Monitoring and Debugging the Quality of Results in Approximate Programs

Michael Ringenburg
Adrian Sampson
University of Washington
{miker,asampson}@cs.washington.edu

Isaac Ackerman
Cornell University *
ica23@cornell.edu

Luis Ceze Dan Grossman
University of Washington
{luisceze,djg}@cs.washington.edu

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 accuracy for significant gains in energy efficiency. However, debugging the actual cause of output quality problems in approximate programs is challenging. This paper presents dynamic techniques to debug and monitor the quality of approximate computations. We propose both offline debugging tools that instrument code to determine the key sources of output degradation and online approaches that monitor the quality of deployed applications.

We present two offline debugging techniques and three online monitoring mechanisms. The first offline tool identifies correlations between output quality and the execution of individual approximate operations. The second tracks approximate operations that flow into a particular value. Our online monitoring mechanisms are complementary approaches designed for detecting quality problems in deployed applications, while still maintaining the energy savings from approximation.

We present implementations of our techniques and describe their usage with seven applications. Our online monitors control output quality while still maintaining significant energy efficiency gains, and our offline tools provide new insights into the effects of approximation on output quality.

Categories and Subject Descriptors D.2.5 [*Testing and Debugging*]: Debugging aids; D.2.5 [*Testing and Debugging*]: Monitors

Keywords Approximate computing; debugging; monitoring

* Research performed while at the University of Washington.

ASPLOS '15, March 14–18, 2015, Istanbul, Turkey..

ACM 978-1-4503-2835-7/15/03. . . $15.00.
http://dx.doi.org/10.1145/2694344.2694365

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 the dark silicon problem limits the amount of usable silicon resources [9].

Approximate computing is a promising approach that allows systems to trade accuracy for energy efficiency or performance [2, 3, 8, 10, 15, 22, 27, 28]. 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 [20] and loop perforation skips some loop iterations [28].

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, 22, 27]. In Relax [8], programmers mark regions of code where hardware errors can safely go uncorrected. In EnerJ [27], 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. The situation is analogous to conventional software development, where static tools like Coverity [7]

or Lint [16] and dynamic tools like Valgrind [24] play important and complementary roles in software quality.

This paper presents tools that use dynamic analysis to help debug and control the QoR of approximate programs. We first propose *offline* debugging tools that instrument programs to determine the critical data locations and code points that most affect QoR—to the best of our knowledge, these are the first tools to address debugging approximate code. We then propose *online* mechanisms to monitor quality and let programs adjust approximation levels or re-execute code.

We argue that both styles are important pieces of an approximate programming ecosystem. Offline tools, while too heavyweight for usage in deployment, are well-suited for pre-deployment debugging and QoR investigation. They help programmers better understand why QoR suffers and where approximation is valuable. Conversely, online monitoring can be lightweight enough to run in deployed code and adaptively adjust approximation levels or correct erroneous results when faced with QoR issues that arise post-deployment (due, for example, to unanticipated program inputs or variations in approximate hardware).

Our contributions include:

- a debugging tool for dynamically tracking approximate dataflow,
- a debugging tool for identifying correlations between approximate operations and output quality,
- three approaches to online quality monitoring, and
- an extensible online monitoring framework.

Section 2 describes a strawman approach to quality control, and explains why it is too heavyweight for the online setting and too weak for offline debugging. Section 3 discusses two novel styles of offline approximate code instrumentation and tracing that can be used to identify problematic code points and data locations for debugging. We also describe where these fit into the design space of offline approximation debugging tools. Section 4 introduces three approaches to low-overhead online monitoring. Two of our approaches (precise sampling and verification functions) are similar to previous proposals, but, to the best of our knowledge, the third (fuzzy memoization) is novel in the context of online quality monitoring. Section 5 describes the design space of online quality monitors, and where our approaches fit into this space. Section 6 describes salient implementation details of our systems. Section 7 describes our experience with using the prototypes to investigate and control QoR, including three use cases where we combined offline debugging with online monitoring. Using our tools in concert typically lets us prevent 50–100% of the errors due to approximation while retaining 44–78% of the original energy savings. The final two sections discuss related work and conclude.

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 rate of correct classifications.

