Dynamic Analyses of Result Quality in Energy-Aware Approximate Programs

Michael F. Ringenburg

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

Doctor of Philosophy

University of Washington

2014

Reading Committee:

Luis Ceze, Chair

Dan Grossman, Chair

Mark Oskin

Program Authorized to Offer Degree:
Computer Science and Engineering

University of Washington

Abstract

Dynamic Analyses of Result Quality in Energy-Aware Approximate Programs

Michael F. Ringenburg

Co-Chairs of the Supervisory Committee:
Dr. Luis Ceze
Computer Science and Engineering

Dr. Dan Grossman
Computer Science and Engineering

Energy efficiency is a key concern in the design of modern computer systems. One promising approach to energy-efficient computation, *approximate computing*, trades off output precision for energy efficiency. However, this tradeoff can have unexpected effects on computation quality. This thesis presents dynamic analysis tools to study, debug, and monitor the quality and energy efficiency of approximate computations. We propose three styles of tools: prototyping tools that allow developers to experiment with approximation in their applications, offline tools that instrument code to determine the key sources of error, and online tools that monitor the quality of deployed applications in real time.

Our prototyping tool is based on an extension to the functional language OCaml. We add approximation constructs to the language, an approximation simulator to the runtime, and profiling and auto-tuning tools for studying and experimenting with energy–quality tradeoffs. We also present two offline debugging tools and three online monitoring tools. The first offline tool identifies correlations between output quality and the total number of executions of, and errors in, individual approximate operations. The second tracks the number of approximate operations that flow into a

particular value. Our online tools comprise three low-cost approaches to dynamic quality monitoring. They are designed to monitor quality in deployed applications without spending more energy than is saved by approximation. Online monitors can be used to perform real time adjustments to energy usage in order to meet specific quality goals.

We present prototype implementations of all of these tools and describe their usage with several applications. Our prototyping, profiling, and autotuning tools allow us to experiment with approximation strategies and identify new strategies, our offline tools succeed in providing new insights into the effects of approximation on output quality, and our monitors succeed in controlling output quality while still maintaining significant energy efficiency gains.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank my advisers, Dan Grossman and Luis Ceze, for their guidance and support. Thanks as well to Adrian Sampson for helping me get started with EnerJ, and for collaborating on the projects described in this thesis. I would also like to thank Tom Cormen for inspiring me to enter the field of Computer Science. I am very appreciative of the flexibility shown to me by Cray, Inc., in allowing me to continue to work for them part time while completing this thesis. And of course I am extremely grateful to my family for their love and support.

DEDICATION

To my wife, Severine, and my daughters, Emilie and Sarah

Chapter 1

INTRODUCTION

Energy efficiency has become a critical component of computer system design [8, 21]. The dark silicon problem limits the amount of chip area that will be usable in future chip generations due to power constraints: at the 22 nm process step, 21% of the chip must be powered down at any time [21]. Furthermore, battery life is a major concern in mobile devices and power and cooling bills represent large parts of the costs of running data centers and supercomputers. For example, as of November 2013, the average power consumption of a top 10 system from the Top 500 supercomputer list was 6.50 megawatts [44]. These power costs can run into the millions of dollars annually.

Approximate computing is a promising technique for reducing the power consumption and improving the performance of computing systems [1, 5, 9, 17, 22, 27, 40, 48, 49, 52, 53]. For example, Sidiroglou et al. [53] describe loop perforation, which reduces the amount of work performed by an application by skipping the execution of some loop iterations. Perforation can significantly improve the performance of various benchmarks without introducing unreasonable distortion. Another example is the EnerJ compiler, runtime, and Java language extensions [52]. EnerJ takes a more disciplined approach to approximate computing. Programmers annotate data that can be approximated; the compiler and hardware (or simulator) then cooperate to execute in a low-power, approximate mode when dealing with this data. For example, the system could store data in DRAM with a lower refresh rate that occasionally experiences data corruption, or utilize a lower-powered processor pipeline that suffers from infrequent errors. The EnerJ type system ensures that approximate data does not flow

into non-approximate data unless the programmer explicitly *endorses* (approves) the flow. This allows developers to take advantage of the benefits of approximate computing without sacrificing safety. The EnerJ approach can yield significant energy savings across a variety of benchmarks using custom hardware [22].

However, imprecise computation must be used carefully to avoid compromising too much on software quality. Previous work has given programmers control over the use of approximation [5, 9, 17, 40, 52]. In Relax [17], programmers mark regions of code where hardware errors can safely go uncorrected. In EnerJ [52], a type system distinguishes data that can tolerate errors from data that requires full precision and typing rules prevent approximate-to-precise information flow. Carbin et al [9] propose a proof system for reasoning about acceptability properties in the face of imprecision. The Rely system [10] statically determines the probability that values produced by an approximate computation are correct.

These static approaches are valuable and help bound the negative effects of approximation. However, even with static safety guarantees that prevent outright crashes and bound error margins (such as Relax's spatial error bounding or EnerJ's non-interference), some approximations can be more pernicious than others in terms of their effect on the program's *quality of result* (or QoR). In light of this, we contend that dynamic tools should also play an important role in addressing quality concerns. This is analogous to conventional (non-approximate) software development, where static tools like Coverity [14] or Lint [28] and dynamic tools like Valgrind [43] all play important roles in ensuring software quality.

Based on this observation, this thesis proposes the use of dynamic tools in the context of developing programs with approximation. Specifically, we design tools that can provide more precise understanding of, and control over, the QoR of approximate applications.

We first propose an approximation prototyping system called EnerCaml. EnerCaml extends the OCaml programming language [45] (an ML variant) with constructs

for approximate computing, and adds an approximation simulator to the OCaml runtime. OCaml is known for its strengths as a prototyping language, and these extensions allow it to be used for prototyping approximate applications. The EnerCaml prototyping toolkit also contains two tools for assessing and improving the quality of the prototyped applications and algorithms. The first is an approximation profiler that allows developers to estimate the output quality and energy savings of the approximations they introduced. The second is an autotuner for EnerCaml programs that suggests alternate approximation strategies and points out which strategies lie at optimal points on the quality–efficiency Pareto curve.

We then propose two offline tools for approximation aware programming environments that instrument programs to determine the critical data locations and code points that have the most impact on quality of result. We implemented these tools as LLVM [31] compiler passes for an approximate version of C and C++. Our offline tools can track approximate dataflow into variables and expression results, and can determine correlations between the final output quality of an approximate application or algorithm and the executions of, and errors in, specific approximate instructions. These tools can help debug the quality of approximate applications, and can also provide new insights into the safety and effectiveness of various approximations.

Finally, we propose online tools that dynamically monitor quality and can let programs self-heal by adjusting approximation (or energy) levels or re-executing code in response to quality degradations. We implemented our monitors on top of the EnerJ [52] language, runtime, and simulator. These monitors are specifically designed to have low enough overhead to run during approximate executions while still retaining significant energy savings. This allows the programs to dynamically react to new and unanticipated input patterns or environmental conditions that might otherwise negatively impact quality.

We argue that all three styles (prototyping, offline instrumentation, and online monitoring tools) are important pieces of an approximate programming ecosystem.

Prototyping allows developers to explore candidate applications for approximate programming, and to estimate the energy saving benefits and quality of result tradeoffs. Our prototyping toolkit can also suggest alternate approximation strategies that may improve these tradeoffs. Our offline tools, while too heavyweight for usage in deployment (the costs would more than overwhelm the savings from approximation), are excellent tools for pre-deployment debugging and understanding of quality issues in the application. They help programmers better understand where they can safely use approximation. The online monitoring tools, on the other hand, are lightweight enough to run in deployed code and constantly adjust approximation levels or correct erroneous results when faced with quality issues that arise post-deployment (due, for example, to unanticipated program inputs or variations in approximate hardware). However, they are unable to provide the detailed analyses and important insights into program behavior that our offline tools provide. Taken together, the various tools proposed in this thesis can greatly enhance developers ability to produce high-quality, energy-efficient, approximate applications.

Our contributions include:

- a system for prototyping, experimenting with, and tuning approximation strategies (Chapter 3),
- a tool for dynamically tracking approximate dataflow (Chapter 4),
- a tool for determining correlations between approximate operations and output quality (Chapter 4),
- three approaches to online quality monitoring (Chapter 5), and
- a framework for online monitoring and side effect management (Chapter 5),

Prior to discussing our main contributions, we will provide more background on approximate programming, with a focus on the EnerJ model of approximation (Chap-

ter 2). All of the tool implementations described in this thesis use an approximation model similar to EnerJ. However, the ideas behind them are applicable to alternate models of, and strategies for, approximate computing.

Chapter 2

APPROXIMATE PROGRAMMING: BACKGROUND AND RELATED WORK

This chapter presents background material and related work on approximate programming (Section 2.1) and Quality of Result, or QoR (Section 2.2). Further related work that is more specific to our various tools and techniques will be described in the relevant chapters. We also present a more in-depth discussion of the EnerJ language, simulator, and approximation model, as all of our tools were built on top of similar systems (Section 2.3). The principles behind the tools, however, can be applied to other approximation models.

2.1 Approximate Programming

Approximate programming is a model that allows programmers to trade computation accuracy for energy efficiency or performance. In a language with support for approximate programming, programmers distinguish parts of a program—variables, operations, methods, loops, and so on—that are tolerant to error. The semantics of approximate data and operations are relaxed to allow the execution substrate, in the form of either software [1, 5, 27, 48, 49, 53, 61] or hardware [12, 17, 22, 32, 37, 42, 52], to permit errors to occur where they would otherwise need to be prevented. Several studies have shown that a wide variety of applications can tolerate the resulting imprecision with acceptable results [18, 33, 34, 60].

Many applications have kernels that are amenable to approximation. For example, applications that work with audio, video, or images are inherently tolerant of some error. In fact, the most common storage formats for these media involve lossy com-

pression schemes. Any code that involves a randomized or approximate algorithm is also an excellent choice for approximate programming.[1] Simulations of physical systems are also sometimes good candidates for approximation as they may already involve rounding (i.e., approximation) of various physical quantities such as position and velocity.

Several hardware and software techniques exist that take advantage of the relaxed semantics of approximation-annotated programs to increase their energy efficiency or performance: for example, Flikker [37] reduces the DRAM refresh rate for memory used to store approximate data at the expense of occasional bit flips in the data and several systems [5,27,53] reduce the iteration count of loops in approximate code. Because of the diversity of approaches to exploiting approximation for efficiency gains, a suitably general approximation-aware programming language must allow a wide range of "incorrect" behavior. For example, the semantics of EnerJ [52] define approximate computations to behave arbitrarily, giving no formal guarantees on the output of approximate operations or the consistency of approximate variables. (Informally, the programmer can expect each approximate operation to be faulty but to bear some resemblance to its precise counterpart: for example, an approximate addition may be expected to perform addition with occasional faults, or it may be expected to always produces answers "close" to the correct result.) This "chaotic" definition of approximation allows EnerJ-like languages to generalize to a broad range of hardware and software optimizations, but it hinders programmers' understanding of approximate code and their ability to write reliable programs.

In addition, even the most approximable applications require some code to execute precisely. For example, memory allocation, control flow, and bounds checking calculations usually need to execute precisely to avoid faults. Particular applications

[1] In fact, one of the sample applications we looked at for EnerCaml was a genetic algorithm, but it turned out that when we added approximation the results were a *better* fit for the data than when we ran it precisely.

may also have certain phases that must execute precisely. For instance, an application that saves an image may be able to tolerate some approximation in the pixels of the image, but any approximation in the image header will result in at best a severely distorted image and at worst a completely unreadable image.

2.2 Measuring Quality of Result

These issues motivated us to investigate the problem of understanding and controlling the output quality of approximate applications. While resource usage—time or power, for example—can be measured directly, quality must be assessed using a program-specific metric. We refer to this application-defined notion of output quality as the *quality of result* or QoR. For example, in an object recognition application the QoR metric may be the number of correct classifications.

One way to measure QoR is to run the application (or its approximate portions) twice with identical inputs—once approximately, and once precisely—and compare results. In an offline setting, this could be done repeatedly in a controlled test environment, using a variety of expected inputs. We refer to this as *approximation profiling*. In an online setting, we could do this in real time with every input seen "in the wild". We refer to this as *complete online monitoring*. This section argues why complete online monitoring is inappropriate for the online setting and why approximation profiling (by itself) is insufficient for the offline setting (although it may be useful as part of a package of tools like our EnerCaml system, described in Chapter 3). Along the way, we also discuss some of the key issues that any QoR tool must address. These approaches thus serve as "quality strawmen" to motivate the rest of this dissertation.

The high-level goal of any approximate QoR tool is to measure the effects of approximation on a piece of approximate code or data. For instance, if the code contains approximate arithmetic, we want to detect when arithmetic errors cause the code's output to differ too much from what the results would have been if only

precise arithmetic had been used. For example, consider a ray tracer, where we wish to evaluate the approximate computation of each pixel:

```
evaluate { tracePx(x, y); }
```

The strawman approach mentioned above would run this code twice, and compare the results to see if they are within an acceptable threshold:

```
approx = tracePx(x, y);
precise = runPrecise { tracePx(x, y); }
if (abs(approx - precise) > Threshold)
  throw new FailedQoR();
```

This approach provides exactly what we would like in an online tool: real-time updates (as each approximate computation completes) on the quality of the approximation. This enables programs to respond immediately, e.g., by adjusting parameters to improve future approximations, or by reexecuting erroneous computations. Unfortunately, there are four problems with this approach. First, the code assumes idempotency of the monitored code block. Except in a purely functional setting, approximate computations can and often do have side effects. If we wish to run a non-idempotent code block twice, we need to buffer or roll back side effects. Second, even if we provide some form of side effect buffering or rollback, this approach does not deal with the issue of side effects' impact on quality, which we cover in much more detail in Section 5.4.3 of Chapter 5. Briefly, approximate computations may have side effects that are not directly reflected in the returned result, but may have an impact on perceived quality. Any online approach must address this issue. Third, comparing numeric return values is insufficient for measuring the QoR of many applications. QoR is inherently domain-specific, so we must support application-specific metrics. For example, a video application may prefer neighboring frames that are distorted in the same way (thus preventing jitter) over neighboring frames with smaller average distortion but which are distorted in different ways. Another example is a

greedy algorithm that searches for local optima. An approximate version that selects a different optimum from the precise version can have equal—or possibly even superior—result quality. Lastly, and most importantly, an online monitoring scheme *must not cost more than the savings provided by the original approximation.* By executing the code approximately, and then reexecuting it precisely, we spend strictly more energy than the original, non-approximate code. Chapter 5 shows three ways that we can relax this strawman monitoring approach to provide flexible and lightweight online monitors that effectively control side effects and provide customizable quality metrics.

In the offline case, on the other hand, the cost of this strawman approach is not prohibitive. Offline tools are intended for predeployment usage, during quality testing and debugging, where spending extra time and energy to improve performance in the field is wise. On the other hand, freed from these cost constraints, there is much more that we could do than a simple quality profiler to provide programmers more information about the behavior of approximate programs. The strawman tells us only *what* the final QoR was, and does not indicate *why* it was high or low. It gives us no indication of which approximate operations or data are critical to QoR. Developers need more program introspection, especially when working with approximation. It also gives us no recommendation for how to improve the quality–energy tradeoffs of the application.

This thesis addresses the shortcomings of simple offline profiling. In particular, Chapter 3 proposes a complete offline prototyping system that includes an approximation simulator and profiler, in addition to an autotuner that leverage the profiler to provide approximation recommendations that may improve quality–energy tradeoffs. Then, Chapter 4 proposes approaches that let us track degrees of approximation, as well as executions and errors of approximate calculations, at a much finer grain (individual operations or variables). We can then correlate these with output quality. These tools thus help us find the source of the quality issue, rather than merely to

determine that an issue exists.

2.3 The EnerJ Language

The EnerJ language [52] is an extension to Java that provides support for disciplined approximate programming. This is accomplished via a type qualifier for variables containing approximate data, and typing rules that prevent dataflow from approximate variables to precise variables without an explicit endorsement of the safety of such flow from the programmer. Operations on precise data are guaranteed to return the correct result. Operations on approximate data may return arbitrary results (but for the sake of usability, they should "usually" return the correct answer, or something close to correct). For example, if we store the value 42 to an approximate integer, and then later load it, we will *probably* get back 42, but we *may* get back any integer. Similarly, if we add two approximate integers containing the values 1 and 2, it will probably return 3, but may return any integer. This approach also supports approximation hardware that approximates floating point values by shortening the mantissa length. For example, we could perform a floating point calculation that normally returns the value 3.14159 and instead get back a rounded value such as 3.142. The EnerJ system [52] also includes a type checker that verifies the typing rules, and an approximation simulator (built into the Java runtime) that simulates the effects of approximate hardware for operations performed on approximate data.

Programmers indicate approximate data by using the `@Approx` type qualifier. (A `@Precise` qualifier is also provided, but it is not usually used as it is the default if no qualifiers are present.) For example, to indicate that a pixel value is approximate, a developer could write:

```
@Approx int pixelValue = 0;
```

Any loads from, stores to, or operations involving `pixelVal` may then be executed approximately by the underlying compiler, runtime, and/or hardware. If we try to

assign an approximate value to a non-approximate (i.e., precise) variable, the type checker will issue an error. For example,

```
int brighterPixel = pixelValue + 10;
```

generates the error:

```
error: incompatible types.
    int brighterPixel = pixelValue + 10;


found   : @Approx int
required: @Precise int
```

Similarly, using an approximate result as a control flow condition, such as in the following example, will generate an error:

```
error: The type in a conditional operation cannot be approximate!
    if (pixelValue > 100) {
       ^
     Found @Approx boolean.
```

In order to use an approximate result to set a precise variable, or as a control flow condition, the programmer must explicitly make the result precise by *endorsing* it, via the `endorse` operation:

```
if (endorse(pixelValue > 100)) {
  pixelValue = 100;
}
```

The `endorse` operation can be thought of as an identity function which takes an approximate value, and returns the equivalent precise value. Equivalently, it can be viewed as a type cast from a type `@Approx T` to a type `@Precise T`.

In the absence of endorsements, this system provides a *non-interference* property, as shown in [52]. Specifically, changing approximate values or results during execution will not result in a change to any precise values or results. This allows developers to

reason about their approximate applications, and gain confidence that approximation will not effect values they wish to keep precise unless they explicitly endorse the flow.

The hardware modeled by the EnerJ simulator contains the following enhancements to trade off precision for energy savings when appropriate:

- **Lowering the DRAM refresh rate:** We can significantly lower the refresh rate of dynamic RAM, and still achieve mostly correct operation, as shown by Flikker [37]. The EnerJ model (like Flikker) assumes that the hardware will lower the refresh rate of cache lines that store approximate values. Errors due to a lower refresh rate are modeled by the EnerJ simulator as random bit flips in the stored values.

- **Narrower mantissa widths:** Ignoring part of the mantissa can reduce energy usage of floating point calculations, as shown in [58]. Errors from mantissa rounding will manifest as increased rounding of floating point values. The EnerJ simulator models this by masking off lower order mantissa bits.

- **Processor voltage scaling:** By scaling the voltage to the logic circuits of the processor, we can save energy at the expense of occasional errors. For example, the work described in [20] shows that we can reduce the energy allocated to the circuit by 22% with an expected error rate of just 0.01%. Errors can manifest as bit flips, random values, or reusing the last value computed by the pipeline. The EnerJ simulator can model all of these, but by default uses random single-bit flips.

- **SRAM voltage scaling:** Lowering the voltage to registers and cache can also save energy, at the cost of occasional read and write failures (which manifest as bit flips), as shown in [24] and [30]. The EnerJ model assumes that approximate data may be stored in approximate registers and cache, and the simulator models this as well.

For all four of the above approximation strategies, the EnerJ simulator provides mild, medium, and aggressive levels of approximation. The higher levels are modeled by either smaller mantissas or higher error probabilities, as described in more detail in [52]. They are meant to model more aggressive approximation as well as larger potential energy savings. In some of the systems described in Chapter 5, we switch between levels based on dynamically monitored QoR.

The Truffle hardware proposed in [22] provides approximation that fits this model and provides approximate versions of most ISA instructions. The Truffle processor can switch between high-voltage (precise) and low-voltage (approximate) operation as dictated by the instruction.

The results in this thesis all assume a similar hardware model, and the monitoring work described in Chapter 5 directly uses the EnerJ language and simulator as its base. The other systems are implemented in different languages (OCaml and C), but their approximation models are based on EnerJ's model.

