# Dynamic Analysis of Approximate Program Quality

Michael F. Ringenburg[1] Adrian Sampson[1] Isaac Ackerman[2] Luis Ceze[1] Dan Grossman[1]
University of Washington
{miker,asampson,luisceze,djg}@cs.washington.edu[1] h2ccoch3@gmail.com[2]

## Abstract

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 trade-off can have unexpected effects on computation quality. This paper presents dynamic-analysis tools to debug and monitor the quality of approximate computations. We propose both offline tools that instrument code to determine the key sources of error and online tools that monitor the quality of deployed applications.

We present two offline and three online 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.

We present prototype implementations of these tools and describe their usage with seven applications. Our monitors succeed in controlling output quality while still maintaining significant energy efficiency gains, and our offline tools succeed in providing new insights into the effects of approximation on output quality.

## 1. Introduction

Energy efficiency has become a critical component of computer system design. Battery life is a major concern in mobile and embedded devices; power bills make up a large part of the cost of running data centers and supercomputers; and the dark silicon problem limits the amount of usable chip area [9].

*Approximate computing* is a promising approach that allows systems to trade accuracy for energy efficiency or performance [2, 3, 8, 11, 12, 17, 21, 22]. These techniques exploit applications' tolerance to occasional errors and provide imprecise but efficient computation for certain parts of the program. For example, a lossy image compression algorithm can tolerate some small errors: users are unlikely to notice minor image imperfections and lossy codecs already compromise on image fidelity. To exploit this tolerance, the algorithm can selectively use unreliable hardware components or unsound code transformations. For example, reducing the DRAM refresh rate saves energy at the cost of occasional memory errors [16] and loop perforation skips some loop iterations [22].

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 [2, 3, 8, 17, 21]. In Relax [8], programmers mark regions of code where hardware errors can safely go uncorrected. In EnerJ [21], 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 [3] propose a proof system for reasoning about acceptability properties in the face of imprecision. The Rely system [4] 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 noninterference), some approximations can be more pernicious than others in terms of their effect on the program's *quality of result* (QoR). Hence dynamic tools are important too. This is analagous to conventional software development, where static tools like Coverity [7] or Lint [13] and dynamic tools like Valgrind [20] all play an important role in ensuring software quality.

This paper 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 offline tools that instrument programs to determine the critical data locations and code points that have the most impact on QoR. We then propose online tools that dynamically monitor quality and can let programs self-heal by adjusting approximation levels or re-executing code in response to quality degradations. We argue that both styles (online and offline tools) are important pieces of an approximate programming ecosystem. The offline tools, while too heavyweight for usage in deployment (the costs would more than overwhelm the savings from ap-

proximation), 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). Our contributions include:

- a tool for dynamically tracking approximate dataflow,
- a tool for determining correlations between approximate operations and output quality,
- three approaches to online quality monitoring, and
- a framework for online monitoring and side effect management.

In Section 2, we describe a simplistic strawman approach to quality control and consider why it is too heavyweight for the online setting and too weak for offline investigations. Section 3 discusses two styles of offline code instrumentation that can be used to identify critical code points and data locations. Section 4 introduces three approaches to low-overhead online monitoring and shows how they can be viewed as lightweight relaxations of our strawman approach. Two of our approaches (precise sampling and verification functions) are similar to previous proposals, but we believe the third (fuzzy memoization) is novel in the context of online monitoring. Next, Section 5 describes the APIs and usage of our prototype systems. Section 6 describes salient implementation details of both systems. Finally, Section 7 describes our experiences using the prototypes to investigate and control the QoR of approximate applications. Our tools typically allow us to prevent 50–100% of the errors due to approximation, and our online monitors retain 44–78% of the original energy savings from approximation.

## 2. The Quality-Control Strawman

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 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 approximate portion(s) of the application 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 these are inappropriate for the online setting and insufficient for the offline setting. They thus serve as "quality strawmen" to motivate the rest of our paper.