One way to measure QoR is to run the approximate portion(s) of a program twice for each input—once approximately, once precisely—and compare results. In an offline setting, this could be done repeatedly in a controlled environment with a variety of inputs. We refer to this as *approximation profiling*. In an online setting, we could do this in real time with each input seen "in the wild." We refer to this as *complete online monitoring*. This section argues that approximation profiling is insufficient and complete online monitoring is inappropriate.

For example, consider evaluating approximate pixel computations in a ray tracer:

```
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();
```

In an online tool, this approach provides real-time quality updates as each approximate computation completes. This enables programs to respond immediately, e.g., by adjusting energy parameters, or by re-executing erroneous computations. Unfortunately, such an approach also has fundamental problems. First, the code assumes idempotence of the monitored code block, but approximate computations can and often do have side effects. 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 less distortion but which are distorted in different ways. Another example is an 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*. Executing twice clearly violates this requirement.

Section 4 addresses the online cost problem by proposing three significantly cheaper monitoring schemes, each

of which work for different application scenarios. 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 (see [25] for details on our API) and with a framework that provides transparent side effect isolation (see Sections 5.3 and 6.2).

In the offline case, on the other hand, cost is not an issue. During pre-deployment quality testing and debugging, spending extra time and energy to improve performance in the field is wise. In fact, we can do much more than just approximation profiling. The strawman tells us only *what* the final QoR was, with no reason *why*. It gives no indication of which operations or data are critical to QoR. Developers need more program introspection when working with approximation. We propose approaches that let us track approximation and errors at a much finer grain (individual operations or variables) and correlate them with output quality. These tools thus help identify the source of a quality issue rather than merely determining that an issue exists.

3. Offline QoR Debugging

This section describes two offline tools for investigating the QoR of approximate applications. *Dataflow instrumentation* (Section 3.1) tracks the number of approximate operations that flow into each approximate variable. *Correlation instrumentation* (Section 3.2) tracks the number of times each approximate operation executes and the number of times it produces an incorrect result. By running the instrumented program multiple times, we can identify the operations and errors that are most correlated with poor QoR and thus may represent quality bugs.

3.1 Dataflow Instrumentation

There can be wide variance in how many approximate operations flow into the computation of various values. This variance may not be proportional to the savings provided by approximation. In such cases, the approximate computation of values can degrade QoR much more than the energy savings would justify. For example, an image transform may compute a scaling factor for its output by traversing the input to compute the difference between the maximum and minimum pixels. If the input image is approximate data, this computation requires many approximate operations. If any such operation experiences a bit flip, the scaling factor and hence *every* output pixel will be incorrect. Thus, even though this computation likely comprises only a small portion of the program's energy usage, it can have disastrous impact on QoR.

Such scenarios motivate 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 O_1 *flows* into R if $O_1 = O_2$ *or* if O_1 flows into one of $i_1, ..., i_n$. Our tool tracks the number of approximate operations that flow into each result and memory location.

We built our tool on top of 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, 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 to store the counter associated with the store's value operand into a shadow memory (after adding one if the store itself is approximate). For load operations, we retrieve the associated counter from the shadow memory 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 parameters and results.

We also provide API-level access to these counters. See [25] for details.

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 always executes may have much more impact than one in a rarely used control flow branch. 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 vector coordinates are most highly correlated with QoR.

Our instrumentation maintains two counters for each approximate code point (approximate LLVM bytecodes in our implementation). One counter records the number of times the code point executes. The other counts the number of errors, determined as follows: For operations other than memory references, we re-execute the operation precisely and compare the result. 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), along with the code point performing the store. At loads, we look up the loaded address in the shadow memory and compare the result to the approximately loaded value. If there is a mismatch, we can charge the error counters of the store (obtained from the shadow memory), the load, or both. This decision can be programmer-driven or chosen by the tool. Here we have assumed a model where approximate data is only accessed via

approximate loads *and* approximate stores. We can generalize this by simply using the shadow memory for *every* store.

This approach is not context-sensitive—we are tracking correlations to individual program counters. Nothing conceptually prevents adding context sensitivity, but we have not found it necessary for the applications we studied. It would greatly increase the number of counters and could obscure some correlations.