Chapter 3

ENERCAML: PROTOTYPING, PROFILING, AND AUTOTUNING ENERGY-AWARE APPROXIMATE COMPUTATIONS IN OCAML

3.1 Introduction

This chapter considers the challenge of quickly prototyping approximate algorithms and applications, and exploring their quality–energy tradeoffs. We tackle this problem by proposing a system that allows approximate programming and simulation in a mostly functional language known for its strengths as a base for prototyping. We then take advantage of the pure and function-oriented nature of functional code to develop a powerful profiler and autotuner that help users better manage, understand, and explore the approximate semantics of their prototyped applications. It quickly identifies portions of the code that must execute precisely to avoid severe quality of result degradation, and suggests code that can be profitably approximated. This automatic discovery of good places to approximate code is complementary to emerging approaches to verify (statically) that a program with approximation retains application-specific correctness properties [9].

This work is the first (to our knowledge) use of a mostly functional language—OCaml [45] in this case—for approximate programming, and the first autotuner that allows developers to explore quality–energy relationships in this manner. Our system runs entirely on *conventional, commodity* hardware and is intended as a tool for prototyping and investigating the potential quality of result and energy efficiency impacts of approximation on *future* approximate hardware. We show that the functional nature of OCaml lends itself well to a *code-centric* approach to approximation

(i.e., indicating approximating by annotating approximate computations—e.g., function applications/calls—rather than approximate data) and that this in turn leads to a natural implementation of an approximation autotuner. Our autotuner investigates the quality–efficiency tradeoffs of each approximate function application site. For example, if the application contained calls to an approximate dot product function, the autotuner would investigate the quality and energy efficiency impacts of individually converting each of those calls to a precise version of the dot product function. Based on this information, we are able to recommend potential code changes that improve efficiency and/or quality of result. We have implemented this system, which we call EnerCaml, as a set of modifications to the OCaml bytecode compiler and interpreter (available for download at our website [19]). The functional, code-centric approach to approximation that we take in EnerCaml could also lend itself nicely to future hardware models with coarse-grained approximation. For example, architectures with low-energy approximate cores, as we mention in Section 6.2 of Chapter 6.

We also consider an alternative to our code-centric approximation strategy for OCaml that reformulates data-centric approximation as a monad. We show how this formulation can be used to achieve the same data-centric approximation style and safety guarantees as EnerJ but with a much simpler implementation. In particular, our monad-based system can be entirely implemented as a simple OCaml module without changes to the compiler or runtime.

3.1.1 Outline

We begin by describing the user interface and implementation of the three layers of the EnerCaml system:

- The approximation layer and EnerCaml language extensions, which allow programmers to specify which portions of their code should execute approximately (Section 3.2).

- The profiling layer and its associated APIs, which track quality of result and the proportion of approximated operations (a proxy for the potential energy savings on future approximate hardware) and allow the developer to specify *how* operations should be approximated (Section 3.3).

- The autotuning layer, which helps programmers identify code changes that could improve their quality–efficiency tradeoffs (Section 3.4).

Section 3.5 discusses three sample applications that we approximated, profiled, and autotuned with EnerCaml. We describe our monadic approach to data-centric approximation in Section 3.6. Finally, Section 3.7 describes related work.

3.2 The EnerCaml Approximation Layer

The EnerCaml approximation layer adds approximation semantics to OCaml [45]. Developers create approximation by using EnerCaml-specific primitives in their OCaml programs. These primitives provide a *code-centric* model for specifying approximation, i.e., approximation is specified over blocks of code (as in Relax [17]). This is in contrast to *data-centric* models (such as EnerJ [52]), where approximation is specified for individual pieces of data. Later, in Section 3.6, we will describe how data-centric approximation can be implemented in OCaml with a monad-based approach.

This section first describes and motivates our code-centric approximation approach (Section 3.2.1). We then list the primitives for specifying approximation (Section 3.2.2). Finally, we describe the implementation of approximation in EnerCaml (Section 3.2.3).

3.2.1 EnerCaml's Approximation Model

Users create approximation in EnerCaml programs by passing a thunked code block to the primitive `EnerCaml.approximate`, which has type `(unit -> 'a) -> 'a approx`. The EnerCaml system then executes the thunk approximately and returns

the result wrapped inside an approximate type. Before using the approximately-typed result in a precise computation, the user must endorse it with a call to a function `EnerCaml.endorse` of type `'a approx -> 'a`. The use of approximate types and explicit endorsements is modeled after EnerJ. It enforces a boundary between approximate and precise computations and requires users to explicitly acknowledge every location where data crosses the boundary from the approximate realm into the precise realm.

For example, consider the following code snippet from a ray-tracer (downloaded from the website of Flying Frog Consultancy [25]), where the function `intersect` is used to determine where a ray intersects a scene:

```
let x, n = intersect zero dir (inf, zero) scene in
let g = dot n light in ...
```

To execute the intersection approximately, we simply write:

```
let x, n = EnerCaml.endorse(EnerCaml.approximate(
  fun () -> intersect zero dir (inf, zero) scene))
in
let g = dot n light in ...
```

The call to `EnerCaml.approximate` causes the EnerCaml system to simulate executing the intersection computation on approximate hardware (we discuss this in more depth in Section 3.2.3). The call to `EnerCaml.endorse` allows the values returned from the approximate intersection to be used in future precise computations. Alternatively, approximate values can be passed to future approximate computations via the `continue_approx` primitive. The `continue_approx` primitive has type `'a approx -> ('a->'b) -> 'b approx`. It takes an approximate value and a function and approximately applies the function to the value, returning another approximate value.

The current version of EnerCaml approximately executes floating point arithmetic

operations, integer arithmetic and comparison operations, and floating-point and integer array loads. Throughout the rest of this chapter, we refer to the above operations as the *approximable operations*. We discuss how this approximation can be specified and controlled (including allowing developers to define their own arbitrary approximation functions) in Section 3.3. For safety, certain operations such as memory allocation and garbage collection must always be done precisely. We assume that any future approximate hardware will have the ability to do some critical operations such as these precisely.

In programs written with the basic EnerCaml system described here, we typically see endorsements coupled tightly with approximation calls as seen in the above example. This might seem to suggest eliminating the approximate types and endorsements altogether, which would allow us to write the previous example as simply:

```
let l, n = EnerCaml.approximate (fun () ->
  intersect zero dir (inf, zero) scene) in
let g = dot n light in
if g <= 0. then 0. else
...
```

However, as we shall see in Section 3.6, the approximate data types and endorsements of EnerCaml allow us to combine it nicely with our monad-based data-centric approach. This allows us to combine code-centric and data-centric approximation in the same programs.

Code-centric approximation as described above is a natural fit for a mostly functional language such as OCaml. In a functional programming style, functions generally do not modify state or arguments. If a developer wishes to compute the result of such a pure function approximately, she or he should be able to execute the entire function approximately without worrying about any approximated effects. It is also easy to introduce approximation anywhere via our code-centric method, because in a functional language, almost everything is modeled as a function call—even small

operations like arithmetic.

Data-centric approximation, on the other hand, does not fit as well with a functional programming style. Because of the lack of mutability, functional code tends to introduce new variables instead of computing on existing variables. Thus it is not always easy to determine which data to annotate. With the code-centric approach, in contrast, we can simply annotate the function call corresponding to the computation we wish to approximate. Imperative languages are better fits for data-centric approaches because their computations tend to be more focused on the data than on the functions.

Code-centric approximation designs such as what we have described above also map naturally to hypothetical future processor designs where approximate code is executed on a lower-powered, approximate core. An underlying runtime implementation of `EnerCaml.approximate` would simply execute the thunk on an approximate core and return to the precise core when the thunk completes. While the thunk is executing, the precise core could either power down or execute code from another process or thread. Our design can also be mapped easily to processors that support per-instruction approximation such as Truffle [22]. The compiler can simply output an approximate version of every function that is called approximately and output approximate instructions when compiling this approximate version.

3.2.2 New Approximation Primitives

Table 3.1 lists the EnerCaml approximation primitives. We already described the `approximate`, `continue_approx`, and `endorse` primitives in Section 3.2.1. The `precise` primitive allows programmers to specify that certain code should always be executed precisely, even inside an approximate dynamic context. `precise` takes a thunked block of code as its argument and executes it precisely, returning the return value of the thunk. Outside of an approximate dynamic context (or directly nested inside another precise context), the `precise` primitive is simply a direct application

`approximate`	`(unit->'a) -> 'a approx`	Executes its thunked argument approximately, wraps the result in an approximate type, and returns it.
`continue_approx`	`'a approx -> ('a->'b) -> 'b approx`	Takes an approximate value and a function, and approximately applies the function to the value.
`endorse`	`'a approx -> 'a`	Transforms its approximately-typed argument into a precisely-typed return value.
`precise`	`(unit->'a) -> 'a`	Executes its thunked argument precisely, and returns the thunk's result.
`lift`	`'a approx approx-> 'a approx`	Lifts an `approx approx` type to an `approx` type.

Table 3.1: The EnerCaml approximation primitives.

of the thunk. We also provide a `lift` primitive that converts an `approx approx` type into an `approx` type. This is useful when an approximate thunk returns the result of a nested approximate thunk, resulting in an `'a approx approx` when we would prefer an `'a approx`. This could be handled by the `endorse` primitive, but that would be misleading because we are not really endorsing a flow as much as saying that multiple levels of `approx` are equivalent to a single level.

3.2.3 Implementation of the Approximate Simulator

The EnerCaml approximate simulator implementation is designed around the idea of tracking precise and approximate execution by using dual versions of each function—a precise version and an approximate version. The precise version is called whenever we apply the function in a precise context (i.e., inside precise code) or execute the thunked argument of an `EnerCaml.precise` call. The approximate version is called whenever we apply the function in an approximate context (i.e., inside approximate code) or execute the thunked argument of an `EnerCaml.approximate` call. We track the two versions of each function by adding a second code pointer to each function closure. We also create approximate versions of some of the OCaml primitives by adding the `_approx` suffix to their names and placing pointers to them in the approximate slots of their original primitives' closures. This is useful for handling approximation of floating point operations and array loads because these operations are all handled by calls to primitives in the OCaml runtime.

This approach works well for prototyping and profiling, which is the goal of EnerCaml. On real energy-saving approximate hardware, however, it may be less compelling because the extra space required for dual closures would use more energy. Thus designers of such systems should consider alternate approaches that send code to an approximate core when an approximate call is encountered or track the current approximate state (e.g., via a bit in hardware) and execute either approximate or precise instructions based on that state.

The changes to the bytecode compiler to support prototyping approximate computations were straightforward and localized. No changes had to be made to the front end of the compiler, since the EnerCaml functionality is entirely defined by calls to primitives in our new EnerCaml module. We had to modify a few data structures and instructions in the back end to track the additional code pointer (to the approximate version of the function) present in EnerCaml closures. We also had to modify the compiler to output two versions of each function. When it outputs the approximate version of a function, the compiler replaces integer arithmetic and function application bytecodes with new `_approx` versions of those bytecodes. The `_approx` versions of the integer arithmetic bytecodes specify that the interpreter should apply the integer approximation function (see below) to the result of the computation. For function applications, the `_approx` version of the bytecode specifies that the approximate code pointer should be followed (rather than the precise pointer). This includes applications of primitive functions, which results in the approximate versions of the floating point and array load primitives being called where appropriate.

We also changed the bytecode interpreter to support approximation in EnerCaml. As with the compiler, we modified a few data structures and instructions to track the dual function closures. We also added approximate versions of every function application bytecode and made them follow the approximate code pointer. We modified the code that constructs closures for primitives to search for `_approx` versions of the primitives. If found, we place the pointer to the `_approx` version of a primitive in the approximate code pointer slot of the original primitive's closure. Otherwise, we place a pointer to the standard version of the primitive in both code pointer slots (precise computation is always a legal approximation). To simulate integer arithmetic approximation, we added cases for the `_approx` version of each integer operation to the main interpreter loop. These cases all call the `approx_int_arith` routine, which in turn applies either a user-specified integer approximation function (see Section 3.3) or a default bit-flip approximator. To simulate approximation of array loads and floating

point operations, we added _approx versions of the appropriate primitives. Like the approximate integer bytecodes, these approximate primitives pass their results to a routine that applies either the default approximator or a user-specified approximator.

The final piece of the approximation layer is the implementation of the application approximation primitives (Table 3.1). The precise primitive simply passes its argument to the callback routines that are provided as part of the OCaml-C interface. For the approximate primitive, we create new approximate versions of these callback routines that follow the approximate code pointer rather than the precise code pointer. The endorse primitive does not require a C implementation. The approximate type is implemented as an abstract type (type 'a approx = 'a) in the EnerCaml module, so endorse is simply the identity function:

```
let endorse (x : 'a approx) = (x : 'a)
```

The lift primitive is identical. The continue_approx primitive is also implemented in the EnerCaml module:

```
let continue_approx (x: 'a approx) (fn: 'a->'b) =
  approximate(fun () -> fn x)
```

3.3 The EnerCaml Profiling Layer

The EnerCaml profiling layer evaluates the quality of result and potential energy efficiency gains of approximate programs prototyped with the EnerCaml approximation primitives. It also allows developers and researchers to simulate and experiment with different potential types of future approximate hardware by using EnerCaml primitives that specify *how* operations are approximated. In particular, developers can write their own OCaml approximation functions for integer, floating point, and array load operations. This section first gives an overview of the profiler and how it is used (Section 3.3.1). We then describe how users can specify their own approximation functions in Section 3.3.2. Finally, we describe our implementation of profiling inside

the EnerCaml system in Section 3.3.3.

3.3.1 Overview of EnerCaml Profiling

In its normal mode of operation (specified by running the EnerCaml interpreter with the `-profile_simple` command-line option), the EnerCaml profiler runs the specified program twice. During the first run, the profiler executes everything precisely (even code inside approximate thunks). During the second run, the profiler executes all approximable operations that appear in an approximate context approximately. After the approximate run, the profiler outputs the quality of result, the total number of operations that were approximated, and the total number of approximable operations (i.e., the number of operations that would have been approximated if the whole program was executed in an approximate context). We can use the number of approximated operations as a proxy for the amount of energy saved by approximation. The number of approximable operations represents an upper bound on the savings. We also output the percent of approximable operations that were approximated, which tells users how much of the maximum possible efficiency gains they managed to achieve. Figure 3.3.1 shows the output of a simple profile of our earlier ray tracer example [25].

To calculate the quality of result of each approximate run, the profiler must collect data from the precise run, store it, and compare it with corresponding data collected during each approximate run. The developer tells the profiler what data to collect and how to compare that data by inserting calls to the `EnerCaml` module functions `record_profile_output` and `eval_qor`. The `record_profile_output` function records its argument in a list of data to be compared across runs (as described below). In the current version of EnerCaml, this argument must be a float (i.e., `record_profile_output` has type `float -> unit`).[1] For example, if we

[1] In future versions, we would like to explore using the OCaml data marshalling library to allow the profiler to record arbitrarily-typed data. However, in the applications we have seen so far,

```
-------------Beginning precise run--------------


--------Beginning full approximate run--------
PSNR = 26.929870
Approximated operations: 149012123 (0 integer,
                         149012123 float, 0 array load)
Approximable operations: 157188070
Percent approximated: 94.798621
```

Figure 3.1: The output from running the simple profiler on a ray tracer with a 0.5% error rate.

wanted to compare the pixel values output by our ray tracer, we could insert calls to `record_profile_output` right before we output each pixel:

```
(* Cap pixel brightness at 255 for PGM format*)
let g = if (gt > 255.) then 255. else gt in
let () = EnerCaml.record_profile_output g in
Printf.fprintf pgm_file "%c" (char_of_int
                                (int_of_float g))
```

The developer then inserts a call to `eval_qor` after all of the data has been collected and passes it a quality of result evaluation function. The evaluation function should process two lists, one of precise data and one of approximate data, and output a floating point quality of result value.[2] Since we are measuring quality rather than error, higher is better (this only matters if the developer plans to use the autotuning tool, however). For example, if we wanted to use peak-signal-to-noise ratio (PSNR) as our quality of result metric for the ray tracer, we could add the following code after we output the last pixel:

```
let rec se_sum prec_l app_l =
  match prec_l, app_l with
    prc_hd::prc_tl, app_hd::app_tl ->
      (app_hd -. prc_hd) *. (app_hd -. prc_hd) +.
      (se_sum prc_tl app_tl)
  | _ -> 0.
in
let mse prec app = (se_sum prec app) /.
```

---

this has not been a significant restriction. The data we wished to record for quality of result computations was always either floating point or easily convertible to floating point (e.g., via `float_of_int`).

[2]We currently require the output to be floating-point so that the autotuner can compare the quality of result of multiple approximate runs. Future versions could allow the output to be arbitrarily typed, and simply require the developer to specify a comparison function for the chosen type. Once again, we have not found this to be a significant restriction in the examples we have investigated.

```
  (float_of_int (List.length prec)) in
let psnr prec_l app_l = 10. *. (log10
  ((255. *. 255.)/.(mse prec_l app_l))) in
EnerCaml.eval_qor psnr
```

During the precise execution, `eval_qor` simply stores the precise data collected via calls to `record_profile_output`. During the subsequent approximate execution, `eval_qor` applies the evaluation function to two lists, one containing the stored precise run data, and the other containing the data from the current approximate run. The evaluation function may assume that both lists will be temporally ordered. After executing the evaluation function, the profiler outputs the computed quality of result as well as the approximate/approximable operation breakdown mentioned earlier.

3.3.2 User-Specified Approximation

The EnerCaml profiling layer also provides primitives that allow developers to pass arbitrary approximation routines to the approximation layer. These approximation routines specify how the approximable operations should be approximated. For example, if a developer wants to simulate an approximate integer pipeline that supplies less power to low order bits, they can specify an approximation routine that flips low-order bits 10% of the time:

```
let random_low_order_flip pct i =
  (if ((Random.float 1.0) < pct) then (i lxor 1)
                                 else i)
in
EnerCaml.set_integer_approximation
  (random_low_order_flip 0.10)
```

The EnerCaml profiler supports this functionality for all types of approximable operations, which in the current version includes integer arithmetic operations, floating point operations, and integer and floating point array loads.

The EnerCaml primitives that specify approximation routines are listed in Table 3.2. They all take an approximation routine of the appropriate type (e.g., `int -> int` for the integer approximation routines) and return `unit`. If no approximation functions are specified with these primitives, the EnerCaml system uses its built-in, default approximators:

- For array loads and integer arithmetic, EnerCaml will randomly decide (with a configurable probability, defaulting to 1%) whether or not to inject an approximation error. If so, it will randomly select one bit of the result and flip it.

- For floating point calculations, EnerCaml will also randomly decide (with configurable probability defaulting to 1%) whether or not to inject an approximation error. If so, it will randomly select a result within a small, configurable window around the actual computed result.

We provide additional primitives to set the probability of these bit flip and floating point errors (as well as the size of the floating point errors) for the default approximators.

3.3.3 Profiler Implementation

The code changes required to implement the EnerCaml profiler were once again straightforward. We modified the interpreter to run the code multiple times. In simple profiling mode, there are only two runs: one fully precise run and one run with all user-specified approximation enabled. Section 3.4 describes our autotuning layer, which adds additional runs. During the fully precise run, the interpreter simply

`set_float_approximation`	`(float->float) -> unit`	Specifies the float approximation function.
`set_integer_approximation`	`(int->int) -> unit`	Specifies the integer approximation function.
`set_load_approximation`	`(int->int) -> unit`	Specifies the integer array load approximation function.
`set_load_float_approximation`	`(float->float) -> unit`	Specifies the float array load approximation function.

Table 3.2: EnerCaml primitives for specifying how operations should be approximated.

follows the precise code pointer at every function application bytecode (including the `_approx` application bytecodes). The only other change required for profiling EnerCaml programs was to implement the `EnerCaml.eval_qor` primitive. For this to work we had to ensure that the list of output data from the precise run did not get moved or deleted by the garbage collector once we started subsequent approximate runs. To accomplish this, we take advantage of the fact that `eval_qor` is called after all of the data is produced. From the user's perspective, `eval_qor` does nothing during a precise run. However, behind the scenes, we modified it to copy the precise output data list from the OCaml heap to the C heap. We store a pointer to the copied list. On subsequent approximate runs, `eval_qor` computes the quality of result by applying the passed-in QoR function to this previously-stored precise output list and the approximate output list collected during the approximate run.

3.4 The EnerCaml Autotuning Layer

The final layer of the EnerCaml system is the autotuner. The autotuner searches for alternative precise/approximate decompositions of user programs that improve their efficiency versus quality of result tradeoffs. It outputs the best potential code changes it is able to identify along with their impact on the quality of result and efficiency of the application.

Developers invoke autotuning by running the interpreter with the `-autotune` flag. When autotuning is requested, the interpreter and profiler first run the code fully precisely and then run it with all user-specified approximation (similar to the simple profiler). Next, we search for approximation that can be removed from the original specification and perform additional runs to evaluate the impact of the removal.[3] The domain of our search is the set of static call sites that were executed approximately

[3]Note that we never *add* approximation to code that was originally specified as precise because doing so may be unsafe. If the programmer wants to attempt approximation over the whole program, she or he can wrap the entire program in a call to `EnerCaml.approximate`.