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 would effectively execute this as:

```
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 a couple 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, 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.

Section 4 presents solutions to the cost problem that each work for different application scenarios by proposing three relaxations of this complete monitoring scheme that make it cheap enough to run "always-on" in production. Any approach to QoR monitoring will also 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 5.2) and transparent side effect isolation (Sections 4.4 and 6.2).

In the offline case, on the other hand, the cost of this 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.

We propose approaches that let us track approximation and errors at a much finer grain (individual operations or variables) and to correlate them with output quality. These tools thus help us find the source of the quality issue, rather than merely to determine that an issue exists.

## 3. Offline QoR Instrumention

This section describes two offline tools for investigating the QoR of approximate applications. The first, *dataflow instrumention* (Section 3.1), tracks the number of approximate operations that flow into each approximate variable. Programmers can query these values via an API to determine how much approximation has flowed into each result. The second, *correlation instrumentation* (Section 3.2), 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.

### 3.1 Dataflow Instrumentation

In many approximate applications, there can be wide variance in how many approximate operations flow into the computation of various values. This variance is not always proportional to the savings provided by computing those values approximately, however. In these cases, the approximate computation of values can have far greater impact on QoR 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 (c.f. the Sobel filter application in Section 7). If the input image is approximate data, computing the min and max 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 likely comprises only a small portion of the energy usage of the program, it may have a very large impact on 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 an 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. For store operations, we add code that stores the counter associated with the store's value operand into a shadow memory. 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.

We also provide an API for developers to access the shadow counter values of user variables. This is described in Section 5.1.

### 3.2 Correlation Instrumentation

In many approximate applications, particular *approximate code points* (operations that are executed approximately) are more likely to cause poor QoR 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 these 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 a QoR and execution and error counts for every approximate code point. 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 approxi-

mate results. For stores, we precisely store an identical value into a shadow memory (e.g., a hash table keyed on memory addresses—see Section 6.1), 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.[1] 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.

This approach is not context-senstive—we are merely tracking correlations to individual program counters. 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, it would greatly increase the number of counters and could obscure some correlations.

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 5.1.

# 4. Online QoR Monitoring

Next, we turn our attention to online quality monitoring. In contrast with offline tools, online monitors are designed to detect and compensate for QoR problems in the field. As code that is deployed with the application, these tools must have very low overheads. If the cost of monitoring outweighs the savings from approximation, developers would be better off running precise code rather than monitored approximate code.

As we have seen, the "strawman" approach to monitoring (Section 2) is too expensive. This section presents three more realistic approaches that limit generality to achieve tenable overheads. They can all be viewed as low cost relaxations of the strawman.

## 4.1 Precise Sampling

The first approach we consider is *precise sampling*. Like our strawman, precise sampling compares the results of the precise and approximate versions of the monitored code. Unlike the strawman, 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 the strawman.

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

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 precise and 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 4.4.

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 7), precise sampling could not guarantee that asteroids *never* jump across the screen.

## 4.2 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* a result at significantly lower cost than *computing* the result. In contrast to precise sampling, this approach lowers the cost of the strawman by reducing the cost of each check rather than reducing the number of checks.

We consider three forms of verifier functions, each of which requires different inputs. The first form, *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 input (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 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 func-

tion 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 a pixel's value is similar to its value in a previous frame.

### 4.3 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. [6] and Alvarez et al. [1] 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 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 address 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 enhance the model with data from additional 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 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())
      // Rerun precisely, update memos
    else throw new FailedQoR();
}
image[x][y] = result;
```

Here, the `addMemo` call adds a new result to the model and `compNearestMemos` finds nearby memoized results to compare with our current result.

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). A self-balanced binary search tree suffices. 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.

### 4.4 Handling Side Effects

Monitored computations 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.

Unanticipated side effects can harm quality in ways that the developer-specified metrics 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 identify 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* 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 6.2, can be done relatively cheaply.

We contend that ignoring side effects pushes too much of the burden onto the developer and that ensuring identical side effects creates too much overhead. Therefore, our prototype monitor pursues the third option, as discussed in Section 6.2. Note that side effects are less of an issue in the offline case, as we generally work at the granularity of the entire application.