3.3 Offline Debugging Design Space

Our offline debugging approaches can be viewed as flow analysis (dataflow instrumentation) and statistical correlation (correlation instrumentation). To our knowledge, these are *the* two general ways of assigning blame to a program point. Variants on these approaches are possible, including:

- **Context sensitivity**: As mentioned above, our correlation instrumentation approach is not context-sensitive: We analyze code points inside library functions as single points even for common functions like `memset` where uses from some callers may correlate more with QoR than uses from other callers.
- **Interactivity**: Our prototypes do not offer user interactivity. For example, one could imagine supporting *breakpoints* (like in a debugger) and providing access to our counters while execution is paused.
- **Granularity of instrumentation**: Our prototypes instrument at a fine-grained level (IR operations for correlation, and individual expression results for dataflow). Aggregating information at coarser granularity may provide complementary insights.

4. Online QoR Monitoring

Next, we turn our attention to online quality monitoring in order 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 monitoring cost outweighs the approximation savings, it is trivially better to run precise code rather than monitor approximate code.

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.

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

```
res = tracePx(x, y);
if (random() < sampleFreq) {
  prec = runPrecise {tracePx(x, y);}
  if (!compare(result, prec, approxOut,
               preciseOut))
    throw new FailedQoR();
}
image[x][y] = res;
```

Here `compare` is a developer-supplied function that compares precise and approximate results to decide whether the QoR is acceptable. The `approxOut` and `preciseOut` arguments capture any memory side effects of the two executions. This is left intentionally vague here; side effects are an orthogonal issue discussed in Section 5.3.

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. Precise sampling also assumes good quality inputs to the monitored computation, otherwise the precise run may not generate a high quality output to compare with.

Our monitoring framework and API (described in [25]) includes support for precise sampling with customized QoR metrics and sampling parameters. We discuss example use cases in Section 7.3.

4.2 Verification Functions

Verification functions are routines supplied by the developer to check the QoR of a computation. Verification functions are useful whenever we can *check* a result at significantly lower cost than *computing* the result.

Our framework supports three distinct forms of verification functions. Each form requires different inputs and is useful in different situations. The first, *traditional verification*, verifies the outputs of the current execution based on its inputs. For example, a 3-SAT verifier could check that the variable assignments satisfy the formula clauses. The second, *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. 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. Like precise sampling, traditional verification functions rely on good quality inputs to the computation, since the inputs are used to check the output. The other two forms rely only on outputs, so they are less sensitive to problems with the input.

Our monitoring APIs (described in [25]) support all three forms of verification function.

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. We estimate the QoR by checking how different the current output is from the predicted output. Similar ideas have been used by Chaudhuri et al. [6] and Alvarez et al. [1] to *provide* approximate execution rather than to check execution quality.

We identify several variations distinguished by their prediction mechanisms. The simplest, which we call *simple fuzzy memoization*, predicts the previously recorded output with the most similar input. Another variation (*interpolated fuzzy memoization*) performs interpolation between a set of similar previous inputs. Finally, *learned fuzzy memoization* applies machine learning techniques to learn the relation between inputs and outputs. 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). Our framework implements the interpolated style.

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 early in the program execution to seed the model with some initial values, and with random sampling 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.