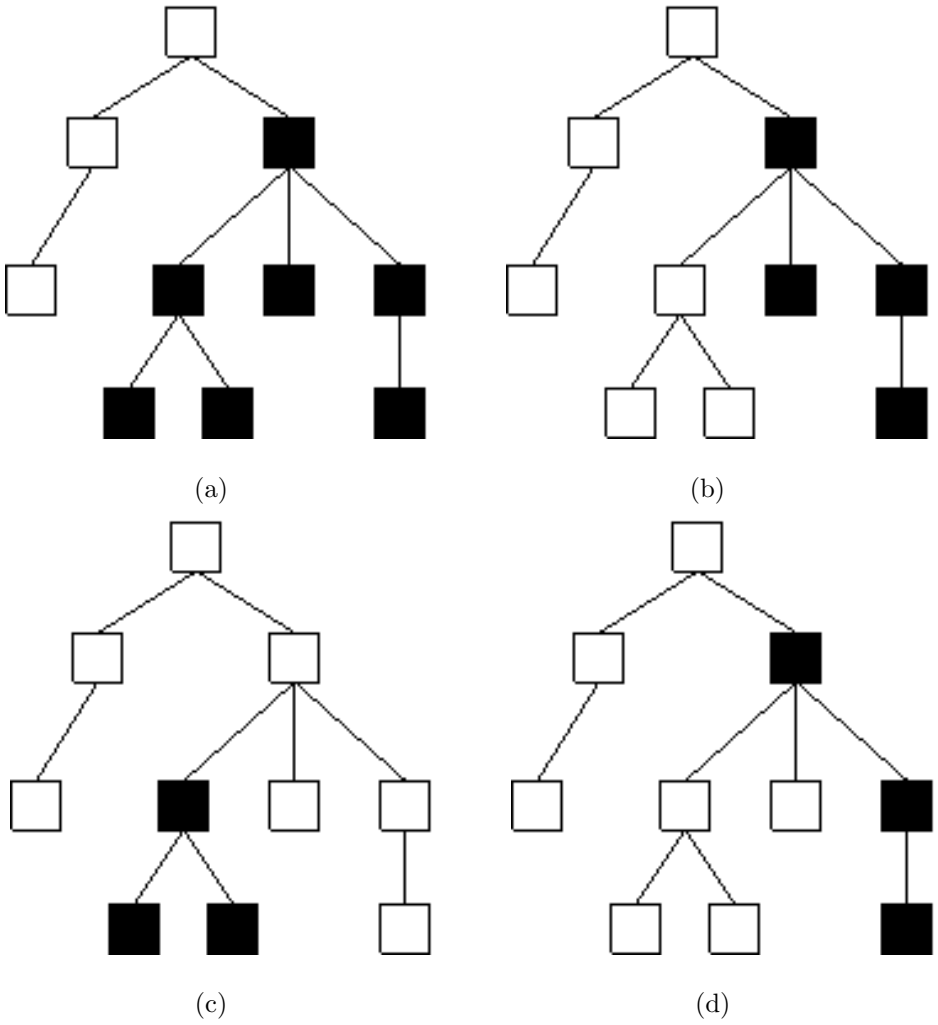


Figure 3.2: Static call trees illustrating the various strategies we use to search the precise-approximate decompositions of EnerCaml programs for improved quality of result versus efficiency tradeoffs. A black node represents an approximate function application and a white node represents a precise application. Figure (a) shows the originally specified approximation. Figure (b) shows the result of treating one of the approximate applications as if it were called inside a precise thunk. Figure (c) shows the result of narrowing the approximation to just that same call site. Finally, figure (d) illustrates the result of making two sibling call sites precise.

in the original approximate run. Figure 3.2 uses call trees to illustrate the various strategies we use to search this space. For each of the originally approximate call sites, the autotuner first attempts a run where it treats the call site as if it were enclosed in a call to the `precise` primitive (Figure 3.2b)—i.e., the call and any calls under it will follow their precise code pointers. The autotuner then attempts a run where we narrow the approximation to just that call site and its descendants (Figure 3.2c). In other words, that call site will act as if it was surrounded by an `approximate` call, but any other `approximate` calls will be removed. Finally, we search for pairs of function applications that appear in the same calling function, and perform a run that makes them both precise (Figure 3.2d). If we were to attempt all combinations of call sites, we would end up with an exponential explosion of our search space. The intuition behind looking at just the pairs in the same calling function is that these pairs are more likely to have a synergistic effect—i.e., the benefit of making them both precise might be more than the sum of the benefits of making them individually precise. For example, the two may pass data from one to the other or both may pass data to a third function.

The autotuner outputs the quality of result and approximate operation counts for each of the partial approximate runs it attempts. If one result has both better quality of result and better efficiency (as measured by approximate operation count) than another result, we say that the former result *dominates* the latter. Once all runs have been completed, the autotuner reports all runs that are not dominated by other runs. This report represents a frontier curve of the best discovered quality versus efficiency tradeoffs. In addition to reporting detailed measurements for each point on this frontier curve, the tool plots the runs graphically to help the programmer visualize the discovered space of quality–efficiency tradeoffs. Figures 3.3 and 3.4 depict the autotuner's textual and graphical output for our ray tracer example (Section 3.5.1).

```
BEST RESULTS:
Narrowing approximation to ray_trace_orig.ml,
line 16, character 10:
QoR: 37.644753, Approximated/approximable operations:
35382840/158794029 (22.282223)

Narrowing approximation to ray_trace_orig.ml,
line 36, character 13:
QoR: 32.663749, Approximated/approximable operations:
100324605/158144874 (63.438417)

Making precise ray_trace_orig.ml, line 55,
character 47:
QoR: 28.351986, Approximated/approximable operations:
148988255/157164711 (94.797524)

Making pair precise:
ray_trace_orig.ml, line 47, character 39
ray_trace_orig.ml, line 46, character 39
QoR: 28.240102, Approximated/approximable operations:
149001041/157177354 (94.798034)
```

Figure 3.3: Textual output from autotuning a ray tracer application. The tool produces a textual (excerpted here) and graphical (Figure 3.4) depiction of the best results (frontier curve) among the profiled executions. Note that the last two results listed on the left achieve nearly the same approximation levels as the original (94.8% approximate), but with better QoR (28.4 and 28.2 versus 26.9 for the original annotation).

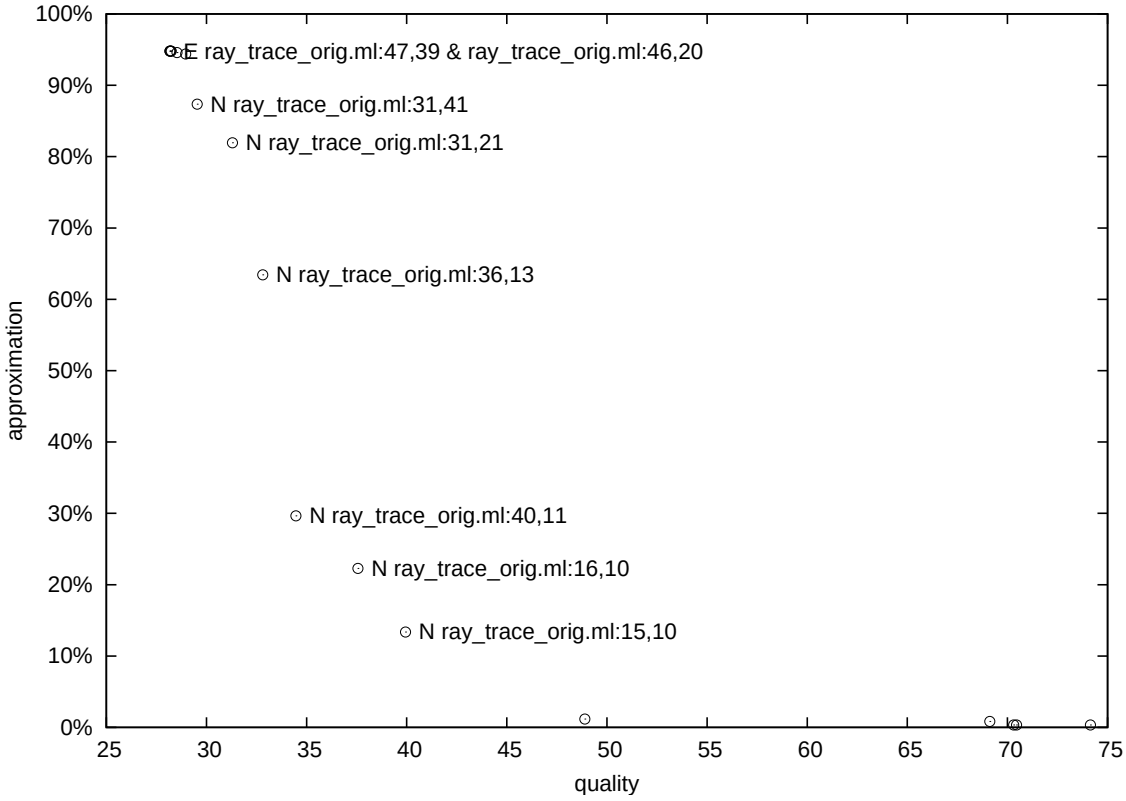


Figure 3.4: Graphical output from autotuning a ray tracer application. The tool produces a textual (Figure 3.3) and graphical represenation (depicted here) of the best results (frontier curve) among the profiled executions. The graphical depiction plots the results on axes of approximation (proportion of approximable operations executed approximately) and quality of result (here, peak signal-to-noise ratio), labeling each point with "N" for *narrowing approximation to* a call site (i.e., leaving that call site and its children approximate, but making everything else precise) or "E" for *excluding approximation at* call sites (i.e., making that call site and it's children precise, but leaving the rest of the approximation untouched).

3.4.1 Implementation of Autotuning

The code changes required to implement the EnerCaml autotuning layer were straightforward and localized. In particular, the only compiler change that was necessary was to track and record the source location of every function application bytecode. We were able to reuse code that supports the OCaml debugger to do this, with a few small additions. The bytecode offset and corresponding source location are stored in a file which is read in by the interpreter when it executes in autotuning mode. This allows the autotuner to map function applications back to locations in the source, which in turn allows it to report the changes it made for each partial approximate run in a user-readable form.

We also changed the interpreter to implement the autotuner's search strategies. In autotuning mode, our interpreter records each approximate function application that it executes during the original approximate run as well as the calling function that contains it (necessary for the final strategy that pairs function applications with the same parent). The program counters of these applications are stored in a simple hash table with no duplicates. After the fully approximate run we gather all of the application PCs into an array and use it to determine which function applications should be precise and which should be approximate in the subsequent partially approximate runs. We then check the current PC every time we execute an approximate function application bytecode in a partial approximate run to see if we need to follow the precise code pointer.

3.5 Case Studies

We used the EnerCaml system to profile and tune the approximation properties of three existing OCaml applications, none of which were written by us. This section discusses our experiences with those applications. First, Section 3.5.1 describes profiling the ray tracer application mentioned previously. Next, Section 3.5.2 discusses our

experiences with profiling an N-body simulation application. Finally, section 3.5.3 discusses a collision detection kernel.

3.5.1 Ray Tracer

Our initial experience with the EnerCaml system involved adding approximation to a ray tracer (downloaded from the website of Flying Frog Consultancy [25]). The ray tracer has two phases: scene creation and ray tracing. The scene creation phase creates a scene consisting of a number of spheres of different sizes. The ray tracing phase then generates an image by sending a series of rays at the scene.

We started by approximating both phases of the computation. To approximate scene creation, we simply added a call to `EnerCaml.approximate` around a thunk containing the call to `create`:

```
let app_scene = EnerCaml.approximate(fun () ->
    create level {x=0.; y= -1.; z=4.} 1.);;
```

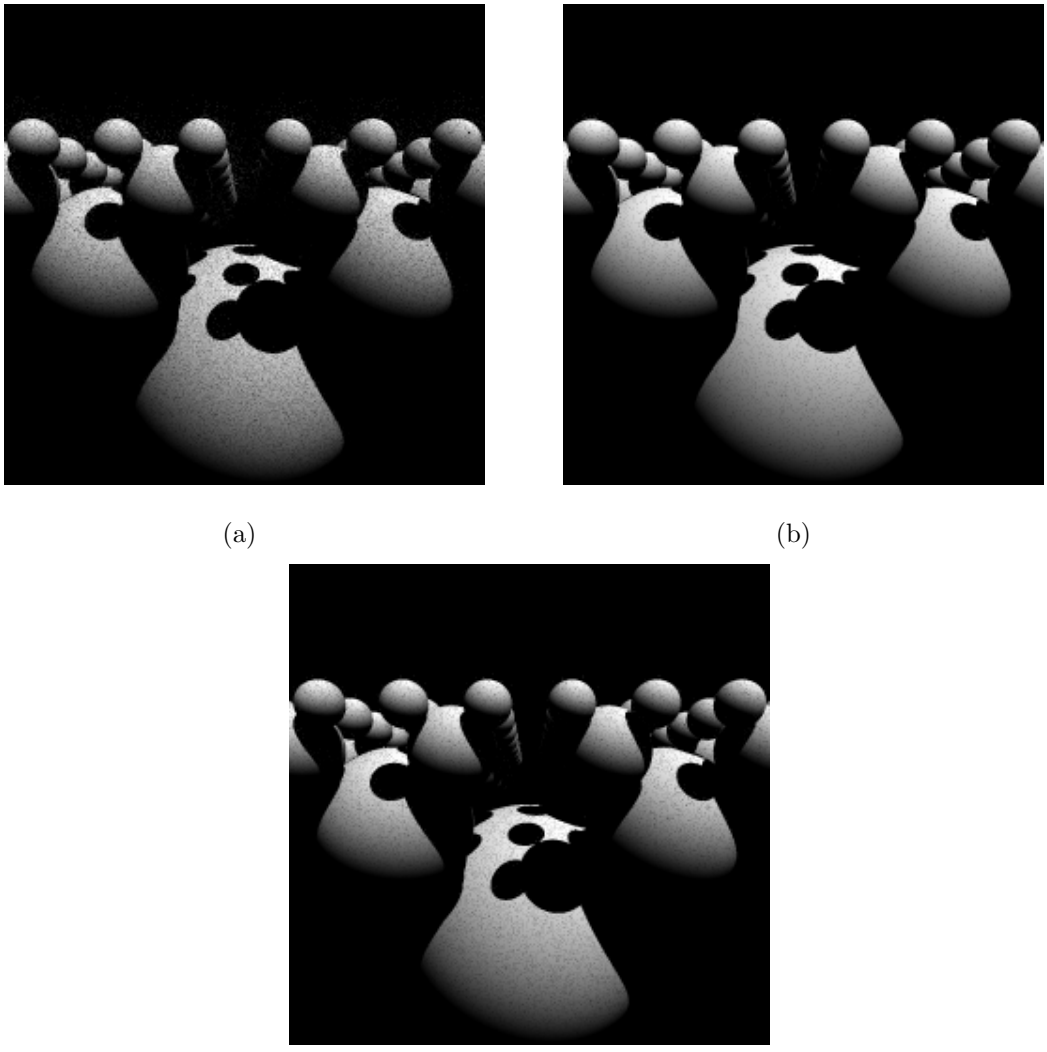
To approximate the ray tracing phase, we wrapped the calls to `ray_trace` (which traces an individual ray) inside another thunk and passed it to `approximate`:

```
let approx_g = EnerCaml.approximate(fun () ->
    ray_trace dir scene) in
```

We used the default EnerCaml approximation routines, with an error rate set to 0.5%[4] (i.e., one out of every 200 approximable operations returns an incorrect result). Figure 3.5a shows an image generated by this approximation of the the ray tracer.

We next instrumented the program for profiling and autotuning so that we could search for ways to improve the quality of the initial image (Figure 3.5a). Recall that profiling in EnerCaml involves specifying a quality of result evaluation function and adding calls to collect the data required for the evaluation. We chose peak signal-to-noise ratio (PSNR) for our quality of result. We pass the quality of result function

[4]Lower error rates did not add enough error to make the investigation interesting.



(a) (b)

(c)

Figure 3.5: The images generated by our ray tracer with various mixtures of approximate and precise execution. Image **a** (PSNR 26.9) represents the result of approximating the entire ray tracing and scene creation computations. Image **b** (PSNR 36.9) limits the approximation to a single dot product inside the `ray_sphere` function. Image **c** (PSNR 33.5) approximates the `ray_sphere` function, but executes two of its dot products precisely. It has slightly lower quality than image **c**, but almost twice as much approximation.

to the eval_qor routine, which in turn passes it lists of data from precise and approximate runs:

```
let rec se_sum prec_l app_l =
  match prec_l, app_l with
    prc_hd::prc_tl, app_hd::app_tl ->
      (app_hd -. prc_hd) *. (app_hd -. prc_hd) +.
      (se_sum prc_tl app_tl)
  | _ -> 0.
in
let mse prec app = (se_sum prec app) /.
  (float_of_int (List.length prec)) in
let psnr prec_l app_l = 10. *. (log10
  ((255. *. 255.)/.(mse prec_l app_l))) in
EnerCaml.eval_qor psnr
```

The data points for our PSNR calculation are the pixels of the output image. We collect the data as it is written to the image file:

```
let () = EnerCaml.record_profile_output g in
Printf.fprintf file "%c" (char_of_int
                            (int_of_float g))
```

After instrumenting the code, we ran it through the simple profiler to determine the quality of result and efficiency of our initial attempt at approximation. Figure 3.3.1 shows the result of simple profiling. The PSNR was 26.9, and 94.8% of the approximable operations were executed in an approximate context. We next ran our autotuner to see if we could improve on these results. Figure 3.4 shows a plot of the best results (i.e., the quality of result/efficiency frontier curve), and Figure 3.3 displays the textual output for a selection of these results. The first thing that jumps out of these results is that we can obtain better quality (PSNR of 28.4), while only giving up a very small amount of approximation by making the scene creation precise.

Intuitively, small changes in the positions of spheres can have significant impacts on the errors of some pixels because they can move the boundary between shadowed (dark) and non-shadowed (bright) pixels. These errors may not be as noticeable to a human viewer as the random errors generated by approximating rays, but they have a significant impact on our chosen metric, PSNR. These types of errors would also be more noticeable to humans in a video setting, where small shifts in the positions of objects could create inter-frame jitter. Since approximating scene creation also had a negligible impact on efficiency (most of the energy is spent on tracing the scene, not creating it), we removed it and reran the autotuner.

On our second autotuning run, the most interesting results consisted of a PSNR of 29.9 with 86.3% of approximable operations approximated, and of a PSNR of 36.9 with 22.3% of approximable operations approximated (Figure 3.5b). The 29.9 PSNR result was obtained by narrowing the approximation to just the call to the `ray_sphere` function (which computes the first intersection of a ray and a sphere). The 36.9 PSNR result was obtained by narrowing the approximation to a particular dot product computation inside of the `ray_sphere` function. These results led us to focus our approximation efforts on the ray-sphere intersection code. We moved the approximation primitive in the ray tracer code to just the `ray_sphere` call site, and reran the profiler. This gave us a number of new interesting points along our frontier curve, including a PSNR of 33.6 with 41.8% approximation, a PSNR of 32.9 with 50.7% approximation, and a PSNR of 31.5 with 64.1% approximation. All three of these results were obtained by making either individual calculations or pairs of calculations inside `ray_sphere` precise. Our favorite result, with PSNR 33.5 and 41.8% approximation, is shown in Figure 3.5c. It has slightly lower quality than the PSNR 36.9 result in Figure 3.5b, but nearly twice the number of approximated operations.

It would have required significantly more effort to characterize the effects of approximation without the assistance of our profiling and autotuning tool. The auto-

tuner allowed us to quickly remove scene creation from consideration due to its poor tradeoff between quality and efficiency. It then pointed us to the importance of the `ray_sphere` function and allowed us to focus our efforts there.

3.5.2 N-Body Simulation

The next application that we looked at was an N-body simulation (downloaded from the Computer Language Benchmarks Game website [59]). We started by adding a simple quality of result metric that calculates the inverse of the average error. The simulation first initializes the N-body system with a call to `offset_momentum` and then calls `advance` in a loop to advance the state of the simulation one step at a time. We wrapped both calls in approximate thunks:

```
EnerCaml.approximate
  (fun () -> offset_momentum bodies);
...
for i = 1 to n do
  EnerCaml.approximate
    (fun () -> advance bodies 0.01)
done;
```

When we first ran our approximated N-body simulation, it threw an index out of bounds exception. The offending array indices were calculated by integer arithmetic that was approximated. We could have chosen to wrap the relevant calculations in a precise thunk, but we instead decided to see if our autotuner could help us. When the autotuner encounters an uncaught exception on one run, it simply terminates that run (without recording it as a potential best result) and continues to explore alternate approximations of the code. Another possibility that we intend to explore further is to modify future versions of the EnerCaml runtime to convert out-of-bounds array references in approximate code to in-bounds references.