## 5. APIs and Usage

This section describes the usage of our offline (Section 5.1) and online (Section 5.2) QoR tools. Due to space constraints, we have elided some details of the monitoring APIs. These additional details can be found in the supplemental material. For historical reasons, our offline tools are implemented in C and our online tools in Java, but the ideas are language-agnostic.

### 5.1 Offline QoR APIs

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, ...);
```

Developers can also access the counter values with the following API:

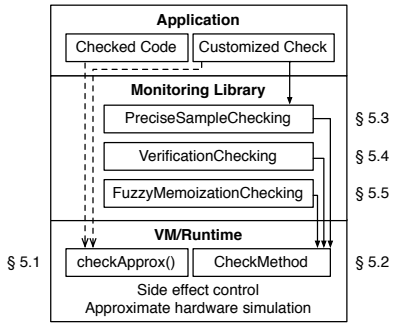



**Figure 1.** The architecture of our monitoring framework. Solid arrows indicate inheritance; dashed arrows indicate parameters.

```
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 stores a vector consisting of all of the correlation counters, as well as a floating-point QoR. It also resets the counters to 0, in case we wish to track correlations and qualities 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 QoR.[2] These APIs were sufficient for all of the use cases described in Section 7, but we also provide some additional APIs as described in the supplemental material.

### 5.2 APIs for Online Monitoring

We have designed a runtime system for monitoring the quality of, and restricting the side effects in, monitored Java code blocks. Our system is built on top of EnerJ [21] and is flexible enough to support all approaches presented in Section 4.

Our system is divided into the three layers depicted in Figure 1. As in any layered system, the orthogonality and

[2] 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.

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, maintaining side-effect restrictions, and copying objects to support reexecution. 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. Section 5.2.1 describes 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 5.2.2 then describes the interfaces that the middle layer provides to the top layer for different forms of monitoring.

#### 5.2.1 The Bottom Layer

Developers monitor code with our `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 monitor:

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

The `argList` array contains the inputs to the checked computation. The `outputObjs` array 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 neither allocated by the checked computation nor members of this list are considered illegal side effects and cause an exception. Finally, `checkM` specifies a method of quality monitoring, via the `CheckMethod` interface.

The `CheckMethod` interface is a flexible interface whose implementations correspond to different monitoring approaches. Details of the interface are available in the supplemental material. The next subsection describes implementations of this interface that support our three proposed approaches: precise sampling, verification functions, and fuzzy memoization.

#### 5.2.2 Middle Layer: Quality Monitors

The `PreciseSampleChecking` class (an implementation of `CheckMethod`) provides monitoriong via precise sampling. Developers provide a sampling frequency as well as an implementation of an interface encapsulating a QoR metric and a constraint. The `QoR` method computes the QoR by comparing the precise outputs and return value with the approximate outputs and return value. The `constraint` method returns true if quality is acceptable. This 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.

The `VerificationChecking` class implements verification function monitoring. 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 and a `constraint` method that returns true if the QoR 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 $n$th 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.

Finally, we implement fuzzy memoization monitoring via the `FuzzyMemoizationChecking` class. We 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. The second interface specifies error thresholds for accepting a monitored computation's result and for triggering additions or adjustments to the model. When developers are confident that quality is high, adding points to the fuzzy model may improve it. However, this must also be balanced with the requirement that the model's overhead not grow too high. Adjusting the model can also be useful if we believe that the model's predictions are poor, e.g., due to discontinuities in the output of the monitored function. These interfaces are described in more detail in the supplemental material.

## 6. Implementation Issues

This section describes some of the most interesting details of our QoR tool implementations. Section 6.1 describes implementing shadow memories for our offline approaches. Section 6.2 discusses handling side effects in monitored code, including controlling their impact on quality and buffering/rollback for approaches that may require multiple executions.

### 6.1 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. 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.[3] 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.

### 6.2 Side Effects in Monitored Code

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 we call `checkApprox`, 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 or specified as output data sets its region number to the number of the current region. 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 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. If copy-on-write is necessary (e.g., if we are doing the first execution of a sampled execution), 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 the copies and unset the copy-on-write flag. This allows the subsequent run to start as if from scratch.