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

```
if (preciseSeedingRun()) {
  result = runPrecise { tracePx(x, y); }
  addMemo(x, y, result, sideEffects);
} else {
  result = tracePx(x, y);
  if (!compNearestMemos(x, y, result,
                        sideEffects))
    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.

Efficient fuzzy memoization requires storing previous results as input–output pairs that can be efficiently retrieved when we encounter nearby inputs. A self-balanced binary search tree ($O(\log n)$ lookup) worked well in our prototype.

Our monitoring APIs (described in [25]) provide support for interpolated fuzzy memoization.

5. The Online Monitoring Design Space

We now consider the broader design space of online quality monitoring. We begin with a discussion of the dimensions of the design space (Section 5.1). We then mention a possible additional dimension—code-centric versus data-centric annotation—and argue why code-centric is the better choice (Section 5.2). Finally, we discuss how to handle side effects and re-execution (Section 5.3).

5.1 Design Space Dimensions

We can describe the design space of online 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—must it be reducible to a number that we verify is close to an expected value, 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 5.3 to discussing it.

Approach	What Is Checked	Checking Parameters	Applicability and Checkable Quality Properties
Precise Sampling	Sampled executions	Approximate and precise outputs	Applicable to code that can be re-executed. Desired properties must not require examining every execution.
Verification Functions	Every execution		Applicable when checking a result is cheaper than computing it.
Traditional		Approximate inputs and outputs	
Streaming		Current and old approximate outputs	
Consistency		Approximate outputs	
Fuzzy Memoization	Every execution	Current and old inputs and outputs	Applicable to computations with easily learnable outputs. The desired property must be representable as a distance metric.

Table 1. How each of our monitoring approaches fits into the full design space. Handling side effects (not shown) is orthogonal.

Certain regions of this space are undesirable or impossible. For example, requiring the results of a precise computation combined with checking every execution leads to the prohibitive overhead of our strawman. Similarly, it is impossible to use sampling to provide error bounds on every execution.

Table 1 fits the approaches from Section 4 into this space.

5.2 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.

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:

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

Here `checkComp` will call `computePixel(args)` and pass its result to `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 *monitoring* 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.

Code-centric monitoring has three important advantages:

Developer expectations. Code-centric annotations apply to the result of a computation, which is generally what programmers 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 monitors 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 loosen the semantics of data-centric annotations to check the value at only certain points, they essentially become code-centric annotations.

Implementation of monitoring and recovery. Limiting checking to explicit points in the computation simplifies the implementation of the monitoring framework. Monitor-

ing 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, such as re-execution, in response to quality monitoring.

5.3 Dealing With 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, violating 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 checked by the monitor, then an equivalent precise execution would have produced the same modification. In the general case this requires significant, high-overhead, dynamic cooperation from the runtime system.
- **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 runtime cooperation, but can be done relatively cheaply (as shown in Section 6.2).

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. Thus our monitoring framework pursues the third option, as discussed in Section 6.2.

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. More details can be found in [25].

6.1 Shadow Memories

Both 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.[1] Stores correspond to hash table inserts and loads to table lookups. Key reinsertions (i.e., stores to a previously-used address) replace the old value.

6.2 Side Effects in Monitored Code

Our monitoring system restricts side effects by allowing monitored code to write only to objects that are either part of the output list or local to the monitored code. Other memory writes result 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 it is built on top of the EnerJ simulator [27], which already tracks per-object state for other purposes.

In particular, our prototype tracks whether objects are writable by augmenting heap-allocated objects with per-object state containing a region number. This state is updated and tracked by the same runtime routines that manage our framework's dynamic monitors. Specifically, users of our framework monitor computations by wrapping the computation in an implementation of a function object interface that we provide. They then pass the computation to our primary monitoring API [25], `checkApprox`. They may also optionally provide a list of *output data*: objects that the computation is allowed to modify.[2] When we call

[1] 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.

[2] If an output list is specified, our framework also automatically passes it to the QoR metric, verification function, or fuzzy memoization predictor, so that modifications to these objects may also be monitored.