Initially, our autotuner was not able to tell us very much because the only function applications it identified were the two outer-level calls to `offset_momentum` and `advance` that we had wrapped in approximate thunks. We looked at the code and discovered that the simulation code was written in a very imperative style. A doubly-nested loop over the bodies calculates the effect of each body on every other body. We were quickly able to convert the code to the more functional style that our autotuner is designed for by identifying various subcomponents of the calculation, and wrapping them in function calls. When we reran the autotuner, we found that two of the subcomponents of the calculation could be profitably approximated with no out-of-bounds exceptions *and* very low impact on the quality of result:

```
QoR = 5562.919330
Approximated operations: 45000007
Approximable operations: 184000490
Percent approximated: 24.456460
...
QoR = 7436.822960
Approximated operations: 45000007
Approximable operations: 184000490
Percent approximated: 24.456460
```

Both of these are significantly better than the original approximation. With integer approximation temporarily turned off (via setting its probability to 0) to avoid the exceptions, the original approximation had a QoR of 0.009556—so low as to be unusable (although it did achieve 94.3% approximation). We also tried approximating both of the identified computations to see if we could still get good quality of result with a larger fraction of operations approximated. Our results were promising:

```
QoR = 3821.292285
Approximated operations: 90000007
```

```
Approximable operations: 184000490
Percent approximated: 48.912917
```

In the case of the N-body simulation code, the autotuner allowed us to work around the initial errors that were caused by attempting to approximate calculations that needed to be performed precisely. It then allowed us to identify (after straightforward code modifications) portions of the simulation calculation that could be profitably approximated without significantly impacting the quality of result. The whole process took roughly one hour for someone unfamiliar with the code.

3.5.3 Collision Detector

Our final example is a simple collision detection kernel (downloaded from [46]) that checks whether or not two triangles in 3D space intersect each other. Our quality of result metric calculates the percentage of the intersection tests where we correctly detect whether or not the triangles intersect. Based on our experiences with the previous examples, we did not attempt to approximate the initialization. Instead, we just surrounded the call to the intersection routine with an approximate thunk. We then passed the result of the intersection routine to a function that parses the result and records it for the profiler:

```
let intersects = EnerCaml.endorse(
  EnerCaml.approximate(
    fun () -> tri_tri_intersect tri1 tri2))
in
record_output_coll intersects
```

As usual, we started by running the simple profiler to get a baseline:

```
Percent correct = 97.810000
Approximated operations: 4038704 (0 integer,
          1830788 float, 2207916 array load)
```

```
Approximable operations: 4298283
Percent approximated: 93.960868
```

Our initial results were reasonable—97.81% of collisions were correctly detected and almost 94% of approximable operations were approximated. We then ran the autotuner to see if we could do even better. Most of the interesting results along the frontier curve involved making some combination of the function applications from four different source lines approximate. These four lines can be split into two pairs. The first pair test whether all three points of one triangle lie on the same side of the plane of the other triangle (indicating no intersection):

```
if ((sign da1) = (sign da2) &&
    (sign da2) = (sign da3)) then
  NoIntersection
...
if ((sign db1) = (sign db2) &&
    (sign db2) = (sign db3)) then
  NoIntersection
...
```

The other pair compute the normals of the planes containing the two triangles, which is an essential input to the computation we just described:

```
let na = vnormal a.(0) a.(1) a.(2)
and nb = vnormal b.(0) b.(1) b.(2) in
```

We experimented with making these computations precise. When we made both of the plane normal calculations precise, our quality of result increased to 98.88% correct, but our approximation percentage dropped a bit, to 67.1%. When we instead made the no-intersection checks precise, our quality of result did not increase by as much, only rising to 97.94%. However, our approximation was almost unchanged at 93.0%. When we combined both changes we were able detect 98.93% of collisions

correctly and still approximate 66.1% of the approximable operations. Compared to the original annotation, we were able to eliminate over 51% of the errors while losing less than 30% of the approximation. This whole process took under an hour for someone unfamiliar with the code.

3.6 A Monad-Based Approach to Data-Centric Approximation

The EnerCaml system described above provides a code-centric approach to approximation for OCaml programs. As we discussed earlier (Section 3.2.1), this approach works well for mostly functional programs. However, OCaml also supports an imperative programming style and mutable references. For imperative code, a data-centric approach to approximation (such as EnerJ [52]) is often preferable. This section shows how monads can be used to implement data-centric approximation in OCaml with no changes to the compiler, runtime, or interpreter. Section 3.6.1 describes our monad-based approach. Section 3.6.2 gives a short example.

3.6.1 Approximation Monads

We can formulate data-centric approximation as a monad. OCaml has no special language support for monads, but one can still define them with the module system. We let the monadic value store both an approximate value and an approximation function that we apply every time we read the value:

```
module ApproxMonad =
  struct
    type 'a approx = Approx of 'a * ('a->'a)
    ...
```

We can then define the standard `bind` and `return` functions as follows:

```
let bind (Approx(x, appr)) (f:'a -> 'b approx) =
  f (appr x)
```

```
let return x fn = Approx(x, fn)
```

The `bind` function applies the approximation function `appr` to the data `x` prior to passing it to the function `f`. This preserves the invariant that the stored approximate data may only be accessed via the approximation function. The `return` function creates a new monadic value. We store the data `x` and the approximation function `fn` but we do not immediately apply the approximation function. The approximation function will be applied before any use of the data, so if we were to apply it here as well, it would be applied *twice* before any use.

As with EnerCaml and EnerJ [52], we require an explicit endorsement to use approximate data in a precise calculation:

```
let endorse (Approx(x, appr)) = appr x
```

As was the case with `bind`, `endorse` must call the approximation function `appr` on the approximate data `x` before returning it. This is necessary to maintain the invariant that the data is accessed only via the approximation function.

We can also use `bind` and `return` to define a number of other useful functions for dealing with values in our approximate monad. A monad `join` function is useful for our monad because an `'a approx approx` should be equivalent to an `'a approx`:

```
let join apm = bind apm (fun x -> x)
```

As usual, the `join` function converts a doubly approximate type to a single approximate type (i.e., it has type `'a approx approx -> 'a approx`). A map function is also useful to allow us to apply arbitrary functions to our approximate data. We define our `map` in terms of `bind` to ensure that the access-via-approximation function invariant is obeyed. We need to supply an approximation function for the result of the map because the return type of the mapped function may not be the same as the input type. Thus the approximation function for the input may not be applicable to the output.

```
let map x f app_fn =
  bind x (fun x -> return (f x) app_fn)
```

As expected, this has type `'a approx -> ('a -> 'b) -> ('b -> 'b) -> 'b approx`. The arguments are the input monadic value of type `'a approx`, the mapped function of type `'a -> 'b`, and a new approximation function for the output type of the mapped function (`'b -> 'b`). Our `map` returns a `'b approx` containing the new approximation function and the result of applying the mapped function to the approximated input. We also provide a means for reusing the input's approximation function when the input and output types of the mapped function are identical. We call this version `map_st` ("st" stands for same type):

```
let map_st (Approx(d, appr) as x) f =
  bind x (fun x -> return (f x) appr)
```

This has the expected type `'a approx -> ('a -> 'a) -> 'a approx` (the third argument from `map` is omitted because we are reusing the approximation function from the input monadic value). We can also define bind and map operations for two-argument functions:

```
let bind2 x1 x2 f =
  bind x1 (fun x -> bind x2 (f x))


let map2 x1 x2 f appr =
  map x1 (fun x -> endorse (map x2 (f x)
                                   (fun x -> x)))
      appr
```

Note that the presence of the `endorse` call in the above definition of `map2` does not result in an extra layer of approximation because we apply the endorsement to an intermediate value whose approximation function is the identity function. Thus, the approximation introduced by the endorsement is a no-op. We execute two "real"

approximations, one on x1, and one on x2, when we execute their respective map calls. Once again, map2 requires an approximation function as an argument because the output type of the mapped function may not be identical to either of the input types. We provide alternate implementations for the case where the output type does match one of the input types. For mapped functions of type 'a -> 'b -> 'a, we provide:

```
let map2_st1 (Approx(d, appr) as x1) x2 f =
  map x1 (fun x -> endorse (map x2 (f x)
                                   (fun x->x)))
      appr
```

and for mapped functions of type 'a -> 'b -> 'b, we provide:

```
let map2_st2 x1 (Approx(d, appr) as x2) f =
  map x1 (fun x -> endorse (map x2 (f x)
                                   (fun x->x)))
      appr
```

We can now use this map functionality to define approximate operations that compute approximately over our approximate data. We first define a helper function:

```
let m2_and_approx (Approx(x, appr) as x1) x2 f =
  map2_st1 x1 x2 (fun x y -> appr (f x y))
```

The m2_and_approx function performs a two-argument map with a mapped function whose first argument and return type match, and applies an approximation function to the result of the mapped function. Note that the map call will also apply approximation functions to the input data as it is being read in. This is intended to simulate the effect of performing an operation approximately over two values stored in approximate memory. We use this helper function to define approximate operations such as integer and floating point arithmetic:

```
let app_plus   x y = m2_and_approx x y (+)
```

```
let app_sub    x y = m2_and_approx x y (-)
let app_mult   x y = m2_and_approx x y ( * )
let app_div    x y = m2_and_approx x y (/)
let app_plus_f x y = m2_and_approx x y (+.)
let app_sub_f  x y = m2_and_approx x y (-.)
let app_mult_f x y = m2_and_approx x y ( *.)
let app_div_f  x y = m2_and_approx x y (/.)
```

We can also define infix syntactic sugar for the above operations:

```
let ( +$ )  a b = app_plus   a b
let ( +$. ) a b = app_plus_f a b
let ( -$ )  a b = app_sub    a b
let ( -$. ) a b = app_sub_f  a b
let ( *$ )  a b = app_mult   a b
let ( *$. ) a b = app_mult_f a b
let ( /$ )  a b = app_div    a b
let ( /$. ) a b = app_div_f  a b
```

It is also useful to be able to *flatten* approximate values that contain data structures (e.g., an approximate monadic value containing a list) into structures that instead contain monadic values (e.g., a list of approximate monadic values). This can arise when we map functions that produce structures from simpler input. For example, if we were to map a function `n_copies` that produces a list containing n copies of an integer, we would end up with an output of type `int list approx` where we likely want an `int approx list`. We can write simple functions to perform these types of transformations for different data structures. For example, we added the following code to `ApproxMonad` to flatten lists:

```
let flatten_list (Approx(lst,f)) appr =
  List.map (fun x -> return x appr) lst
```

As expected, this function has the type `'a list approx -> ('a -> 'a) -> 'a approx list`. Note that we need to supply a new approximation function (`'a -> 'a`) because we are changing the type of the data contained in the monad from an `'a list` to an `'a`.

This monad-based approach for OCaml provides many of the same capabilities that EnerJ [52] provides for Java. The programmer can mark data as approximate and perform approximate operations on it. The type system ensures that approximate data does not flow into precise variables without an explicit endorsement from the programmer. Unlike EnerJ, however, we were able to do all of this with a simple module written entirely in OCaml. No changes to the compiler, runtime, or interpreter were required.

Our current monad-based data-centric approximation system does not include a profiling capability like the code-centric EnerCaml system that we discussed previously. In Section 6.2, we discuss ideas for implementing an approximation profiler and autotuner for approximate monads.

3.6.2 Example

To demonstrate this monad-based approach, consider our previous ray tracer example (from [25]). As we mentioned in Section 3.2.1, data-centric approximation is a better fit for an imperative programming style than a functional programming style. Thus, to demonstrate our data-centric, monad-based approach we consider the one imperative piece of the ray tracer: the loop which computes the pixel values for the generated image. Each iteration of the loop defines a mutable reference `g` that sums the contributions of every ray to the pixel we are currently considering:

```
for y = n - 1 downto 0 do
  for x = 0 to n - 1 do
    let g = ref 0. in
    for dx = 0 to ss - 1 do
```

```
      for dy = 0 to ss - 1 do
        let aux x d =
          float x -. float n /. 2. +.
          float d /. float ss
        in
        let dir = unitise {x=aux x dx;
                           y=aux y dy;
                           z=float n}
        in
        g := !g +. ray_trace dir scene;
      done;
    done;
    (* Output pixel value g to image file *)
    ...
  done;
done;
```

A data-centric approximation approach allows us to model the effect of storing the mutable reference g in approximate memory and performing operations on it approximately. We simply make g an Approx with an appropriate approximation function, and route all operations on g through the appropriate monadic operators:

```
let float_error pct1 pct2 i =
  (if ((Random.float 1.0) < pct1)
   then (i *. ((Random.float (pct2 *. 2.)) +.
               1. -. pct2))
   else i);;

let appfn = float_error 0.001 0.01;;

for y = n - 1 downto 0 do
```

```
  for x = 0 to n - 1 do
    let g = ref Approx(0., appfn) in
    for dx = 0 to ss - 1 do
      for dy = 0 to ss - 1 do
        let aux x d =
          float x -. float n /. 2. +.
          float d /. float ss
        in
        let dir = unitise {x=aux x dx;
                           y=aux y dy;
                           z=float n}
        in
        g := !g +$. (return (ray_trace dir scene)
                             appfn);
      done;
    done;
    (* Output pixel value g to image file *)
    ...
  done;
done;
```

We define `appfn` to be a float approximation function that returns the precise result 99.9% of the time, and returns a result approximated by up to 1% the rest of the time. We then replace the mutable float reference `g` with a mutable reference to an approximate monadic value containing a float (and the float approximator `appfn`). Finally, we wrap the float returned from `ray_trace` in an approximate monadic value and use our floating point approximate addition routine `(+$.)` to add it to `g`.

3.7 Related Work

Some language-level techniques seek to help programmers mitigate the negative effects of relaxed semantics to achieve a reasonable balance between execution quality and resource consumption. These techniques are complementary to EnerCaml's autotuner, which seeks to identify at a high-level (and during the protoyping phase of development) which portions of an application have the most potential for good energy–quality trade-offs. For example, Carbin et al. [9] propose a proof system for verifying programmer-specified correctness properties in relaxed programs. Misailovic et al. [39] use probabilistic reasoning to prove accuracy bounds on relaxed program transformations. EnerJ [52] provides a simple noninterference guarantee. These techniques are all static and conservatively bound imprecision. Programmers writing to a relaxed programming model should use these static techniques in tandem with dynamic tools like Enercaml (and the others we propose in Chapters 4 and 5) to obtain an empirical picture of quality loss.

Quality-of-service profiling [40] is a similar technique that identifies code that has little influence on application output quality and that the programmer should consider relaxing to improve performance. In contrast, our tool uses a priori programmer annotations to identify approximate portions of programs that should be made *more* accurate to achieve a desired QoR level. EnerCaml is a closed-loop system that suggests specific code modifications to achieve better energy–quality tradeoffs.

The SAGE system [51] implements approximation for CUDA GPU kernels. On of their two phases is a runtime tuning phase that bears some similarities to our autotuning. Their system target a much later phase of the development workflow than ours—they are looking at runtime tuning of deployed applications in a specific target environment, rather than exploration of approximation strategies in a prototype. Thus, their tuning focuses on tweaking approximation parameters, rather than exploring which parts of a computation are most amenable to approximation.

Ansel et al. [4] describe the PetaBricks language extensions and compiler features that allow developers to auto-tune what they call *variable-accuracy algorithms*, a category which includes approximate algorithms. Their focus is on performance rather than the quality–energy trade-offs we are looking at, but some of the language extensions they propose may be useful in the energy-aware approximate computing arena as well.

This chapter focuses on the use of functional languages for profiling and prototyping approximate computations, but we also briefly touch on hardware considerations for implementing code-centric approximation for a language like EnerCaml. We hope to pursue this further in future research. Along these lines, there is a long history of research at the intersection of hardware design and functional languages. In fact, one of the predecessors of the well-known International Conference on Functional Programming (ICFP) was the FPCA conference: Functional Programming and Computer Architecture. More recently, the Lava project [6] used Haskell to design circuits. HML [36] (or Hardware ML) is a hardware description language based on Standard ML. The FLaSH project [41] developed their own functional specification language, SAFL, that could be compiled directly (behaviorally) into hardware. Frankau and Mycroft [26] developed an extension to SAFL called SASL that added support for stream processing.

Our implementation of EnerCaml borrows from techniques employed by other projects. Our dual closure approach to tracking the approximate versus precise context is derived from previous work that we did—specifically, the dual closures that we used to track atomicity context and provide transactional memory semantics for OCaml in our AtomCaml [50] project. That work is not included in this thesis as it is not germane to our topic, but it formed the basis for many of the ideas that went into the EnerCaml system. Others have discussed the use of monads in OCaml libraries. The `pa_monad` system [11] adds Haskell-like monad syntax to OCaml. The lwt library (available at [38]) and the async library from Jane Street (available at [3])

are both based on monads. F# [56] has built-in support for monads (which it calls *workflows*).

Chapter 4

OFFLINE ANALYSIS OF APPROXIMATE PROGRAMS

4.1 Introduction

The previous chapter proposed a tool for prototyping approximate algorithms and exploring their QoR–energy tradeoffs (Chapter 3). This chapter focuses instead on a later stage of approximate application development: quality testing and debugging. In particular, it presents two dynamic, offline tools that can provide a more fine-grained look at the nature of the QoR of approximate applications. The next chapter will discuss an even later stage: post-deployment online quality monitoring and real-time adjustments. All of these tools are important pieces of an approximate programming ecosystem. The prototyping tools allow developers to explore algorithms at a high-level, and determine the key quality–energy tradeoffs. The fine-grained offline tools (discussed in this Chapter), while too heavyweight for usage in deployment (the costs would more than overwhelm the savings from approximation), are excellent tools for pre-deployment debugging and understanding of quality issues in the application. They help programmers better understand where they can safely use approximation. The online monitoring tools, on the other hand, are lightweight enough to run in deployed code and can allow applications to constantly adjust approximation levels or correct erroneous results when faced with quality issues that arise post-deployment (due, for example, to unanticipated program inputs or variations in approximate hardware).

The tools proposed in this chapter instrument programs to determine the critical data locations and code points that have the most impact on QoR. Our first tool tracks approximate data-flow, and allows developers to determine which results and data

depend on the largest number of approximate operations. Since each approximate operation has a non-zero chance of returning a wrong answer, this is a good proxy for the likelihood that a result or value may be incorrect. Our second tool tracks the number of times each approximate operation executes, as well as the number of times it produces an incorrect result. If we run the instrumented program multiple times, we can calculate which operations and errors are most correlated with QoR. This can help developers identify key operations that impact the final QoR of the application. Both tools of our tools are implemented as LLVM [31] compiler passes that add code instrumentation to C and C++ programs extended with EnerJ-style qualifiers, combined with runtime libraries that collect and output the data generated by the instrumentation.

This rest of this chapter is organized as follows:

- Section 4.2 describes our dataflow tracking tool.
- Section 4.3 presents our correlation finding tool.
- Section 4.4 discusses the usage and APIs for our tools.
- Section 4.5 describes some of the most interesting implementation details of our tools.
- Section 4.6 discusses the use cases we used to validate the usefulness of our tools.
- Section 4.7 describes related work.

4.2 Dataflow Instrumentation

In many approximate applications, there can be a wide variance in the number of approximate operations that flow into the computation of different results. Results

that depend on more approximate operations will typically have a higher chance of being incorrect. However, the number of approximate operations that go into the computation of these results is not always proportional to the savings provided by computing the results approximately. In such cases, the approximate computation of these results can have far greater impact on the probability of poor QoR (if they depend on many approximate operations) than would be justified by the amount of savings they provide. For example, an image transform may compute a scaling factor for its output by traversing the input and determining the difference between the maximum and minimum pixels (cf. the Sobel filter application in Section 5.7). If the input image is approximate data, computing the minimum and maximum values will require a large number of approximate operations. If any of these operations go wrong, then the scaling factor (and hence *every* output pixel) will be incorrect. Thus, even though this computation may comprise only a small portion of the energy usage of the program, it may have a very large impact on expected QoR. We may thus be better off executing it precisely.