[3] 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.

## 7. Use Cases

To evaluate our dynamic QoR tools, we experimented with seven approximate applications. For three of them, we added both instrumentation and monitoring,[4] for one we added just instrumentation, and for three we added just monitoring. For two programs, we created two monitored versions using different monitoring approaches, resulting in a total of eight monitoring configurations.

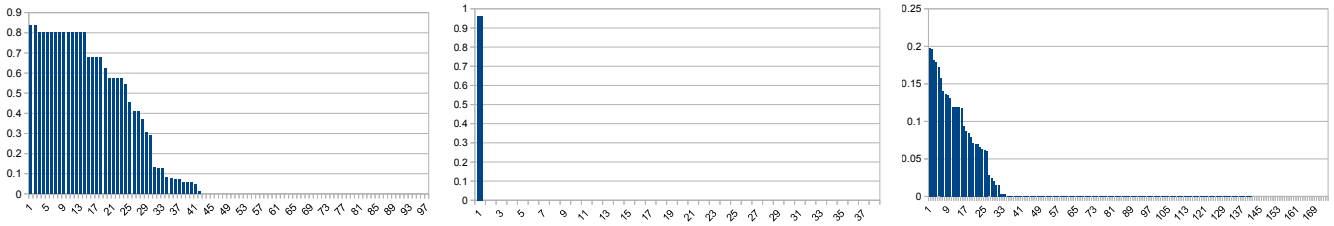
Section 7.1 describes the approximate applications we used. Section 7.2 discusses the insights gained from our offline tool. Finally, Section 7.3 details our online monitoring results.

### 7.1 Applications

We considered the following seven approximate applications. These cover a wide variety of approximation use cases, ranging from scientific computing, to image processing, simulations, and even games:

- We used precise sampling online monitoring with an approximate **simple ray tracer** (from [21]) that traces a scene consisting of a plane with a checkboard texture.
- We used two types of verification function monitoring (streaming and consistency) with an approximated version of the classic **Asteroids** game [14]. We added approximation by placing the object position/velocity array in approximate storage.
- We used traditional verification to monitor the quality of the approximated **JME triangle intersection kernel** from [21].
- We used both dataflow and correlation instrumentation to debug the QoR of an approximate **Sobel filter** application

[4] Note our offline tools were built on top of the LLVM-based EnerC infrastructure for approximate computing in C/C++, and our monitoring prototype was built on top of the EnerJ infrastructure for approximate computing in Java, so we were able to use both tools only in applications where we could find similarly coded C and Java versions.



(a) (b) (c)

**Figure 2.** Graphs showing correlations between code points and QoR in (a) `canneal`, (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.

and used fuzzy memoization monitoring to monitor and correct the remaining errors.

- We used dataflow instrumentation to better understand the approximation patterns of an approximate **FFT kernel** (from [21]) and used this to inform our design of two online monitors: a consistency verification monitor and a fuzzy memoization monitor.
- 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 a monitor was not necessary (errors often improved the result).
- We used correlation isntrumentation to debug the quality of an approximate version of the PARSEC **Black Scholes** benchmark (and a Java port of it). We then built a consistency verification monitor to detect any remaining errors.

### 7.2 Offline Tool Results

***Results summary.*** Our offline tools allowed us to narrow in on the key quality issues in our applications and to better understand their characteristics. This section describes our experiences with the Sobel filter, FFT, simulated annealing, and Black Scholes applications. The graphs in Figure 2 show how correlations are distributed in the applications that used correlation instrumentation. In one case, we were able to solve an intermittent segmentation fault. In others, we were able to identify and correct the issues that most commonly led to poor output QoR, and in another, we were able to achieve some insights that led us to design a better monitor. In all cases, we reached important insights through the use of our offline tools. These insights helped us improve quality, understand behavior, and design better monitors.

***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 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 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 compuation, 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.

