ_t to m_t, of Σ_f to m_f, of H_{ct} to H_{st}, and of H_{cf} to H_{sf} are as expected. Thus, when $(n, H_s, \textbf{if}\ c\ s_1\ s_2) \to (m, H_s', \textbf{skip})$, we know that $(\Sigma, H_c, \textbf{if}\ c\ s_1\ s_2) \to (\Sigma', H_c', \textbf{skip})$ as required, where Σ' is Σ_t or Σ_f depending on the condition, and where H_c' is H_{ct} or H_{cf}, again depending on the loop condition.

All that remains to prove is that the symbolic inference rule for the **if** rule correctly combines H_{st} and H_{sf}, and likewise correctly combines m_t and m_f. Recall that b is the value of the loop condition, and the loop conditional evaluates symbolically to x_s. We do a case analysis on b. First, suppose b is true. Then $\Sigma' = \Sigma_t$, so we know that $\Sigma' = \sigma_1 : \cdots \sigma_k : \Sigma'$ where $(\Sigma, m) = (\Sigma, \textbf{if}\ x_s\ m_t\ m_f) \Downarrow_o k$. Similarly, since $H_s' = \texttt{merge}(H_{sf}, H_{st}, x_s)$, we know that for all variables v:

$$(\Sigma, H_s'(v)) = (\Sigma, \texttt{merge}(H_{st}, H_{sf}, x_s)(v))$$

This is equal to either $(\Sigma, \textbf{if}\ x_s\ (H_{st}(v))\ (H_{sf}(v)))$ or $(\Sigma, H_{st}(v))$, both of which evaluate to $H_{ct}(v) = H_c'(v)$ because x_s evaluates to b which is true, and because $(\Sigma, H_{st}(v)) = H_{ct}(v)$ by induction. The case where b is false is analogous to the case where b is true.

Thus this case is established.

This was the last remaining case (we assume that s_1 contains no **while** statements), so the lemma is done. □

We now extend the equivalence to programs that contain while loops. We require that the symbolic evaluation terminate.

Lemma 8. *Let* $(n, H_s, s) \rightarrow_s^* (m, H_s', \mathbf{skip})$. *Further suppose that for all variables* v, $(\Sigma, H_s(v)) \Downarrow_o H_c(v)$. *Then* $(\Sigma, H_c, s) \rightarrow_c^* (\Sigma', H_c', \mathbf{skip})$, *and furthermore for all variables* v, $(\Sigma, H_s'(v)) = H_c'(v)$ *and also* $\Sigma = \sigma_1 : \cdots : \sigma_k : \Sigma'$, *where* $(\Sigma, m) \Downarrow_o l + k$ (*where* $(\Sigma, n) \Downarrow_o l$).

Proof. We proceed by structural induction on s.

$v := e$ There are no **while** loops in this statement, so it follows from lemma 7.

$v \leftarrow d$ Analogous to $v := e$.

skip Analogous to $v := e$.

$s_1; s_2$ We must have $(n, H_s, s_1) \rightarrow_s^* (n', H_s'', \mathbf{skip})$, so by induction we also have $(\Sigma, H_c, s_1) \rightarrow_c^* (\Sigma'', H_c'', \mathbf{skip})$, with the usual relation between Σ'' and n', and between H_s'' and H_c''. By induction, $(n', H_s'', s_2) \rightarrow_s^* (m, H_s', \mathbf{skip})$ implies $(\Sigma'', H_c'', s_2) \rightarrow_c^* (\Sigma', H_c', \mathbf{skip})$.

Thus, $(\Sigma, H_c, s_1; s_2) \rightarrow_c^* (\Sigma', H_c', \mathbf{skip})$, and this case is established.

if $c\ s_1\ s_2$ If $(n, H_s, \mathbf{if}\ c\ s_1\ s_2) \rightarrow^* (n', H_s', \mathbf{skip})$, we must have $(n, H_s, s_1) \rightarrow^* (n_t, H_{st}', \mathbf{skip})$ and $(n, H_s, s_2) \rightarrow^* (n_f, H_{sf}', \mathbf{skip})$. Then, analogously to the argument in lemma 7, this case can be established.

while $c\ s$ There are two inference rules concerning the symbolic semantics of **while** loops, so we must prove that both are sound.

First consider the rule WHILE0. If it applies, we must have $(\Sigma, x) \Downarrow_o$ **false** for $(H_s, c) \Downarrow_s \{x\}$, and thus (by lemma 6) $(H_c, c) \Downarrow_c$ **false**. Then $(\Sigma, H_c, \mathbf{while}\ c\ s) \rightarrow^* (\Sigma, H_c, \mathbf{skip})$.

But by assumption, $(n, H_s, \mathbf{while}\ c\ s) \rightarrow (n, H_s, \mathbf{skip})$, so the inductive statement holds.

Second, consider the rule WHILE in the symbolic semantics. It is identical to the corresponding rule for **while**, so by induction this case is established.

These are all the cases for S, so the lemma is proven. □

Finally, we can prove our Theorem 7.

Theorem 8. *Let* $(0, H_0, p) \Downarrow_s \{x\}$, *where* x *is a finite program. Then* $(\Sigma, H_0, p) \Downarrow_c b$ *if and only if* $(\Sigma, x) \Downarrow_o b$.

Proof. First, note that $H_c = H_s = H_0$, so that $(\Sigma, H_s(v)) \Downarrow_o H_c(v)$ for all v (by the rule for constants).

Let the program p be s ; ; assert c. If $(0, H_0, p) \Downarrow_s \{x\}$, then $(0, H_0, s) \rightarrow_s^* (n, H_s, \mathbf{skip})$. Then by lemma 8, $(\Sigma, H_0, s) \rightarrow_s^* (\Sigma', H_c, \mathbf{skip}$, with the expected relation between H_c and H_s. But then due to this relation, if $(H_s, c) \Downarrow_s \{y\}$, $(H_c, c) \Downarrow_c b$ if and only if $(\Sigma, y) \Downarrow_o b$ (the lemma to prove this would be a straightforward induction over y).

Thus, $(\Sigma, H_0, p) \Downarrow_c b$ if and only if $(\Sigma, x) \Downarrow_o b$, and our theorem is proven. □

While complex, this theorem shows that the distribution extraction performed by MAYHAP is sound.

COLOPHON

This thesis was designed using André Miede's *classicthesis* style for LaTeX. The body, titles, and math are set in Crimson Text, a typeface by Sebastian Kosch. Code appears in `Inconsolata` by Raph Levien, which is based on Luc(as) de Groot's Consolas. Both typefaces are open-source projects. The full source code for this dissertation is available online:
`https://github.com/sampsyo/thesis`