`checkApprox`, our framework creates a new region by incrementing a global region counter. To support nested calls, each call to `checkApprox` also 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. If any output objects were specified, we set their region number to the number of region we are entering. Similarly, if any objects are allocated inside a checked region, we set their region number to the number of the current region. When entering a nested region, we first check that any output objects were 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 from a region, we reset the output objects' region numbers to the parent region number. Thus, to enforce side effect restrictions, all stores to heap objects inside monitored code must simply check that the destination object's region number matches the current region number. If there is a mismatch, we throw an exception.

Our implementation requires a static list of output/side-effectable objects, which we found adequate for all of the use cases we examined. If necessary, the implementation could made more flexible by allowing users to register objects-that-can-change midway through an approximate computation. Also note that this approach requires non-reversible side-effects to be buffered as writes to side-effectabe objects.

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 to the copy (initially null). If buffering is needed (e.g., during the first execution of a sampled execution), we iterate through the list of objects marked as outputs (i.e., the objects for which side effects are allowed) 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.

7. Use Cases

To evaluate our dynamic QoR tools, we experimented with seven approximate applications. For three, we used offline debugging in concert with online monitoring,[3] for one we used just offline debugging, 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 our approximate applications. Section 7.2 discusses insights gained via our offline debugging tools. Section 7.3 details our online monitoring results.

7.1 Applications

We considered the following applications, which cover a variety of approximation use cases, including scientific computing, image processing, simulations, and games:

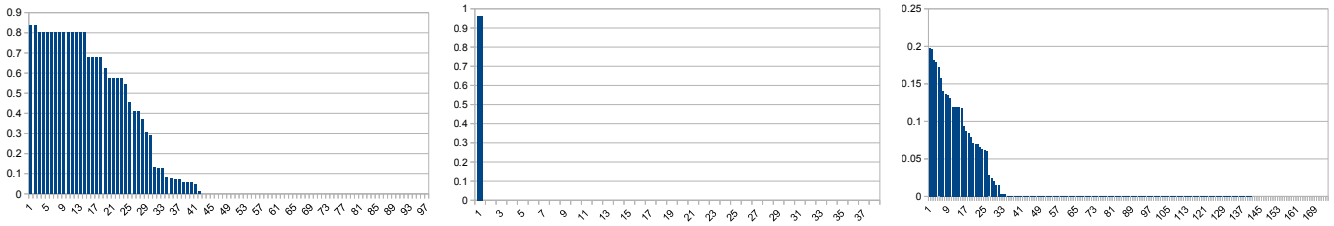
- We used precise sampling online monitoring with an approximate **simple ray tracer** (from [27]).
- We used two types of verification function monitoring (streaming and consistency) with an approximated version of the classic **Asteroids** game [17].
- We used traditional verification to monitor the approximated **JME triangle intersection kernel** from [27].
- We used both dataflow and correlation instrumentation to debug the QoR of a **Sobel filter**, and used fuzzy memoization to monitor and correct the remaining errors.
- We used dataflow instrumentation to better understand the approximation patterns of an **FFT kernel** (from [27]) 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.
- We used correlation instrumentation 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 Debugging Tool Results

Results summary. Our offline debugging tools enabled us to narrow in on the key quality issues in our applications, and to better understand their key characteristics. 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 gain insights that led us to design a better monitor. Figure 1 shows how correlations are distributed in the applications where we used correlation instrumentation. Overall, our tools 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 seg-

[3] 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 1. 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 relatively small number of code points have high correlation to QoR.

mentation 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 a code point where the application was reading from an array. Dataflow instrumentation confirmed that the array index could be influenced by approximate operations.

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 many approximate operations (almost 180K). Due to the structure of the FFT, if an error corrupts an intermediate array value early in the computation, errors can propagate through the rest of the FFT result. This insight led us to design a monitor for the FFT application that catches 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, and convinced us that monitoring was unnecessary for this application.

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. This suggests 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 3. All applications were run five times. The final column displays the percentage of the original energy savings that are retained with monitoring, which ranges from 44% to 78%. The second to last column displays the energy usage of the monitored approximate computation relative to the original precise (and unmonitored) computation. Table 2 contains a detailed breakdown of overheads. Memory overhead is related to the amount of data that must be remembered in order to verify results, and can be reduced through clever monitor design by application developers. Table 4 shows our monitoring accuracy.

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 [27]. 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 [10]. However, our techniques would also work with approximation applied to regions of code instead of individual instructions.

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

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 2. 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 3. 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. Because the Asteroids results include a mix of monitored and unmonitored (post-training) frames, we have included the relative costs of the monitored portion in parentheses in the appropriate columns.

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%.

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% versus 85.5%).