***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 nothing 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.

| 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 1.** The percentage of instructions and memory dedicated to the original computation (compute) and the monitoring (check) for each application and online monitor. We measure these dynamically, as total instructions executed and total memory footprint of the monitored versus unmonitored applications.

| 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 2.** The modeled energy consumption of each monitored application. Energy is measured relative to the precise, unmonitored execution energy (the **Precise** column). The **Approx** column shows the energy use of an unmonitored approximate execution. The **Precise Monitored** column shows the energy use 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 use 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. We measured the Asteroids application for 10,000 frames (see Section 7.3). 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.

***Black Scholes.*** In Black Scholes, our correlation instrumentation identified two locations with particulary 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.

### 7.3 Monitoring Results

***Results summary.*** The overheads of our monitored applications appear in Table 2. The final column displays the percentage of the original energy savings that are retained with monitoring, which ranges from 44% to 78%. A more detailed breakdown of overheads is in Table 1. Table 3 shows the accuracy of our monitors. They detect 8–100% of the errors caught by a high-overhead offline monitor with low false positive rates (at most 2.5%). This section discusses our experiences using monitors with our approximate applications.

***A Note on our Energy Model.*** To evaluate the energy overhead of online monitoring, we reuse the energy simulation model from the evaluation of EnerJ [21]. The model quantifies the normalized energy consumed by the CPU and memory systems during an entire program execution. This technique assumes a hardware substrate capable of enabling approximation for each instruction and each cache line as in Truffle [11].

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 online monitoring. Although a monitored

precise execution is not useful in practice, it shows the overall energy overhead of monitoring. To compare monitored and unmonitored 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.

***Ray tracer.*** For the ray tracer, we applied precise sampling with a sampling rate of 1% around the computation of each pixel. Our overheads were relatively high due to the simplicity of the pixel computation kernel (it only traces a scene with a single known plane and texture). Despite the overheads, we still managed to retain nearly 50% of the original energy savings. Our sampling also achieved an accurate estimate of the QoR. The mean average error of the monitor's estimate of the number of bad pixels was just 9.6%. The range of error across all runs was between 0.3% and 17.6%.

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 decreases approximation 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. This 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, it used slightly less energy than the regular monitored system (84.8% of the precise energy usage, versus 85.5% without adjustment).

***Asteroids.*** We tried two varieties of verification functions to monitor our approximate Asteroids game: a streaming verifier and a consistent output verifier. Both check the results after every time step. 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. Our consistent output verifier simply 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. 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 the training phase over. The higher count was necessary for consistent output verification because it detects fewer errors (see Table 3). After declaring training complete, we turned the monitor off. For our results, we let the game run for 10,000 frames to capture an adequate mix of of pre- and post-training energy savings. Streaming verification did slightly better than consistent output verification 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 was better.

***Triangle intersection.*** Our traditional verifier for triangle intersection was based on the insight that triangles that are close together are more likely to intersect than triangles that are far apart. Thus, a traditional verifier could pick a point on each of the two triangles and compute the Euclidean distance between them. If the distance between them was high and the computation returned `true` (intersection), the monitor could declare a possible error. Similarly, if the distance was small and the computation returned `false`, we could also declare an error. We quickly noticed, however, that cases where we declared an intersection between two triangles that were far apart were almost always real errors, but cases where nearby triangles don't 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 flagged erroneous intersections). This optimization allowed us to retain 78% of the energy savings, with a false positive rate of just 0.2%. We detected 47.7% of errors, and our correction reduced the error rate from 5.2%to 2.7%.

***Sobel filter.*** For the Sobel filter, we used fuzzy memoization to monitor the computation of each pixel gradient. For our memoization input, we summed the absolute values of the differences between the north and south neighbors and the east and west neighbors. For the output, we chose the magnitude of the gradient vector (this is what edge detectors look at). 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

| 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 3.** The percentage of errors caught by our online 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 sampled versus real errors. 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.

the computation precisely. Our monitor successfully identified and corrected 86.7% of the erroroneus 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%.