This type of scenario motivates our dataflow instrumentation tool. Given an approximate operation O_1, and a second operation O_2 with inputs $i_1, ..., i_n$ and result R, we say that O_1 *flows* into R if $O_1 = O_2$ *or* if O_1 flows into one of $i_1, ..., i_n$. Our tool is built on a version of LLVM enhanced with approximate versions of the IR operations for arithmetic and memory access. We use a compiler pass to add code after every IR operation to compute the number of approximate operations that flow into the result of the operation. For each IR data location (e.g., user variable, memory address, SSA name), we create a shadow counter that tracks the approximate flow into the result held in that location. For most operations (everything except loads, stores, and calls), we simply sum the shadow counters associated with the operands, add 1 if the operation is approximate, and assign the result into the shadow counter for the result of the operation. Note that if our IR is in SSA form, it may be possible to optimize away much of the arithmetic generated by this process via constant propagation,

constant folding, and dead store elimination.[1] For store operations, we add code that stores the counter associated with the store's value operand into a shadow memory (e.g., a hash table keyed on memory addresses—see Section 4.5.1). If the store is approximate, we add one to the counter value that we place in the shadow memory. For load operations, we look up the load's address in our shadow memory, retrieve the associated counter, and add one if the load operation is approximate. Finally, for calls, we utilize a shadow parameter-return stack to keep track of counts for function parameters and returns. Prior to the call, we push the counter values corresponding to every parameter onto the stack. At function entry, we pop these counter values off, and assign them to the parameter's shadow counter. At function return, we push the counter value associated with the return value onto the stack. The caller than pops this value and assigns it to the shadow counter associated with the location storing the call result.

For example, consider the following code snippet:

```
%3 = %1 + %2       ; approximate
%store %3, %x      ; approximate
%4 = load %x       ; approximate
%5 = call foo(%4)
```

Our instrumentation would change this to the following. Note that for clarity in this example we denote the shadow counter of a location `x` as `x_shadow`, but in reality the instrumentation just introduces a fresh variable:

```
%3 = %1 + %2       ; approximate
%3_shadow = 1 + %1_shadow + %2_shadow
%store %3, %x      ; approximate
%6 = %3_shadow + 1
%call _recordShadowMemory(%x, %6)
```

[1]Performance of the instrumented code is not a high order goal of this work, but anything we can do to reduce the time it takes to use the tool helps increase the likelihood of user adoption.

```
%4 = load %x      ; approximate
%4_shadow = call _fetchShadowMemory(%x) + 1
%call pushPRStack(%4_shadow)
%5 = call foo(%4)
%5_shadow = call popPRStack()
```

As mentioned above, if `%3` is not used again, standard optimizations could eliminate `%3_shadow` and replace the first four lines of the instrumented code with:

```
%3 = %1 + %2      ; approximate
%store %3, %x     ; approximate
%6 = %1_shadow + %2_shadow + 2
```

In order to utilize the information gleaned from this dynamic flow analysis, we provide an API for developers to access the shadow counter values of user variables and expression results. This API is described in Section 4.4.

4.3 Correlation Instrumentation

In many approximate applications, particular *approximate code points* (operations that are executed approximately) are more likely to cause poor quality of result than others. For example, a code point that impacts every pixel may have much more impact than one which impacts only a single pixel. Our second offline approach, *correlation instrumentation*, helps developers identify those critical points by tracking the execution and error frequencies of every approximate code point during multiple program executions with varied QoRs. The result is a series of correlation vectors, where each vector consists of an application execution QoR and a series of execution and error counts for every approximate code point in the application. Off-the-shelf tools can then determine which coordinates of the vector are most highly correlated with QoR.

Our instrumentation proceeds by adding two counters for every approximate code

point (approximate LLVM byecodes in our implementation). The first counter simply counts the number of times the code point executes. The second counter is an error counter. We assume an approximation model where every approximate operation has some probability of returning an incorrect result. For most approximate operations (except memory references), we reexecute the operation precisely and compare the precise and approximate results. For stores, we precisely store an identical value into a shadow memory (e.g., a hash table keyed on memory addresses—see Section 4.5.1), as well as a pointer to the store's error counter. At loads, we look up the loaded address in the shadow memory and compare the result stored there with the approximately loaded value. If there is an error, we can charge the error counters of either the store (obtained from the shadow memory), the load, or both. This decision can be programmer-driven or chosen by the tool implementation.[2] We are assuming a model where approximate loads and stores always access an approximate memory and all accesses to approximate memory are approximate. If these conditions do not hold, we would instead need to utilize the shadow memory for *every* store, rather than just approximate stores.

For an example of our instrumentation process, consider the following intermediate instructions:

```
%3 = %1 + %2      ; approximate
%store %3, %x     ; approximate
%4 = load %x      ; approximate
```

We would instrument this as:

```
%execCnt1 += 1
%3 = %1 + %2      ; approximate
%5 = %1 + %2
%6 = cmp equal %3, %5
```

[2]Note our design assumes that the only side effecting approximate operations are stores, but this can easily be adapted to other models.

```
%7 = select %6, 0, 1
%errorCnt1 += %7
%execCnt2 += 1
%call recordShadowPair(%x, %3, &%errorCnt2)
%store %3, %x      ; approximate
%execCnt3 += 1
%4 = load %x      ; approximate
%8 = call checkShadowMemory(%x, %4)
%errorCnt3 += %8
```

In this example, we have chosen to charge both loads and stores in the event of errors in the approximate store. The call to `checkShadowMemory` will return 1 if the value found in the shadow memory does not match the value passed in by the second parameter, or 0 otherwise. The next line of the instrumentation uses this to update the load's error counter. To charge the store, `checkShadowMemory` will directly increment the store's error counter when it looks up the address in the shadow memory (the `recordShadowPair` routine records both the correct memory value and the store's error counter in the shadow memory).

This approach is not context-sensitive—we are merely tracking correlations to individual program counters, rather than tracking correlations to code points in particular calling contexts. Nothing conceptually prevents us from adding context sensitivity, but we have not yet found it necessary for any of the applications we studied. In addition, the benefits of context-sensitivity are highly application-specific. In some cases, adding context sensitivity could help us identify more precise correlations by identifying code points that are highly correlated to quality *only in specific calling contexts*. In other cases, however, context-sensitivity could obscure important correlations by splitting error-causing code points into multiple components of the correlation vector.

We also provide APIs that let developers create and store execution and error vectors along with associated QoR values. This can be done for the computation as

a whole or for subcomputations. These APIs are described in Section 4.4.

4.4 APIs and Usage

This section describes the usage of our offline instrumentation-based QoR tools. Our offline tools are implemented as LLVM compiler passes that add the appropriate instrumentation to the LLVM IR. We also provide APIs to access and output the counter data produced by the instrumentation.

Our dataflow instrumentation tool tracks the number of approximate operations that dynamically flow into the computation of every result. Users may access these results via the counters associated with each user variable and expression result in the program. The following varargs function dumps the data counters associated with the `count` variable arguments to the file `fname`:

```
void dumpDataCounters(char *fname,
                      int count, ...);
```

The variable arguments in the argument list can be either user variables or expressions (should the developer wish to track the number of approximations involved in computing an expression rather than just a variable). Developers merely need to pass the variables by value. The compiler automatically replaces the variables with their associated data flow counter values, and the library function then simply dumps them to the named file. Expressions work similarly: the instrumentation pass automatically generates a temporary dataflow counter for the result of every operation in the program, and the compiler replaces any expressions in the above argument list with their associated counter. Developers can also access the counter values with the following routine:

```
int dataCounterSum(int count, ...);
```

This returns the sum of the data counters associated with the `count` variable arguments (note that we can call this with `count` of 1 to access the value of a single

counter).

Our correlation instrumentation keeps counters which track the number of executions of, and errors in, each approximate operation in the IR. We primarily access these via the following APIs:

```
void recordAndResetVector(double qor);
void dumpVectors(char *fileName);
void appendVectors(char *fileName);
```

The first routine records a vector consisting of all of the execution and error counters, as well as a floating-point QoR. It also resets the counters to 0, in case we wish to track correlations and quality of results for multiple iterations of a calculation in the same execution. The latter two routines dump the stored vector or vectors to the specified file. The former overwrites the file and the latter appends to it (in case we are attempting to track correlations over multiple executions). In all cases, the vectors are output with descriptive coordinate labels that describe which source line each vector coordinate corresponds to as well as whether it is an execution or error counter. Off-the-shelf tools can then be used to calculate the correlation between each counter and the application's QoR.[3]

4.5 Implementation Issues

This section describes a couple of the most interesting details of our QoR tool implementations. First, Section 4.5.1 describes implementing shadow memories for our offline approaches. Then Section 4.5.2 describes how we chose appropriate ordering of the instrumentation passes relative to other LLVM passes.

[3]In our experiments, we simply imported these vectors as a table into an open-source spreadsheet application and used it to calculate the correlation between each counter column and the QoR column.

4.5.1 Shadow Memories

Both of our offline tools require a shadow memory. Dataflow instrumentation uses the shadow memory to track dataflow counts across loads and stores. When a value is stored, we store its current approximate dataflow in the corresponding location of the shadow memory. Similarly, when a value is loaded, we look up its approximate dataflow in the shadow memory. Correlation instrumentation, on the other hand, uses the shadow memory to track the actual values stored in approximate memory. When we load from an address in approximate memory, we check the corresponding shadow memory address to see if the loaded value is correct (if it is not correct, we increment the appropriate error counter).

Both forms of shadow memory are implemented as hash tables keyed on the real memory address. The values are either the dataflow counter for dataflow instrumentation or the stored value and the address of the store's error counter for correlation instrumentation.[4] Stores correspond to hash table inserts and loads to table lookups. Reinsertions of the same key (i.e., stores to a previously stored-to address) replace the old value.

4.5.2 Instrumentation Timing

The LLVM compiler infrastructure [31] offers great flexibility in adding, removing, and reordering back-end compiler passes. For the purposes of our offline analyses, we found that most optimizations had very little effect. However, for both styles of instrumentation, performing memory-to-register promotion *prior* to instrumentation proved critical.

In both cases, the key issue was the presence of many approximate loads and stores that would never be present in an actual execution. This lead to excessive use of the

[4]We track the store's error counter so that we can assign "blame" to the store in the event of an error due to the approximate memory.

shadow memories. Prior to promotion, every read of an approximate variable results in an approximate load, and every write an approximate store. After promotion, most of these loads and stores are removed because the values are stored in registers. If we instrument prior to register promotion, our tool must add instrumentation for all of these memory accesses. This greatly distorts our dataflow tracking results, because each approximate load and store results in an additional approximate operation to count. For correlation tracking, the problems are not as severe, but they do result in many more approximate operations to track, which can cause the number of entries in the execution/error count vectors to explode. This in turn can make the the correlations more difficult to identify and can also impact scaling of our instrumentation.

Alternately, we could assume that the entire stack is stored in precise memory. In this case, loads and stores of approximate stack variables would not be treated as approximate operations, and thus would not need to be instrumented in the correlation case (they would still require instrumentation in the dataflow case, however, because approximate data can flow into precise values via endorsements). This is a reasonable assumption if you are modelling an architecture where the granularity of approximate storage is relatively coarse, because it may not make sense to keep separate precise and approximate stack regions if there are not likely to be enough approximate stack variables in a given frame to fill the minimal-sized approximate memory region.

4.6 Use Cases

To evaluate our instrumentation-based offline QoR analysis tools, we experimented with four approximate applications. Our offline tools allowed us to narrow in on the key quality issues in our applications and to better understand their characteristics:

- We used both dataflow and correlation instrumentation to analyze an approximate **Sobel filter** application. The application was approximated by declaring the two image arrays (input and working/output), as well as a number of tem-

porary variables, to be approximate data. This analysis allowed us to debug an intermittent crash under approximation, as well as frequent severe QoR degradations.

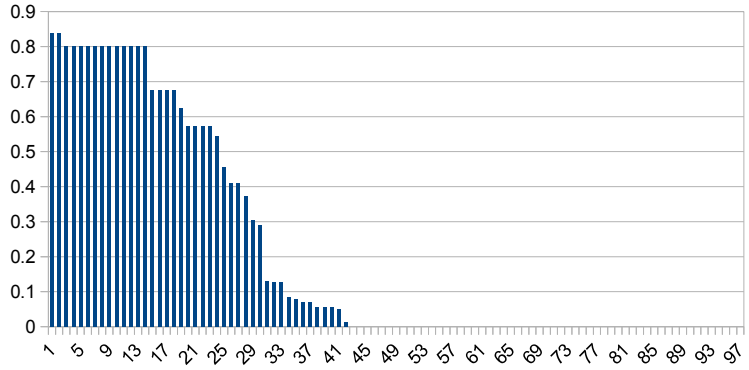
- We used dataflow instrumentation to better understand the approximation patterns of an approximate **FFT kernel** (from [52]) and used this to inform our design of the application-specific portions of the FFT kernel online monitors described in Chapter 5.

- We used correlation instrumentation to better understand the approximation present in an approximate version of the PARSEC `canneal` **simulated annealing** benchmark. This convinced us that the error patterns were such that further refinements (or the addition of a monitor) was not necessary.

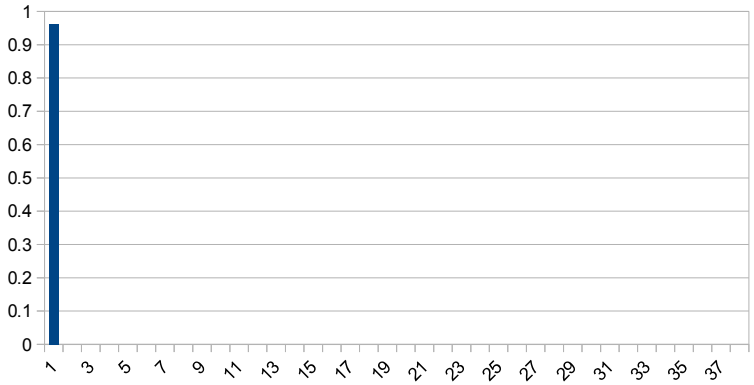
- We used correlation instrumentation to debug the quality of an approximate version of the PARSEC **Black Scholes** benchmark.

This section describes our experiences with these applications in more detail. The graphs in Figure 4.1 show how correlations are distributed in the applications that used correlation instrumentation.

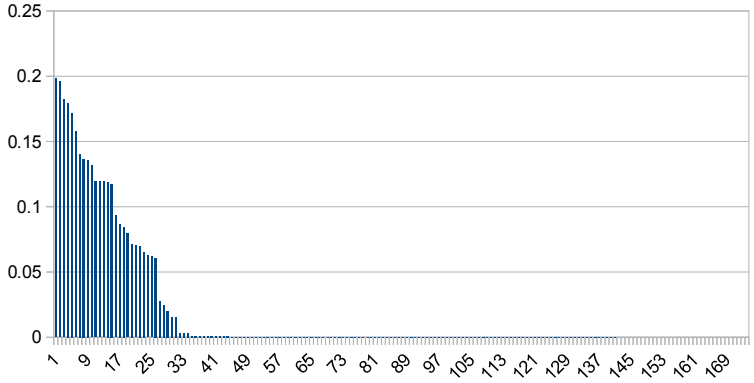
Sobel filter. Our instrumentation of the approximate Sobel filter enabled us to debug two problems: an intermittent segmentation fault and frequent poor quality filter results (e.g., no edges). To track down the crash, we created correlation vectors where the QoR component was determined entirely by whether or not we crashed. These vectors quickly pointed us to an array access code point. Dataflow instrumentation confirmed that this array index could be influenced by approximate operations, leading to the potential for out-of-bounds accesses. To investigate the poor quality results, we created correlation vectors whose QoR was based on the number of correct elements of the result and determined that the highest correlation was with code



(a)



(b)



(c)

Figure 4.1: Graphs showing correlations between code points and QoR in (a) simulated annealing, (b) Sobel filtering, and (c) Black Scholes. The x-axes represent source lines, and the y-axes represent QoR correlations. The x-axes are sorted by correlation value to show how the correlations are distributed: a small and informative number of approximate code points have high correlation to QoR.

that computed a scaling factor which was later applied to every pixel. This scaling factor was being computed approximately by scanning the initial image. Dataflow instrumentation confirmed that there were a large number of approximate operations in this scan. Since this value impacts every output pixel, it was causing our frequent garbage results.

FFT. Our primary insights with the FFT came from dataflow instrumentation. We checked the approximate dataflow into each element of the result and determined that, even with the relatively small FFT we used in testing (32K elements), each element of the output vector was dependent on a large number of approximate operations (almost 180K). Due to the structure of the FFT, if an error corrupts an intermediate array value early in the computation, it can lead to errors that propagate through the rest of the FFT result. This insight led us to design a monitor for the FFT application in such a way that it would catch any errors early so that we could either attempt to correct them or simply restart the computation. This is described in further detail in Chapter 5.

Simulated annealing. For `canneal`, we created correlation vectors where the QoR component was determined by the difference in route length from the precisely computed version. When we plugged our vectors into a spreadsheet to compute correlations, we determined that the results with lower quality were strongly correlated with errors in approximate operations inside the random number generation routines, and not very correlated with anything else. The random number generation is used to compute random steps in the simulation, and these errors appeared to effectively be altering the randomization. This was causing our annealer to simply find different local minima. In fact, a number of these different minima were *better* than the one found by the precise version. This investigation gave us increased confidence in the results of our approximation.

Black Scholes. In Black Scholes, our correlation instrumentation identified two locations with particularly high correlation to QoR. In both cases, we were loading a value from a location in approximate storage that had not been accessed in a long time. In our EnerJ-based approximation model, the decay of an approximate memory value is based on the amount of time since it was last accessed (since an access refreshes the memory). Situations like this suggest that future approximate languages may want a language feature that forces a refresh of approximate storage. Without this, developers must either avoid approximating these variables, or add artificial accesses to force a refresh (and do it in such a way that the compiler does not optimize the accesses away).

4.7 Related Work

This work proposes instrumentation-based approaches that pinpoint precise program points that lead to poor output quality dynamically. These techniques complement prior static approaches and improve on more basic dynamic approaches.

Static approaches conservatively bound the quality impacts of approximate computing. Carbin et al. [9] propose a proof system for verifying programmer-specified correctness properties and other work [39, 61] uses probabilistic reasoning to prove accuracy bounds on program transformations. EnerJ [52] provides a simple noninterference guarantee. The Rely system [10] bounds the probability that values produced by an approximate computation are correct by examining the static data flow of nondeterministic operations. In this sense, it represents a static complement to our dataflow instrumentation technique. Static techniques provide important safety properties but are necessarily conservative; our dynamic techniques are critical to addressing run-time events that static analyses cannot rule out.

The quality-of-service profiling work described in [40] describes an exhaustive search process for identifying program loops that do not need full precision. Our EnerCaml profiler and autotuner (Chapter 3) searches for approximate function appli-

cations that can be made precise to improve quality. In contrast, our instrumentation approaches apply to finer-grained sources of error without resorting to brute-force search.

Chapter 5

ONLINE QUALITY MONITORING OF APPROXIMATE APPLICATIONS

5.1 Introduction

The previous two chapters discussed offline, predeployment tools to help developers understand and improve the quality–energy tradeoffs of their approximate applications. We have shown how these tools can be valuable aids to programmers working to optimize and debug approximate programs. However, as with all offline tools, they depend on developers' ability to anticipate the range of possible inputs that might be seen "in the wild". In addition, offline tools cannot dynamically react to changing conditions (e.g., environmental conditions that might impact the error rate of approximate hardware).

In this chapter we tackle these issues by proposing *online quality monitoring*, which allows applications to dynamically detect and adapt to approximation's effects. It is the first work (to our knowledge) to consider the design challenges of online monitoring of approximate computations and to architect and implement a flexible framework for online monitoring of approximate applications. With online monitoring, approximate programs can self-adapt to cope with changing error rates and/or input patterns that are not anticipated during development.

The key challenge to realizing practical online monitoring is making it cheap enough that its overheads do not obviate the efficiency benefits of approximation. This chapter considers the design space of low-overhead, online QoR monitors. We propose three general approaches corresponding to different classes of algorithms and approximation patterns. We present implementations of these monitoring techniques

and evaluate their effectiveness in the context of six sample applications and eight monitors (two of the applications were implemented with two different monitors). With monitoring enabled, our applications retain between 44% and 78% of the original energy savings from approximation. These savings suggest that approximate applications can take advantage of online QoR monitoring to control the negative effects of approximate computing while still preserving significant energy-saving benefits. We also use monitoring to build a self-adapting ray tracer that adjusts approximation parameters at run time to find their best settings on the fly. This adaptation substantially reduces the tracer's error rate without negating the energy efficiency benefits of approximation.

In Section 5.2, we begin by revisiting the strawman we discussed in Section 2.2 in order to frame the online problem in contrast to the simpler *offline* monitoring problem. Section 5.3 then introduces three approaches to low-overhead online monitoring. Section 5.4 generalizes these approaches to map the design space of online monitors. Next, Sections 5.5 and 5.6 detail the API and implementation of our monitoring prototype. Finally, Section 5.7 describes our experiences using the prototype to monitor and control the QoR of six approximate applications.

5.2 Offline vs. Online Quality Monitoring

When writing code that trades off accuracy for resource usage, it is essential to understand how this trade-off affects computation quality. While resource usage—time or power, for example—can be measured directly, quality must be assessed using a program-specific metric. We refer to this application-defined notion of output quality as the quality of result or QoR. For an object recognition application, for example, the QoR metric may be the number of correct classifications. One way to measure QoR is to run the program repeatedly in a controlled test environment and collect and compare the outputs of precise and approximate computations. We refer to this profiling-based approach as *offline monitoring*.