Asteroids. 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 and check only every fifth time step. 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 4). 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 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 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 verifier 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 magnitude of the input array. Both monitors caught over 90% of the errors, had less than 2% 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 are designed to relax output quality to improve performance or energy consumption via software [2, 15, 28, 30] or hardware [5, 8, 10, 11, 19, 20, 23] techniques. Dynamic QoR monitoring and debugging tools help make these approximate computing techniques more feasible by helping programmers understand and control 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.

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

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 4. 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 result in a false positive (i.e., the monitor reports a QoR error that does not occur).

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.

Grigorian et al. [13, 14] propose Light Weight Checks that are similar to the verification functions we describe. Their success reinforces our assertion that this is an important coordinate in the monitoring design space.

PowerDial [15] 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 [29] 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 [22] 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. Our dynamic, instrumentation-based debugging tools pinpoint program points that lead to poor output quality. These techniques complement prior static and 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 [21, 30] uses probabilistic reasoning to prove accuracy bounds on program transformations. EnerJ [27] provides noninterference guarantees. Rely [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 run-time events that static analyses cannot rule out.

There is also relevant work in the field of dynamic instrumentation. Lam et. al. [18] use instrumentation to determine where they can safely reduce floating point precision (a specific form of approximation). Roth et al. [26] and Ganai et al. [12] apply dynamic dataflow analyses to debugging in other environments, namely MPI and multi-threading.

The aforementioned work on quality-of-service profiling [22] 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 are invaluable for debugging and monitoring quality-of-result for approximate computations. We have shown that offline quality debugging is useful for understanding quality tradeoffs during pre-deployment development and testing, and online quality monitoring can detect and correct quality degradations in 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.

Acknowledgments

This work was supported in part by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. It was also supported by NSF award number 1216611 and gifts from Microsoft Research.

References

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

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

[3] Michael Carbin, Deokhwan Kim, Sasa Misailovic, and Martin C. Rinard. Reasoning about relaxed programs. In *PLDI*, June 2012.

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

[5] 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 *DATE*, 2006.

[6] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara Navidpour. Proving programs robust. In *FSE*, 2011.

[7] `www.coverity.com`.

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

[9] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *ISCA*, 2011.

[10] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Architecture support for disciplined approximate programming. In *ASPLOS*, 2012.

[11] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. Neural acceleration for general-purpose approximate programs. In *MICRO*, 2012.

[12] Malay Ganai, Dongyoon Lee, and Aarti Gupta. Dtam: dynamic taint analysis of multi-threaded programs for relevancy. In *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE)*, 2012.

[13] Beayna Grigorian and Glenn Reinman. Dynamically adaptive and reliable approximate computing using light-weight error analysis. In *NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, 2014.

[14] Beayna Grigorian and Glenn Reinman. Improving coverage and reliability in approximate computing using application-specific, light-weight checks. In *First Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014.

[15] Henry Hoffmann, Stelios Sidiroglou, Michael Carbin, Sasa Misailovic, Anant Agarwal, and Martin Rinard. Dynamic knobs for responsive power-aware computing. In *ASPLOS*, 2011.

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

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

[18] Michael O. Lam, Jeffrey K. Hollingsworth, Bronis R. de Supinski, and Matthew P. Legendre. Automatically adapting programs for mixed-precision floating-point computation. In *27th ACM International Conference on Supercomputing*, 2013.

[19] Larkhoon Leem, Hyungmin Cho, Jason Bau, Quinn A. Jacobson, and Subhasish Mitra. ERSA: Error resilient system architecture for probabilistic applications. In *DATE*, 2010.

[20] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS*, 2011.

[21] Sasa Misailovic, Daniel M. Roy, and Martin C. Rinard. Probabilistically accurate program transformations. In *SAS*, 2011.

[22] Sasa Misailovic, Stelios Sidiroglou, Hank Hoffman, and Martin Rinard. Quality of service profiling. In *ICSE*, 2010.

[23] Sriram Narayanan, John Sartori, Rakesh Kumar, and Douglas L. Jones. Scalable stochastic processors. In *DATE*, 2010.

[24] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

[25] Michael F. Ringenburg. *Dynamic Analyses of Result Quality in Energy-Aware Approximate Programs*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Feb 2014.

[26] Philip Roth and Jeremy Meredith. Value influence analysis for message passing applications. In *28th ACM International Conference on Supercomputing*, 2014.

[27] 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 *PLDI*, 2011.

[28] Stelios Sidiroglou, Sasa Misailovic, Henry Hoffman, and Martin Rinard. Managing performance vs. accuracy tradeoffs with loop perforation. In *FSE*, 2011.

[29] 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.

[30] Zeyuan Allen Zhu, Sasa Misailovic, Jonathan A. Kelner, and Martin Rinard. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL*, 2012.