***FFT.*** Based on the insights from our offline tools, we designed a consistent output verification monitor for the FFT kernel. Rather than applying the verification only at the end of the computation, we checked the intermediate results after every 10 iterations. Our verifer checks that every element of the array is within the maximum possible range based on the size of the input array. This allows us to catch errors early, rather than continuing with a computation that is bound to have poor QoR. In addition, we also implemented a fuzzy memoization monitor that predicts the magnitude of the output array based on the maginitude of the input array. Both monitors caught over 90% of the errors, had no false positives, and retained over 60% of the energy savings.

***Black Scholes.*** Finally, we implemented consistency verification monitoring for Black Scholes. We simply check if the option value is within the maximum possible range, and if not, declare an error. Our monitor reduced the error rate from 3.68% to 1.26%, and retained 44.4% of the energy savings.

## 8. Related Work

Many systems have proposed to trade off output quality to improve performance or energy consumption using both software [2, 12, 22, 24] and hardware techniques [5, 8, 10, 11, 15, 16, 19]. Run-time QoR monitoring and debugging tools help make these approximate computing techniques more applicable by letting programmers understand and control their resulting quality degradations.

***Online quality monitoring.*** Our monitoring 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 [2] 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 [12] 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 [23] adjusts system energy at runtime based on the availability and cost of energy and computational resources. Whereas those systems monitor resource consumption, our work focuses on monitoring quality.

Quality-of-service profiling [17] 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.

Previous work [1, 6] 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.

***Offline quality debugging.*** A complementary problem to online QoR monitoring is quality-oriented debugging. 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. [3] propose a proof system for verifying programmer-specified correctness properties and other work [18, 24] uses probabilistic reasoning to prove accuracy bounds on program transformations. EnerJ [21] provides a simple noninterference guarantee. The Rely system [4] 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 runtime events that static analyses cannot rule out.

The aforementioned work on quality-of-service profiling [17] describes an exhaustive search process for identifying program loops that do not need full precision. In contrast, our instrumentation approaches apply to finer-grained sources of error without resorting to brute-force search.

## 9. Conclusion

Dynamic tools should play a key role in providing and maintaining quality of result for approximate computations. As we have shown in this paper, offline instrumentation tools are useful for obtaining a better understanding of quality tradeoffs during pre-deployment development and testing, and online monitoring tools can dynamically monitor and correct for quality degradations in deployed approximate applications. Just as static and dynamic tools complement each other in other aspects of software development, we view our dynamic tools as a key addition to the tools available for using approximate computing.

## References


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

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

[3] M. Carbin, D. Kim, S. Misailovic, and M. C. Rinard. Reasoning about relaxed programs. In *PLDI*, June 2012.

[4] M. Carbin, S. Misailovic, and M. C. Rinard. Verifying quantitative reliability for programs that execute on unreliable hardware. In *OOPSLA*, 2013.

[5] L. N. Chakrapani, B. E. S. Akgul, S. Cheemalavagu, P. Korkmaz, K. V. Palem, and B. Seshasayee. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE*, 2006.

[6] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. Navidpour. Proving programs robust. In *FSE*, 2011.

[7] Coverity source code security tool. `www.coverity.com`.

[8] M. de Kruijf, S. Nomura, and K. Sankaralingam. Relax: an architectural framework for software recovery of hardware faults. In *ISCA*, 2010.

[9] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

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

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

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

[13] S. C. Johnson. Lint, a C Program Checker. *Unix Programmer's Supplementary Documents*, vol. 1, 1986.

[14] M. Kalisch. Asteroid field. `http://jcolorexpansion.sourceforge.net/asteroid_field.html`.

[15] L. Leem, H. Cho, J. Bau, Q. A. Jacobson, and S. Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *DATE*, 2010.

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

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

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

[19] S. Narayanan, J. Sartori, R. Kumar, and D. L. Jones. Scalable stochastic processors. In *DATE*, 2010.

[20] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[21] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI*, 2011.

[22] S. Sidiroglou, S. Misailovic, H. Hoffman, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

[23] J. Sorber, A. Kostadinov, M. Garber, M. Brennan, M. D. Corner, and E. D. Berger. Eon: a language and runtime system for perpetual systems. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007.

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