This section defines the offline monitoring problem to contrast it with the *online* approaches that comprise this chapter's contribution. Like any technique based on pre-deployment test executions, offline monitoring relies on the assumption that program behavior in deployment will resemble test runs. The purpose of online monitoring is to relax this assumption and directly assess dynamic behavior in the field.

The goal of any QoR monitor is to measure the effects of approximation on a piece of approximate code (e.g., a block, kernel, or function). For instance, if the code contains approximate arithmetic, we want to detect when arithmetic errors cause the code's output to differ too much from what the results would have been if only precise arithmetic had been used. If the outputs are too degraded, the monitor should report a quality violation.

To measure output quality, an offline monitor can execute the monitored block twice—once approximately and once precisely—and compare outputs. For example, consider a ray tracer, where we wish to monitor the computation of each pixel:

```
monitor_block { tracePx(x, y); }
```

An offline monitor would effectively execute this as:

```
approx = tracePx(x, y);
precise = runPrecise { tracePx(x, y); }
if (abs(approx - precise) > Threshold)
  throw new FailedQoR();
```

This approach reveals two problems that must be solved to make any monitoring scheme—online or offline—feasible.

First, the code above assumes idempotency of the monitored code block. Except in a purely functional setting, approximate computations can and often do have side effects. If we wish to run a non-idempotent code block twice, we need to buffer or roll back these side effects. In addition, side effects can differ arbitrarily between different executions of the same approximate code and may impact QoR.

Second, comparing numeric return values is insufficient for measuring the output quality of many applications. QoR is inherently domain-specific, so we must support arbitrary, application-specific metrics. For example, a video application may prefer neighboring frames that are distorted in the same way (thus preventing jitter) over neighboring frames with smaller average distortion but which are distorted in different ways. Another example is a greedy algorithm that searches for local optima. An approximate version that selects a different optimum from the precise version can have equal—or possibly even superior—result quality.

Any approach to QoR monitoring will need to address side effects and provide flexible QoR metrics. We address these needs with a flexible monitoring API that provides hooks for programmer-specified quality metrics (Section 5.5) and transparent side effect isolation (Sections 5.4.3 and 5.6.1).

But, aside from these concerns, the above monitoring approach is inherently tied to the *offline* monitoring problem. Fundamentally, precise re-execution of approximate code does *more* work (and uses more energy) than the original, non-approximated code. To detect unanticipated behavior in the field and adapt on the fly, we need monitoring that does not obviate the efficiency benefits of approximate computing.

Unlike the offline case, *there is no single, efficient, general-purpose solution to the problem of online quality monitoring.* This chapter explores techniques that achieve low overhead at the cost of generality and precision. The next section enumerates three techniques that relax offline monitoring and are cheap enough to run "always-on" in production.

5.3 Approaches to Online Quality Monitoring

This section presents three realistic approaches that limit their generality to achieve tenable overheads. Section 5.4 analyzes these ideas to describe the design space of QoR monitors.

5.3.1 Precise Sampling

The first approach we consider is *precise sampling*. Like offline monitoring, precise sampling compares the results of the precise and approximate versions of the monitored code. Unlike offline monitoring, this strategy checks only a random subset of the executions. In the sampled executions, a developer-provided function compares the output of the two executions. The developer can tune the sampling frequency to manage the trade-off between overhead and monitoring precision. Higher rates detect bad approximations with higher probability but approach the overhead of offline monitoring.

In a ray tracer, a sampling monitor might execute as follows:

```
result = tracePx(x, y);
if (random() < sampleFreq) {
  precise = runPrecise { tracePx(x, y); }
  if (!compare(result, precise, approxOutput,
               preciseOutput))
    throw new FailedQoR();
}
image[x][y] = result;
```

Here `compare` is a developer-supplied function that returns true if the comparison between the precise results and the approximate results indicates acceptable QoR. The `approxOutput` and `preciseOutput` arguments capture any memory side effects of the approximate and precise executions. This is left intentionally vague here; side effects are an orthogonal issue discussed in Section 5.4.3.

Precise sampling is appropriate for applications where quality properties can be checked by looking at a random subset of the output. We can monitor—with probabilistic guarantees—the fraction of correct executions of a code block or its average error. We cannot use precise sampling for applications that require a bound on the

worst-case error. For example, in an asteroid dodging game (Section 5.7.4), precise sampling could not guarantee that asteroids *never* jump across the screen.[1]

When implementing a precise sampling style online monitor, we must keep in mind that the granularity of checked code may not always match the granularity of approximation. For example, we may want to approximate a ray tracer at the granularity of individual rays. However, a more reasonable granularity for checking is likely at the level of pixels (which are the sum of multiple rays): we may not care that a ray which only contributes a small amount to a pixel's brightness is off by 50% if other much brighter rays completely dominate it. Thus implementations of precise sampling require a way to specify precise execution of a block of code, as well as of any code that it calls. In our prototype, this is accomplished by the `checkApprox` infrastructure described in Section 5.5.1.

5.3.2 Verification Functions

Our second approach to quality monitoring is *verification functions*. Verification functions are routines supplied by the developer that can check the QoR of a computation. Verification functions are useful whenever we can *check* the correctness of a result at significantly lower cost than *computing* the result. In contrast to precise sampling, this approach relaxes offline monitoring by reducing the cost of each check rather than reducing the number of checks.

We consider three forms of verification functions, each of which requires different inputs. The first form, which we term *traditional verification*, verifies the outputs of the current execution based on its inputs. For example, a 3-SAT verifier could check that the outputs (the variable assignments) satisfy the inputs (the formula clauses). The second form, *streaming verification*, verifies the output of the current execution based on the output of past executions. For example, a video decoder could check

[1] Although perhaps "teleporting asteroids" could be considered a feature rather than a bug at higher difficulty levels...

that the current frame bears a sufficient resemblance to past frames (possibly modulo motion estimation). The final form, *consistent output verification*, looks only at the output of the current computation and verifies that it holds some desired property: for example, that a computed probability distribution sums to 1.0 or that a number lies within an expected range.

For example, a verification function monitor running our ray tracer might utilize a consistent output verification function that checks properties such as whether the pixel brightness is within an expected range. Alternatively, an animated application could use streaming verification to check that most pixels' values are usually similar to their values in previous frames.

5.3.3 Fuzzy Memoization

Our third approach to quality monitoring is *fuzzy memoization*. Fuzzy memoization records previous inputs and outputs of the checked code and predicts the output of the current execution from past executions with *similar* inputs. Analogous ideas were previously used by Chaudhuri et al. [13] and Alvarez et al. [2] to *provide* approximate execution rather than to check the quality of the execution. We estimate the QoR by checking how different the current output is from the predicted output.

We identify several variations distinguished by their prediction mechanisms. The simplest predicts the previously recorded output with the most similar input. We call this approach *simple fuzzy memoization*. Another variation performs interpolation between a set of similar previous inputs. We refer to this as *interpolated fuzzy memoization*. More complex variations could attempt to perform curve fitting or apply machine learning techniques (e.g., support vector machines) to learn the relation between inputs and outputs. We term this extension *learned fuzzy memoization*. Like verification functions, fuzzy memoization solves the overhead issue with frequent cheap checks rather than rare expensive checks. It is applicable when the function computed by the checked code is relatively continuous (or easily learnable).

QoR monitors based on fuzzy memoization become more accurate as they observe more executions of the monitored code. In the early stages, the prediction model contains few inputs. As execution proceeds, the monitor adds more results to the model and predictions improve. However, if a poorly approximated result is added to the model, it can hurt future estimates. Also, depending on the memoization implementation, adding results may increase memory overheads and eventually outweigh the energy savings from approximation. In addition, even after many results have been added to the model, new results in poorly sampled (or discontinuous) regions of the input space may have poor predictions.

To solve these issues, we propose a three-pronged approach. First, the monitor should use some precise runs to ensure that the model is seeded with known-good values. Precise runs should be used at the beginning of program execution to seed the model with some initial values, and may also be sampled randomly throughout execution to ensure that the model contains data from all regions of the input space. Second, the monitor should limit the number of approximate results added to the model and add only those whose QoR estimates meet a developer-specified threshold. This prevents the model from growing too large and may keep some bad data from corrupting the model. However, as mentioned above, it is also possible that a negative prediction is due to a poor model rather than poor QoR. This may be caused, for example, by a sparsely sampled input region. Thus, our third proposal is that some failed checks lead to precise re-execution in order to improve the model's accuracy.

For example, a simple fuzzy memoization monitor running our ray tracer might execute as follows:

```
if (preciseSeedingRun()) {
  result = runPrecise { tracePx(x, y); }
  addMemo(x, y, result, output);
} else {
  result = tracePx(x, y);
```

```
  if (!compNearestMemos(x,y,result,output))
    if (updateModel()) {
      result = runPrecise { tracePx(x, y); }
      addMemo(x, y, result, output);
    }
    else throw new FailedQoR();
}
image[x][y] = result;
```

Here, `addMemo` adds a new result to the model and `compNearestMemos` finds nearby memoized results to compare with our current result. This example works best if we expect the image to contain large regions of similar color.

Simple fuzzy memoization can be implemented with a data structure that stores previous results as input–output (key–value) pairs that can be efficiently retrieved when we encounter nearby inputs (keys). We propose a self-balanced binary search tree. Given an input key, we can identify neighboring keys in $O(\log n)$ time. The space overhead of storing results mandates that we not allow the record of past results to become too large, so $O(\log n)$ should remain small.

5.4 The Design Space

We now consider the broader design space of quality monitoring. We begin with a discussion of the dimensions of the design space (Section 5.4.1). We then mention a possible additional dimension—code-centric versus data-centric annotation—and argue why code-centric is the better choice (Section 5.4.2). Finally, we discuss the side-effect dimension in more depth (Section 5.4.3).

5.4.1 Design Space Dimensions

We can describe the design space of QoR monitoring algorithms in terms of four (mostly) orthogonal dimensions:

- **What is Checked**: Do we check every execution, or a sample? If we sample, what is the distribution?

- **Checkable Quality Properties**: What can we verify? Do we want every execution to be within some error bound? Or do we want the average error over all executions to be within the bound? Do we want at least some percentage of the executions to match the precise result? Also, how configurable is the QoR metric? Does it need to be reducible to a floating point number that we verify is within an epsilon of expected, or can developers create arbitrary code to check correctness?

- **Checking Parameters**: What inputs are supplied to the checking algorithm? Do we look only at the outputs of the current execution? Or do we look at the inputs as well? Can we also look at inputs and outputs of past executions? Or at the results of a precise execution?

- **Side Effects**: How do we deal with the side effects of approximate computations? This is a significant issue, so we devote Section 5.4.3 to discussing it.

Certain regions of this space are clearly less desirable than others. For example, requiring the results of a precise computation combined with checking every execution leads to the prohibitive overhead of offline monitoring. Similarly, combining sampling with error bounds on every execution is impossible.

Figure 5.1 fits the approaches form Section 5.3 into this space.

5.4.2 The Code-Centric Nature of Quality Measurement

QoR monitoring can be expressed in either a code-centric or a data-centric style. In this section, we describe and contrast both styles and argue for the benefits of a code-centric style.

Figure 5.1: This table shows how each of our monitoring approaches fits into the design space and discusses its applicability. We have left off the side effects dimension as it is orthogonal.

Approach	What Is Checked	Checking Parameters	Applicability & Checkable Quality Properties
Precise Sampling	Sampled executions	Approximate and precise outputs	Applicable to code that can be re-executed (possibly with rollback). Desired properties must be checkable with a random execution sample.
Verification Functions	Every execution		Applicable when checking is cheaper than computing.
Traditional		Approximate inputs and outputs	
Streaming		Current & old approximate outputs	
Consistency		Approximate outputs	
Fuzzy Memoization	Every execution	Current and old inputs and outputs	Applicable to learnable computations. Quality metric must be representable as a distance.

Code-centric annotations specify requirements on regions of code. For example, the following annotation might specify that the value of a pixel should be similar to the previously computed pixel:

```
nextPixel = checkComp(computePixel, args, compareToPrevious);
```

Here `checkComp` will call `computePixel(args)` and pass its result to the checking routine `compareToPrevious`, which will compare that result to the last result it saw.

Data-centric annotations, in contrast, apply to pieces of approximate data. Such requirements are typically relative to the corresponding value at the same time during a precise execution of the program. For example, the following annotation could specify that the value of `pixelA` should vary at most 5% compared to its value during a precise execution:

```
@Approx<0.05> Double pixelA;
```

The two opposing annotation styles have two important similarities. First, the choice of code-centric versus data-centric QoR measurement is orthogonal to the choice of code-centric versus data-centric approximation. For instance, the `computePixel` function in the code-centric example above may base its approximation on data-centric annotations on values used to compute the pixel. Second, code-centric annotations are not limited to checking return values: they can check any data that is live at the end of the computation. The two approaches differ less in which data is checked and more in the frequency and granularity of checks.

For the purposes of online monitoring, code-centric annotations offer a number of advantages: they align better with developers' ultimate goal; they permit more flexibility in the kinds of monitors used; and they make low-overhead implementation easier.

Goal-oriented annotations. Code-centric annotations apply to the result of a computation, which is generally what programmers ultimately care about. Appropriate QoR constraints are less obvious on intermediate values. Data-centric annotations on intermediate values may also generate false positives. For example, a ray tracer computes a pixel by adding the contributions of many rays. If the initial rays have small contributions compared to later rays, even large relative errors in those rays may have little impact on the final computed value.

Checking flexibility. Limiting checking to specific times lets code-centric annotations perform more expensive checks. If we must constantly monitor all intermediate values of a variable, the checks must be extremely cheap. Conversely, with a code-centric specification, QoR requirements can be represented by arbitrary functions, as long as they are cheap *relative to the cost of the computation being checked*. If this computation is, for example, the tracing and summing of all of the rays that contribute to a pixel value, the checking function can be fairly complex. If we attempt to extend data-centric annotations to check the value at only certain points, they essentially become code-centric annotations.

Implementation of monitoring and recovery. Limiting the checking to a single, natural point in the computation simplifies the implementation of the monitoring framework. Monitoring need only be invoked via explicit library calls; we do not need to instrument variable accesses. In addition, code-centric annotations make it easy to take recovery actions in response to quality monitoring. A system can, for example, re-execute a marked block when it fails a QoR check.

5.4.3 Dealing With Side Effects

Code-centric monitors naturally have inputs (arguments) and outputs (return values) that can be checked by a QoR monitor. But what if approximate computations mutate

other data or have other side effects? These side effects impact quality and may differ in an approximate execution. For example, control flow changes in the approximate execution may cause a write to execute that does not occur in the precise version.[2]

These sorts of unanticipated side effects can harm QoR in ways that the developer-specified metrics, constraints, or comparisons do not account for. Side effects can thus violate the expectation that quality monitoring catches all unacceptable precision losses. Any monitoring solution needs to account for this difficulty. We propose three general approaches:

- **Ignore side effects**: This is the simplest approach, but it shifts the burden entirely to the developer. The monitor assumes that the inputs to the QoR function or verifier are the only things that matter. Developers must ensure that any other possible side effects are incidental and will not affect the overall quality of the computation. This can be difficult, however, due to the possibility of unanticipated side effects. It may be more appropriate in a mostly functional language where side effects are less common.

- **Ensure precise and approximate side effects are identical**: In this approach, the compiler and runtime system ensure that, if an approximate execution modifies any data that is not part of the input to the QoR function or verifier, then an equivalent precise execution would have produced the same modification. In the general case this requires significant dynamic cooperation from the runtime system. The overheads required would likely overwhelm the energy-saving benefits of approximation.

- **Restrict side effects**: Our final approach simply detects and forbids side effects in monitored code except for data that is local to the computation *or*

[2]Note that EnerJ forbids using approximate data for control flow. However, control flow can still vary in the approximate execution due to *endorsements*. Endorsements allow the programmer to sign off on assigning an approximately-computed value to a precise variable.

explicitly marked as an output of the computation. The monitor can check this explicitly marked output data and verify its quality. If the runtime detects disallowed side effects, it raises an exception. This approach again requires dynamic cooperation from the runtime but, as we demonstrate in Section 5.6.1, can be done relatively cheaply.

We contend that ignoring side effects pushes too much of the burden on to the developer, and ensuring identical side effects creates too much overhead. Therefore, our prototype monitor pursues the third option, as discussed in Sections 5.5.1 and 5.6.1.

The above discussion focused on side effects that cause incorrect writes. This includes most forms of I/O, as they typically involve an initial write to a buffer. However, the above approaches do not handle program completion or execution length effects. For example, approximation may cause a program to enter a very long (or even infinite) loop that would never occur during a precise execution. We have not encountered approximate side effects of this sort in any of the programs that we have explored. However, if necessary they could be handled by adding developer-specified timeouts to our online monitors.

5.5 A Monitoring API for EnerJ

We have designed a runtime system for monitoring the quality of, and restricting the side effects in, monitored code blocks. Our system is flexible enough to support approaches within the design space outlined in Section 5.4, including the specific approaches we advocate in Section 5.3. Our system is built on top of the EnerJ language and simulator [52] (as previously described in Section 2.3 of Chapter 2).

Our system is divided into the three layers depicted in Figure 5.2. As in any layered system, the orthogonality and clear separation of the layers provides important benefits for extensibility and experimentation. The bottom layer is the EnerJ runtime/simulator, which we extended with support for monitoring code blocks, main-

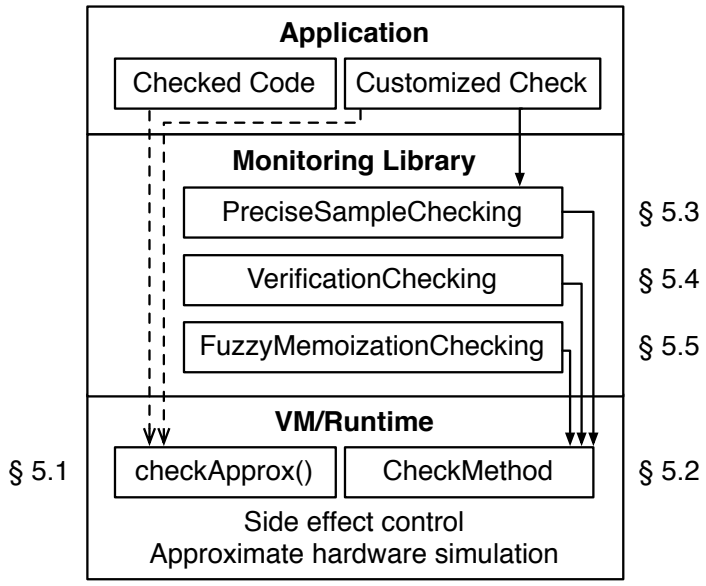


Figure 5.2: The architecture of our monitoring framework prototype. Solid arrows indicate inheritance; dashed arrows indicate parameters to invocations.

taining side-effect restrictions, and copying objects to support re-execution. The middle layer is a pure-Java library that provides classes for precise sampling, verification functions, and fuzzy memoization. The top layer consists of monitors customized by application developers using the middle-layer functionality. This section describes the functionality of the layers and the interfaces between them. Sections 5.5.1 and 5.5.2 describe what the bottom layer provides the middle layer: a method for monitoring a code block and an interface for describing how the monitoring needs to proceed. Sections 5.5.3, 5.5.4, and 5.5.5 then describe the interfaces that the middle layer provides to the top layer for different forms of monitoring. Section 5.6 describes the salient aspects of the implementations of the bottom and middle layers. In Section 5.7, we describe some sample developer-customized (i.e., application-specific) implementations of the top layer.

5.5.1 The checkApprox Function

In our system, developers monitor code by passing it to the `checkApprox` API:

```
Object checkApprox(Object argList[],
                   Object outputObjs[],
                   CheckMethod checkM,
                   CheckApproxCodeBlock m)
```

Here `m` is an instance of the `CheckApproxCodeBlock` interface, whose method `f` is the computation to check:

```
public interface CheckApproxCodeBlock {
   Object f(Object[] args);
}
```

The `argList` array contains the inputs to the checked computation. The array `outputObjs` contains the objects that may be written by side effects in the checked computation (and checked by the monitor). Any stores to objects that are not either allocated by the checked computation or members of this list are considered illegal side effects and cause an exception. This style of allowable side effect specification has been sufficient for the applications we have considered. However, a more flexible API that allowed for dynamic additions to the output list would be straightforward to implement, and useful for situations where the computation may traverse a data structure and potentially modify pieces of it. Finally, the `checkM` argument contains a quality monitor as described below (in Section 5.5.2).

For example, consider monitoring a pixel value computation in a ray tracer:

```
Integer pixel = computePixel(xCoord, yCoord);
```

If `computePixel` has no side effects, the appropriate `checkApprox` call is:

```
// Create CheckApproxCodeBlock containing code to be checked
CheckApproxCodeBlock cacb =
```

```
  new CheckApproxCodeBlock() {
    public Object f(Object[] argL) {
      return computePixel((int)argL[0], (int)argL[1]);
    }
  };
Object arg[] = { xCoord, yCoord };
Object out[] = { };

// Call checkApprox, passing code block, arguments, permissible
// side effects, and the monitor
pixel = (Integer)checkApprox(arg, out, pSampler, cacb);
```

The `checkApprox` function executes the `f` method of `cacb`, passing it the `arg` argument array. Note that `f` may execute arbitrary code, not just a simple function call as shown here. Our framework ensures that the computation modifies only objects that are locally allocated (since `out` is empty). While executing `f`, `checkApprox` uses the quality monitor `pSampler` to monitor the QoR.

If our ray tracer maintains an array `nInt` to track the number of times each object is intersected by a ray, we can simply add `nInt` to the output list `out`. Our framework passes `out` to the verifier, so we can monitor the quality of any modifications to `nInt`.

5.5.2 Quality Monitors

We created a `CheckMethod` interface whose implementations correspond to monitoring approaches. It specifies a set of methods that our framework may call during monitoring:

```
public interface CheckMethod {
  boolean runTwice();
  boolean needPrecise();
  boolean recordPast();
```

```
  void recordInputs(Object[] inp);
  void recordOutputs(Object[] outp);
  void recordReturn(Object ret);

  boolean evaluate(Object[] in, Object[] out, Object returnV,
                   Object[] preOut, Object preReturnV);
}
```

The `runTwice` method returns true if the monitor should run the checked code both approximately and precisely. If so, our framework ensures that side effects are buffered during the first execution (see Section 5.6.1). Precise sampling uses this during sampled runs. The `needPrecise` method returns true if the monitor requires the checked code to be run precisely. Fuzzy memoization uses this to seed the model with precise values. The `recordPast` method returns true if the monitor needs to record past input, output, and return values. If so, `checkApprox` makes shallow copies of these values and passes them to the appropriate `record*` methods. Streaming verification uses this to compare with previous executions. We also allow developers to override the default shallow copying with their own copying methods. This can serve two purposes. First, a full copy may be unnecessary and create extra space, performance, and energy overhead if the monitor only needs to examine pieces of the input, output, or returned data structures. Second, developers may need the monitor to create customized deep copies to prevent corruption of recorded past results.[3]

Finally, `checkApprox` calls the `evaluate` method of the `CheckMethod` interface to determine whether the checked code block met the QoR requirements. It is

[3]However, developers should be aware that deep copying adds additional overhead and may indicate that streaming verification is not the ideal monitoring strategy for the application in question.

passed the input, output, and return values. If a second precise run was performed, it is also passed its outputs and return value. The implementation of this checking varies among the different approaches, but for all of our approaches the implementation of `CheckMethod` contains additional interfaces that allow the developer to specify how the evaluation should determine QoR. The following sections describe these interfaces.

5.5.3 Precise Sampling

The `PreciseSampleChecking` class (an implementation of the `CheckMethod` interface) provides precise sampling online monitoring. The constructor specifies options such as the sampling frequency as well as an implementation of an interface encapsulating a QoR constraint:

```
public interface QoREval<T> {
  T QoR(Object[] out, Object ret, Object[] appOut,
        Object appRet);
  boolean constraint(T q);
}
```

The `QoR` method computes the QoR by comparing the precise outputs and return value (`out`, `ret`) with the approximate outputs and return value (`appOut`, `appRet`). The `constraint` method returns true if quality returned by `QoR` is acceptable. The constraint method may optionally store information about previous QoR values (e.g., a running average) to help decide if quality is acceptable. This is why we separate it out from the `QoR` method, which is intended to compute a single QoR.

5.5.4 Verification Functions

Similarly, developers can create an instance of the `VerificationChecking` class (another implementation of `CheckMethod`) to utilize verification function monitor-

ing. Developers pass the constructor an implementation of one of three interfaces, each of which specifies a different type of verification function. Each interface contains a `qualityVerify` method that computes QoR based on the inputs appropriate to the type of verification function and a `constraint` method that returns true if the quality is acceptable:

- `VerifierStreaming` for streaming verification. We pass the outputs and the return values of the current and last executions to the `qualityVerify` method.

- `VerifierConsistentOutput` for consistent output verification. We pass the current outputs and return value.

- `VerifierTraditional` for traditional verification. We pass current input, output, and return values.

In streaming verification, the overhead of copying and storing the output of every execution can be high. Thus, we allow the developer to optionally specify that the monitor should record only every nth execution and pass that to the next n verifer calls. In this case, the monitor also passes a `distance` argument to the checking function that specifies how long ago the recorded output occurred. Larger variance may be expected when comparing with older outputs. For example, a simulated asteroid may have moved farther. Developers can also provide custom output copiers. These are a useful to reduce data copying overheads if the output contains data not needed to evaluate QoR.

5.5.5 Fuzzy Memoization

Finally, to utilize fuzzy memoization monitors, developers can create an instance of the `FuzzyMemoizationChecking` class (which also implements `CheckMethod`).

They must pass the constructor options specifying when to use precise runs to seed the model, and implementations of two interfaces. The first interface converts the inputs and outputs of the checked code into a point in the space that we are memoizing over. Our prototype uses 2D linear interpolation, so our interface specifies methods to convert the input to an x-coordinate and the output to a y-coordinate:

```
public interface ResultMemoizer {
  Number keyFromInputs(Object[] inputs);
  Number valueFromOutputs(Object[] outputs, Object returnVal);
}
```

The second interface specifies various error thresholds:

```
public interface MemoConstraints {
  boolean      accept(float actual, float predict, float distPrev,
                      float distNext);
  boolean  addToModel(float actual, float predict, float distPrev,
                      float distNext);
  boolean adjustModel(float actual, float predict, float distPrev,
                      float distNext);
}
```

In the above, `actual` and `predict` represent the computed and predicted values of y (representing the output). Similarly, `distNext` and `distPrev` represent the distance of the actual input key x from the keys of the points that were used to interpolate. Prediction may be fuzzier if we are farther from these points, so a larger error may be reasonable.

The `accept` method returns `true` if we should accept the QoR of the current execution. The `addToModel` method lets the developer specify that they would like to add the current result to the interpolation model. If the developer is confident that the QoR is high, then we may improve the model by adding new points. However, developers should also be aware that adding points may increase overhead. The

`adjustModel` method, in contrast, is typically used when we see low QoR and think that it may be due to problems with the model. For example, there could be discontinuities that are not captured by previous data. If `adjustModel` returns `true`, we execute additional precise runs to improve the model (possibly including rerunning the current computation).

5.6 Implementing the Monitoring API

This section discusses the implementation of the monitoring prototype described in Section 5.5. First, Section 5.6.1 describes our changes to the EnerJ runtime to handle side effects. We then describe salient details of our implementations of the three monitoring approaches (Sections 5.6.2, 5.6.3, and 5.6.4).

5.6.1 Handling Side Effects: Restricting and Buffering

Our monitoring system restricts side effects by allowing checked blocks to write only to objects that are either part of the output list or local to the checked code. Any other memory write results in an exception. Possible implementation strategies include reusing existing memory protection mechanisms or keeping per-object data indicating which checked computations may write to the object. Our prototype uses the latter since the EnerJ simulator already tracks per-object state.

Specifically, our prototype tracks whether objects are writable by augmenting heap-allocated objects with per-object state containing a region number. When `checkApprox` is called, it enters a new region by incrementing a global region counter. To support nested calls, each call to `checkApprox` records the region number of the parent call and restores the counter when it returns. Thus, as we return from the `checkApprox` invocations on the call stack, we also unwind the region number stack. Any object allocated inside a checked region sets its region number to the number of the current region. In addition, any objects specified as output data have their region number updated to the current region number. When entering a

nested region, we first check that the output object was writable by the parent region. (It is unsafe to make an object writable by the child when it is not writable by the parent.) Before returning, we reset the output objects' region number to the parent region number.

To enforce the side effect restrictions, all stores to heap objects inside monitored code check the destination object's region number against the current region number. If there is a mismatch, we throw an exception.

In addition to restricting side effects, our implementation needs to buffer and roll back side effects for monitoring approaches that incorporate re-execution (e.g., precise sampling). We provide buffering using a copy-on-write policy for non-local objects. To implement copy-on-write, we add a boolean to each object indicating whether it should be copied when written and a pointer (initially null) to the copy. When we enter `checkApprox`, we call the monitor's `runTwice` method to check if copy-on-write is necessary. Currently this call returns true only if the monitoring method is precise sampling *and* this is the first (approximate) run of a sampled execution (i.e., we will re-execute it), so we incur overhead only in sampled executions. If copy-on write is necessary, we iterate through the output list and set the copy-on-write flag. If a store occurs to an object whose flag is set, we check the copy pointer and create a copy if it is null. We then perform the store to the copy instead of the original object. When we load from an object with the copy-on-write boolean set, we again check the copy pointer and read the copy if it is present. After the first execution completes, we remove all of the copies and unset the copy-on-write flag. This allows the subsequent run to start as if from scratch.

5.6.2 Precise Sampling

At the beginning of every execution of `checkApprox`, we call the passed-in verifier's `runTwice` method. For precise sampling, this method returns true if we should sample the current execution. If so, `checkApprox` performs the approximate execution

while buffering side effects. It stores the outputs and return value and clears the buffer. It then disables approximation and re-executes the code precisely. Since the precise execution is the final execution, we do not need to buffer writes. After the precise execution completes, we call the `evaluate` method of `preciseSampleChecking`, passing it the precise and approximate outputs and return values. The `evaluate` method calls the `QoR` and `constraint` methods from the developer-provided quality interface (see Section 5.5.3) to determine whether to accept the execution. If the execution quality is acceptable, `checkApprox` returns the value returned by the precise execution.[4] If the QoR of the execution is not acceptable, `checkApprox` will instead throw an exception.

5.6.3 Verification Checking

When an approximate application calls the `VerificationChecking` constructor, it sets the verification style based on the particular type of verifier passed in. When we call `checkApprox` with this checker, it executes the checked code block and then calls `evaluate`. The `evaluate` method checks the verification style and passes the appropriate arguments to the developer-specified `qualityVerify` method (see Section 5.5.4). If the style is streaming, `recordPast` will return true and `checkApprox` will utilize the runtime layer's copying functionality to copy the outputs and return value and pass them to the verification layer via `recordOutputs` and `recordReturn`. The verification layer stores these and passes them to the next quality evaluation.

---

[4]We could return the approximate value instead, but it is no more expensive to return the precise value since we have already computed it. And the precise value is guaranteed to have quality as good or better than the approximate value.

5.6.4 Fuzzy Memoization

Our prototype implementation of fuzzy memoization uses simple 2D linear interpolation between neighboring memoized inputs to predict the precise output. We then use the developer-provided functions described in Section 5.5.5 to compare the actual output with the predicted output and determine whether to accept the actual output. As mentioned in Section 5.3.3, this requires a data structure that can store previous results as input–output pairs and retrieve them efficiently when we encounter nearby inputs. We use a red–black tree.

Other possible implementations of fuzzy memoization include higher dimensional linear interpolation, curve fitting, and what we refer to as *memo binning*. In a memo binning approach, inputs are rounded off and placed into bins. For example, the input $(1.1, 2.2, 3.9)$ might get placed into the $(1.0, 2.0, 4.0)$-bin. The outputs of all items placed in a bin are combined (e.g., by averaging) to produce a prediction for future inputs that land in the same bin. 2D linear interpolation was sufficient for the applications we describe in Section 5.7, but memo binning is more general.

5.7 Evaluation

To evaluate our quality monitoring tools, we experimented with adding monitoring to six approximate programs. For two programs, we created two versions using different monitoring techniques. For each of the eight configurations, we quantify the overhead of monitoring in terms of time (instructions), space (memory footprint), and modeled energy (see Section 5.7.1). We also measure the precision of each monitor with respect to an offline (ideal) quality monitor.

Table 5.1 shows the instruction and memory overheads of each quality monitor. Instruction overheads ranged from 3% to 55%, with five configurations having overhead under 10%, and memory overheads ranged from under 1% to 43%. Using a fairly conservative model of energy savings from approximation, described in the next

Application	Instruction Compute	Instruction Check	Instruction Overhead	Memory Compute	Memory Check	Memory Overhead
Triangle intersect, traditional verifier	95.8%	4.2%	**4.4%**	99.0%	1.0%	**1.0%**
Asteroids, streaming verifier	64.7%	35.3%	**54.6%**	89.1%	10.9%	**12.3%**
Asteroids, consistency verifier	74.1%	25.9%	**35.0%**	94.8%	5.2%	**5.5%**
Simple ray tracer, precise sampling	85.7%	14.7%	**17.2%**	69.9%	30.1%	**43.1%**
Sobel filter, fuzzy memoization	93.2%	6.8%	**7.3%**	75.4%	24.7%	**32.7%**
FFT, consistency verifier	93.5%	6.5%	**7.0%**	90.9%	9.1%	**10.0%**
FFT, fuzzy memoization	92.3%	7.7%	**8.3%**	99.7%	0.3%	**0.3%**
Black Scholes, consistency verifier	96.7%	3.3%	**3.4%**	74.2%	24.8%	**34.8%**

Table 5.1: The percentage of instructions and memory dedicated to the original computation (compute) and the monitoring (check) for each application and each monitor. We also list the instruction and memory monitoring overheads, computed by dividing the check columns by the compute columns.

Application	Type of Monitor	Precise	Approx	Precise Monitored	Approx Monitored	Savings Retained
Simple Ray Tracer	Precise Sampling	100%	67.3%	117.2%	85.5%	**44.3%**
Asteroids, 10k frames	Streaming Verifier	100%	91.2%	103.7% (130.0%)	95.0% (121.5%)	**56.8%**
Asteroids, 10k frames	Consistency Verifier	100%	91.2%	104.8% (119.2%)	95.2% (107.4%)	**54.5%**
Triangle Intersection	Traditional Verifier	100%	83.2%	104.3%	86.8%	**77.7%**
Sobel Filter	Fuzzy Memoization	100%	85.6%	107.0%	92.9%	**49.0%**
FFT	Consistency Verifier	100%	72.8%	106.9%	82.5%	**64.3%**
FFT	Fuzzy Memoization	100%	73.4%	108.4%	81.6%	**69.2%**
Black Scholes	Consistency Verifier	100%	73.1%	117.0%	88.1%	**44.4%**

Table 5.2: The modeled energy consumption of each monitored application. We use conservative energy savings and approximation parameters. Energy is measured as the percentage of the precise, unmonitored execution energy (the **Precise** column). The **Approx** column shows the energy usage of an unmonitored approximate execution. The **Precise Monitored** column shows the energy usage of a precise, monitored execution (this would not be useful in practice, but is included to show the overall energy overhead of monitoring). **Approx Monitored** shows the energy usage of an approximate monitored execution and **Savings Retained** is the percentage of the unmonitored energy savings that are retained after we add monitoring. All applications were run five times and the energy averaged. As described in Section 5.7.4, we measured the Asteroids application for 10,000 frames. Because these frames include unmonitored post-training frames, the precise monitored column does not reflect the true overhead of completely monitoring Asteroids. Thus, we have included the relative costs of the training (monitored) portion of asteroids in parentheses in the appropriate columns.

Application	**Type of Monitor**	**Errors caught vs. perfect monitor**	**False Positives**
Simple Ray Tracer	Precise Sampling	Sampling rate (with a 9.6% MAE)	0.0%
Asteroids, 10k frames	Streaming Verifier	54.8%	0.0%
Asteroids, 10k frames	Consistency Verifier	8.0%	0.0%
Triangle Intersection	Traditional Verifier	47.7%	0.2%
Sobel Filter	Fuzzy Memoization	86.7%	2.5%
FFT	Consistency Verifier	100.0%	0.0%
FFT	Fuzzy Memoization	90.1%	1.3%
Black Scholes	Consistency Verifier	65.8%	0.0%

Table 5.3: The percentage of errors caught by our example monitors, when compared with perfect, offline monitors. For precise sampling, the percentage of errors caught will be approximately the sampling rate, with some level of error. We account for this in the table above by indicating that the percentage caught will be the sampling rate, plus or minus the mean average error of the rate of errors in sampled executions versus the rate of errors that would have been detected by the perfect monitor. We also show the percentage of executions that resulted in a false positive—i.e., when the monitor reports a QoR error that did not occur.

section, we translate these performance and memory numbers into estimated energy consumption in Table 5.2. Even with this conservative model, our monitored applications retain between 44% and 78% of the original, unmonitored energy savings. Finally, Table 5.3 shows the accuracy of each monitor. The monitors detect 8–100% of the errors caught by a high-overhead offline monitor with low false positive rates (at most 2.5%).

The rest of this section describes the model for energy consumption and then discusses each application and monitoring scheme in detail.

5.7.1 Energy Model

To evaluate the energy overhead of quality monitoring, we reuse the energy model from the evaluation of EnerJ [52]. The model quantifies the normalized energy consumed by the CPU and memory systems during an entire program execution. Our modeling technique assumes a hardware substrate capable of enabling approximation for each instruction and each cache line as in Truffle [22]. Unless otherwise mentioned, examples use EnerJ's default (medium) energy settings—or, in the cases where applications attempt to train their ideal energy level, this is the initial setting.

The model assumes a fixed energy balance between the processor core, the on-chip SRAM structures (registers and cache), and DRAM (main memory). Each type of instruction—integer, floating point, load, and store—is assigned a cost, a fraction of which is reduced when the instruction is approximate. The energy reduction from approximate SRAM and DRAM is proportional to the fraction of the application's memory footprint that is approximate. Details can be found in [52].

For each program, we consider four configurations: fully precise (the baseline), approximate without monitoring, fully precise with monitoring, and approximate with monitoring. The difference between the approximate executions with and without monitoring reflects the energy "given back" to enable quality monitoring. Although a monitored precise execution is not useful in practice, it shows the overall energy

overhead of monitoring.

To compute the relative energy usage of these configurations, we start by computing the energy usage of the unmonitored exections as in [52]. We then record the additional instruction counts from the monitored executions. We scale the processor energy by the increase in the number of instructions executed and scale the memory energy by the increase in the time that the memory must remain active. In most cases, the latter is also represented by the increase in instruction count. However, in one case (the Asteroids application), the execution time does *not* increase because the application uses `sleep` calls to maintain the proper frame rate. Thus we did not scale its memory energy. We apply the above energy scaling factors to the precise unmonitored execution to determine the energy usage of the precise monitored execution. We then apply the approximation scaling factors from the EnerJ model to the precise monitored energy to determine the approximate monitored energy level. We do this using the precise-approximate instruction and memory breakdown from the monitored execution, since this may differ from the breakdown of the unmonitored execution.[5]

5.7.2 Ray Tracer

We first investigated the approximate ray tracer from [52], which renders a scene consisting of a plane with a checkerboard texture. We applied precise sampling with a sampling rate of 1% around the computation of each pixel. We use RGB color distance [47] to measure QoR and a constraint function that requires that the distance be at most 5% of the maximum RGB color distance.

Our monitor had an energy overhead of 17.2% (see Table 5.2). The pixel computation kernel is very small in this simple ray tracer because it assumes that the

[5]Note that this difference in instruction and memory mix is also the reason that we cannot determine the approximate monitored energy levels by simply applying the scaling factors to the approximate unmonitored energy.

scene consists of a single plane at a known position. A more complex ray tracing kernel would have better justified the cost of the call to `checkApprox` and resulted in lower overheads. Yet, despite this overhead, our monitored ray tracer managed to retain 44% of the energy savings of an unmonitored execution.

Our precise sampling approach also gave an accurate estimate of the QoR of the approximate ray tracer. The mean average error of the monitor's estimate of the number of pixels that were off by more than 5% was 9.6%. The range of error across all runs was between 0.3% and 17.6%. Our false positive rate was 0%, as expected.

5.7.3 Ray Tracer: End-to-end System

To demonstrate the utility of monitoring in practice, we also built an end-to-end system on top of our monitored ray tracer. This system takes advantage of the fact that certain areas of the image are more susceptible to errors than other areas (e.g., areas with smaller features). Our end-to-end monitored application increases the energy whenever the error rate of sampled pixels gets above a configurable maximum threshold over a window of samples. Similarly, we lower the energy if the error rate drops below a configurable minimum threshold. It might also be beneficial to modify the thresholds (or lengthen the windows) if ping-ponging between energy levels is detected, but we did not implement this for our proof-of-concept.

Our end-to-end system reduced the error rate to 4.6%, compared with a rate of 8.6% for the monitored ray tracer without automatic adjustments. In addition, the end-to-end system used slightly less energy than the regular monitored system (84.8% of the precise energy usage, compared with 85.5% for the system with just monitoring). Different strategies for adjusting energy would clearly have led to different results. For example, choosing different error thresholds could result in reduced energy and increased errors, or vice versa.

5.7.4 Asteroids

Our next application was a version of the classic Asteroids game [29]. We added approximation by allowing the array that stores positions and velocities to be stored in approximate memory. This approximation reduced the energy usage by 8.8%.[6] We placed the kernel that updates positions inside a `CheckApproxCodeBlock` and referenced the position-velocity array in the output argument to `checkApprox`.

We tried two varieties of verification functions to monitor our approximate Asteroids game: a streaming verifier and a consistent output verifier. The streaming verifier compares the positions of the asteroids and the ship with their last known positions and verifies that they have not moved by more than the maximum velocity. To reduce overhead, we record only every fifth output and multiply the maximum allowed move distance by the number of frames since the frame we are comparing against. We also used a custom output copier that copies only the necessary pieces of the position array. Our consistent output verifier, on the other hand, merely checks that the velocities are in the allowable range and that the positions are within the screen bounds.

We also explored an end-to-end use case of monitoring in the context of the Asteroids game. To do this, we added hooks to the EnerJ runtime that allow developers to adjust the simulated energy levels of the processor and memory. We then set up our constraint function to check whether the detected error rate was higher than 0.002% of positions. Subjectively, we found that this error rate was sufficient to make the game very playable: most games had no noticeable errors, and the games that did have noticeable errors had only one or two non-fatal (to the player's ship) errors. When our constraint function detected a higher error rate, we raised the energy. Once our monitor detected that the error rate had stayed below the desired rate for 1000 (for streaming verification) or 2000 (for consistent output verification) frames, we declared

[6] More aggressive approximation could have saved more energy. E.g., we could have approximated some of the calculations in addition to the storage.

the training phase over. The higher count was necessary for consistent output verification because it detects fewer errors (see Table 5.3) and would occasionally declare that we were done too quickly.

After declaring training complete, we turned the monitor off. We let the game run for 10,000 frames to capture an adequate mix of of pre- and post-training energy savings. We were able to retain about 55% of the original energy savings with both verifiers. In the limit (as we increase the number of frames executed) the savings retained will approach 100%. Streaming verification did slightly better than consistent output verification (despite a higher overhead, as seen in Table 5.1) because it was able to settle on the correct energy level more quickly and thus turn off monitoring sooner. If we look at just the overhead during the training phase, consistent output verification's overhead of 19.2% was better than streaming verification's overhead of 30.0%. Both approaches had no false positives.

5.7.5 Triangle Intersection

We also looked at the approximate version of the JME triangle intersection kernel described in [52]. We used EnerJ's aggressive approximation/energy levels to get the error rates high enough to be interesting for our purposes. At these levels, we saved about 17% of JME's energy usage with an error rate of 5.2%. We surrounded the triangle intersection code with a call to `checkApprox` that used a traditional verification function. The key insight of the verification function was that triangles that are close together are more likely to intersect than triangles that are far apart. So, our first attempt at a traditional verifier picked a point on each of the two triangles and computed the Euclidean distance between them. If the distance between them was high, and the computation returned `true` (intersection), the monitor declared a possible error. Similarly, if the distance was small, and the computation returned `false`, we declared a possible error. This verifier retained 45% of the original energy savings. Adding code to correct the errors (by re-executing) reduced the savings

retained to 29%. Our error rate with this correction applied dropped to 2.7%.

We noticed, however, that almost all of the monitor-flagged computations were for cases where we declared an intersection between two triangles that were far apart. These cases were almost always real errors. On the other hand, the few hits we got for nearby triangles that the computation determined to not intersect were a mixed bag—many were false positives. So, we changed our verifier to look only for the far-apart/intersecting case. We then corrected errors we caught by declaring that they did not intersect (since we only flag erroneous intersections). The combination of cheaper monitoring and cheaper correction allowed us to retain 78% of the energy savings, with a false positive rate of 0.2%. We detected 47.7% of errors, and our correction reduced the error rate to 2.7%.

5.7.6 Sobel Filter

Our next application used fuzzy memoization to monitor a Sobel filter kernel. The Sobel filter is a matrix convolution used to estimate an image's intensity gradient and is frequently used in edge detection. We approximated the local variables used in the luminance and convolution computations and achieved an energy savings of 14.4% relative to a precise execution. We surrounded the inner loop iteration that computes the gradient at each point with our `checkApprox` monitoring call.

Our prototype fuzzy memoization implementation (Section 5.6.4) uses 2D (one input and one output) linear interpolation between two nearby previous inputs to determine a predicted output. We then compare this against the actual output to estimate QoR. Thus we needed to reduce the output gradient vector to a single value and reduce the 9-pixel input to a single input value that will correlate with the chosen output value. For the output, we chose the magnitude of the gradient vector (this is what edge detectors look at). For the input, we summed the absolute values of the differences between the north and south neighbors and the east and west neighbors. Our constraint accepted the computed value if it was within 60 of the predicted value

(we found empirically that this threshold gave good results). Whenever the monitor indicated a potential quality violation, we re-executed the computation precisely.

Our monitor successfully identified and corrected 86.7% of the erroneous computations, reducing the error rate from 0.66% to 0.09%. This reduction was achieved with an overhead of just 7%, and allowed us to retain nearly 50% of the original energy savings. Our false positive rate was just 2.5%.

5.7.7 FFT Kernel

We created two monitors for the approximate version of the FFT kernel described in [52]. We used the moderate energy savings levels for memory and mantissa width but the mild settings for functional unit voltage, as the error rate is over 80% otherwise. Our first monitor used consistent output verification. After every tenth iteration[7] of the inner FFT kernel loop, we check that the elements of the vector are within the maximum possible range, based on the size of the input. Our second monitor used fuzzy memoization to predict the magnitude of the output of the FFT from the magnitude of the input.

Both monitors caught most of the errors, with a low rate of false positives. In particular, the consistent output verifier caught *every* error where the mean squared error of the output was greater than 0.1,[8] and had no false positives. Our fuzzy memoization monitor was able to catch 90.1% of errors with a false positive rate of just 1.3%. Both monitors had overheads in the 7–8% range, and retained 60–70% of the energy savings.

---

[7]Our decision to check every ten iterations, rather than just at the end, was guided by the insight we gleaned from our offline instrumentation (Chapter 4). Namely, an early error in the FFT computation can cascade and cause many later errors.

[8]We have to pick some "good-enough" threshold, because using an approximation engine that includes narrowing the floating point mantissa means you will never get an exact match on an FFT computation.

5.7.8 Black Scholes.

Finally, we implemented consistency verification monitoring for a Java port of the PARSEC Black Scholes benchmark. We simply check if the option value is within the maximum possible range, and if not, declare an error. Our monitor caught over 65% of the errors, and reduced the error rate from 3.68% to 1.26%. It retained 44.4% of the original energy savings from approximation.

5.8 Related Work

Many systems have proposed to trade off output quality to improve performance or energy consumption using both software [5,27,53,61] and hardware techniques [12,17, 22,23,32,37,42]. Run-time QoR monitoring helps make these approximate computing techniques more applicable by controlling their resulting quality degradations.

This work is the first (to our knowledge) to explore the design space of dynamic quality monitoring for approximate computations and to implement a framework supporting multiple approaches to monitoring. Here we review related efforts to understand or control the impacts of approximation on QoR.

Green [5] is a framework for controlling approximation that can, optionally, invoke user code on a sampling of executions to assess quality. The programmer must provide an appropriate monitoring scheme. One example application uses a manual implementation of precise sampling (with no support for controlling side effects). Our work is complementary: it explores the design space of monitoring schemes and provides reusable implementations for a variety of practical approaches.

PowerDial [27] also dynamically controls an application's degree of approximation. It monitors run-time conditions (e.g., real-time deadlines) and adjusts quality accordingly. Similarly, Eon [54] adjusts system energy at runtime based on the availability and cost of energy and computational resources. Whereas those systems monitor resource consumption, this work focuses on monitoring quality.

Quality-of-service profiling [40] uses offline profiling runs during development to examine the QoR impact of unsound code transformations. The offline calibration steps in Green and PowerDial work similarly. Online quality monitoring, as in our work, requires efficient mechanisms that do not overwhelm the benefits of approximation.

Recent work on modeling and analysis seeks to mitigate the quality impact of approximate computation. Carbin et al. [9] propose a proof system for verifying programmer-specified correctness properties in relaxed programs. Other work [39, 61] uses probabilistic reasoning to prove accuracy bounds on relaxed program transformations. EnerJ [52] provides a simple noninterference guarantee. These static techniques protect fundamental safety properties. But dynamic monitoring is critical to addressing run-time quality degradation that static techniques cannot rule out.

Previous work [2, 13] uses approximate (or fuzzy) memoization to *provide* approximation rather than to check the quality of approximation. In that setting, fuzzy memoization can be more expensive—since it replaces a baseline computation instead of augmenting it—but must also be more accurate.

Blum and Kannan's *program correctness checkers* [7] bear some similarities to our verification function monitors. Their checkers aim to prove the correctness of a program output after the program has executed, rather than to check the quality during execution. They are also mostly concerned with proving correctness rather than maintaining low overhead. However, the basic principle is the same, and some of their ideas may prove useful in designing developer-specified verification functions for our monitors.

Chapter 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

Dynamic tools can play key roles in providing and maintaining quality of result for approximate computations. These roles can span the entire lifecycle of an approximate application. In Chapter 3, we demonstrated the utility of dynamic tools during the prototyping phase. Our EnerCaml system allows users to prototype, experiment with, and tune approximate algorithms. In Chapter 4, we showed how instrumentation-based dynamic tools can aid in debugging and quality tuning of approximate applications. Our dataflow tool allows developers to determine the amount of approximation that flows into the computation of results, and our correlation tool shows them which approximate operations are most highly correlated with the quality (or lack thereof) of the application's results. Finally, Chapter 5 demonstrated that approximate applications can take advantage of real-time online quality monitoring without totally sacrificing the energy-saving benefits of approximation. We proposed and prototyped three approaches to monitoring, and mapped out the design space of monitors.

Just as static and dynamic tools complement each other in other aspects of software development, we view our dynamic tools for approximate application quality as a key addition, and complement, to the static tools currently available for controlling and understanding the quality of approximate programs.

6.2 Future Work

The research described in this thesis suggests numerous interesting directions for future work. We enumerate a few such ideas here:

- We propose further refinements to the approximation monad scheme described in Section 3.6. In particular, we would like to add autotuning capabilities similar to those present in the EnerCaml code-centric system. Instead of turning off approximation at function applications, a data-centric monad-based autotuner could disable approximation for particular uses of the monad. This could be implemented by either replacing the approximation function with the identity function or by storing a flag in the monad that indicates that the approximation function should be skipped at bind and endorse sites. Either of these approaches could be taken by an automated profiler that experiments with various combinations of precise and approximate data.

- We also propose investigating hardware models designed for the code-centric approximation described in Chapter 3. If the hardware supported appropriate sandboxing of data [15, 16, 35, 55, 57], it could be possible to safely approximate many more operations than we currently do. For example, we could approximate address calculations and control flow comparisons. We imagine future hardware with low-powered approximate cores that execute most or all of their operations approximately. With sufficient sandboxing, this could be done without danger of corrupting the rest of the execution.

- The correlation vectors generated by the tool described in Section 4.3 could be fed into a machine learning algorithm. This could in turn help developers better understand the relationships between approximate operations in their code and output quality, and suggest possible improvements.

- The correlation vectors (Section 4.3) and dataflow vectors (Section 4.2) could potentially be used to guide autotuners (like that described in Section 3.4). Both techniques point to potential sources of approximation quality problems. The interesting challenges here would be figuring out how to automatically translate

this information into alternate annotations (and potentially endorsements) in a data-centric approximation environment.

- For the verification function monitors described in Section 5.3.2, we propose providing a domain-specific language for some common properties, such as a specified range for the checked computation's output, or a specified relationship between the input parameters and the output. To provide full generality, however, we will still need to allow verification routines to be written in the language of the application (e.g., OCaml for EnerCaml or Java for EnerJ).

- We propose exploring the tradeoffs of alternate fuzzy memoization strategies (Section 5.3.3). Our prototype utilized two-dimensional linear interpolation due to its relatively low overhead, but other strategies may be able to effectively learn approximate functions with fewer precise data points, thus making up for their potentially higher overhead by requiring fewer precise runs and less storage. One particularly intriguing technique is the memo-binning approach described in Section 5.6.4.

- We also propose exploring additional monitoring approaches within the design space described in Section 5.4. One approach that we considered but ultimately did not try was Constraint Checking. In this approach, the developer would specify various constraints on relationships between inputs and outputs (or just outputs) of a monitored approximate computation. We did not attempt this because we felt it was too similar to the verification function approach described in Section 5.3.2. However, if we combined this with static or dynamic analysis to automatically infer these constraints, it could potentially by a powerful approach.

- We would also like to look at adding a more flexible API for specifying output

objects for our online monitors. The current API requires all objects that may be modified to be specified up front, in an array. A more flexible API would allow dynamic additions to the list during the computation, as we traverse paths through data structures.

- Finally, we would like to enhance our online monitors to handle execution length side effects. For example, an approximation may cause the program to enter an infinite loop that would never occur during a precise execution.

BIBLIOGRAPHY

[1] Anant Agarwal, Martin Rinard, Stelios Sidiroglou, Sasa Misailovic, and Henry Hoffman. Using code perforation to improve performance, reduce energy consumption, and respond to failures. Technical Report MIT-CSAIL-TR-2009-042, MIT, 2009.

[2] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Transactions on Computers*, 54(7):922 – 927, July 2005.

[3] `https://ocaml.janestreet.com/?q=node/100`, 2011.

[4] Jason Ansel, Yee Lok Wong, Cy Chan, Marek Olszewski, Alan Edelman, and Saman Amarasinghe. Language and compiler support for auto-tuning variable-accuracy algorithms. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 85–96, Washington, DC, 2011. IEEE Computer Society.

[5] Woongki Baek and Trishul M. Chilimbi. Green: a framework for supporting energy-conscious programming using controlled approximation. In *ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010.

[6] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: hardware design in Haskell. In *1998 International Conference on Functional Programming*, 1998.

[7] Manuel Blum and Sampath Kannan. Designing programs that check their work. In *Proceedings of the Twenty-first Annual ACM Symposium on Theory of Computing*, STOC '89, pages 86–97, New York, NY, 1989. ACM.

[8] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[9] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Reasoning about relaxed programs. In *2012 Conference on Programming Language Design and Implementation*, June 2012.

[10] Michael Carbin, Sasa Misailovic, and Martin C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *2013 ACM international conference on Object oriented programming systems languages and applications*, 2013.

[11] Jacques Carette, Lydia E. van Dijk, and Oleg Kiselyov. Syntax extension for monads in OCaml. `http://www.cas.mcmaster.ca/~carette/pa_monad/`, 2008.

[12] 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 *Proceedings of the 2006 Conference on Design, Automation, and Test in Europe*, 2006.

[13] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *ACM SIGSOFT 2011 Symposium on the Foundations of Software Engineering*, 2011.

[14] `www.coverity.com`.

[15] Jedidiah R. Crandall and Frederic T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 221–232, Washington, DC, 2004. IEEE Computer Society.

[16] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: A flexible information flow architecture for software security. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 482–493, New York, NY, 2007. ACM.

[17] Marc de Kruijf, Shuou Nomura, and Karthikeyan Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *2010 ACM/IEEE International Symposium on Computer Architecture*, 2010.

[18] Marc de Kruijf and Karthikeyan Sankaralingam. Exploring the synergy of emerging workloads and silicon reliability trends. In *Silicon Errors in Logic—System Effects*, 2009.

[19] `http://www.cs.washington.edu/homes/miker/enercaml`, March 2012.

[20] Dan Ernst, Nam Sung Kim, Shidhartha Das, Sanjay Pant, Rajeev Rao, Toan Pham, Conrad Ziesler, David Blaauw, Todd Austin, Krisztian Flautner, and Trevor Mudge. Razor: a low-power pipeline based on circuit-level timing speculation. In *The Thirty-sixth Annual IEEE/ACM International Symposium on Microarchitecture*, 2003.

[21] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Thirty-eighth International Symposium on Computer Architecture*, 2011.

[22] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[23] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *The Forty-fifth Annual IEEE/ACM International Symposium on Microarchitecture*, 2012.

[24] Krisztian Flautner, Nam Sung Kim, Steve Martin, David Blaauw, and Trevor Mudge. Drowsy caches: simple techniques for reducing leakage power. In *The Twenty-ninth International Symposium on Computer Architecture*, 2002.

[25] `http://www.ffconsultancy.com/languages/ray_tracer/comparison.html`, 2007.

[26] Simon Frankau and Alan Mycroft. Stream processing hardware from functional language specifications. In *Hawaii International Conference on System Sciences*, 2003.

[27] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[28] Steven Curtis Johnson. Lint, a C Program Checker. *Unix Programmer's Supplementary Documents*, Volume 1, 1986.

[29] Matthias Kalisch. Asteroid field. `http://jcolorexpansion.sourceforge.net/asteroid_field.html`.

[30] Animesh Kumar. *SRAM Leakage-Power Optimization Framework: a System Level Approach*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2008.

[31] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *2004 International Symposium on Code Generation and Optimization with Special Emphasis on Feedback-Directed and Runtime Optimization*, San Jose, CA, Mar 2004.

[32] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *Proceedings of the 2010 Conference on Design, Automation, and Test in Europe*, 2010.

[33] Xuanhua Li and Donald Yeung. Exploiting soft computing for increased fault tolerance. In *Workshop on Architectural Support for Gigascale Integration*, 2006.

[34] Xuanhua Li and Donald Yeung. Application-level correctness and its impact on fault tolerance. In *Thirteenth International Symposium on High-Performance Computer Architecture*, 2007.

[35] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, 2011. ACM.

[36] Yanbing Li and Miriam Leeser. HML: An innovative hardware description language and its translation to VHDL. In *Conference on Hardware Description Languages*, 1995.

[37] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[38] `http://ocsigen.org/lwt/`, 2008.

[39] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *Eighteenth International Static Analysis Symposium*, 2011.

[40] Sasa Misailovic, Stelios Sidiroglou, Hank Hoffman, and Martin Rinard. Quality of service profiling. In *ACM/IEEE Thirty-second International Conference on Software Engineering*, 2010.

[41] Alan Mycroft and Richard Sharp. The FLaSH project: Resource-aware synthesis of declarative specifications. In *International Workshop on Logic Synthesis*, 2000.

[42] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *Proceedings of the 2010 Conference on Design, Automation, and Test in Europe*, 2010.

[43] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *The ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation*, 2007.

[44] `http://top500.org/lists/2013/11/`, November 2013.

[45] `http://caml.inria.fr/ocaml/index.en.html`, March 2012.

[46] Elliott Oti. Collision detection: triangle-triangle intersection. `http://www.elliottoti.com/index.php?p=28`, August 2007.

[47] Thiadmer Riemersma. Colour metric. `http://www.compuphase.com/cmetric.htm`, 2012.

[48] Martin Rinard. Using early phase termination to eliminate load imbalances at barrier synchronization points. In *2007 ACM international conference on Object oriented programming systems languages and applications*, 2007.

[49] Martin Rinard, Henry Hoffman, Sasa Misailovic, and Stelios Sidiroglou. Patterns and statistical analysis for understanding reduced resource computation. In *Onward!*, 2010.

[50] Michael F. Ringenburg and Dan Grossman. Atomcaml: first-class atomicity via rollback. In *2005 International Conference on Functional Programming*, 2005.

[51] Mehrzad Samadi, Janghaeng Lee, D. Anoushe Jamshidi, Amir Hormati, and Scott Mahlke. Sage: Self-tuning approximation for graphics engines. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 13–24, New York, NY, 2013. ACM.

[52] 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 *Thirty-second ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.

[53] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffman, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *2011 ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 2011.

[54] 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 *Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007.

[55] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 85–96, New York, NY, 2004. ACM.

[56] Don Syme. F#: Taking succinct, efficient, typed functional programming into the mainstream. In *2010 ACM international conference on Object oriented programming systems languages and applications*, 2010.

[57] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 109–120, New York, NY, 2009. ACM.

[58] Jonathan Ying Fai Tong, David Nagle, and Rob A. Rutenbar. Reducing power by optimizing the necessary precision/range of floating-point arithmetic. *IEEE Transactions on Very Large Scale Integration Systems*, 8(3):273–285, June 2000.

[59] Cristophe Troestler. n-body OCaml program: Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/u32/program.php?test=nbody&lang=ocaml&id=1`, January 2012.

[60] Vicky Wong and Mark Horowitz. Soft error resilience of probabilistic inference applications. In *Silicon Errors in Logic—System Effects*, 2006.

[61] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate

computations. In *Thirty-ninth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.