Profiling and Autotuning for Energy-Aware Approximate Programming

Michael F. Ringenburg Adrian Sampson Luis Ceze Dan Grossman
Department of Computer Science and Engineering, University of Washington
{miker,asampson,luisceze,djg}@cs.washington.edu

Abstract

One promising approach to energy-efficient computation, approximate computing*, trades off output precision for gains in energy efficiency. Many applications can easily tolerate small errors, especially if they are handled in a disciplined manner. However, approximation introduces an inherent tradeoff between quality of result and energy efficiency. Existing approaches lack ways to quantify and study these tradeoffs. This paper proposes tools to prototype, profile, and automatically tune the quality of programs designed to run on future approximate hardware. We describe the software layers required in such a system and discuss design considerations. We built an OCaml-based prototype of our set of tools and performed three case studies.*

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 due to power constraints [9].

Approximate computing is a promising approach that allows systems to trade accuracy for energy efficiency or performance. If applications can tolerate occasional errors, hardware can consume less power. For example, reducing the refresh rate of DRAM saves energy at the cost of occasional memory errors [15]. Similarly, we can execute instructions on a low-powered pipeline if we can tolerate occasional logic errors [10].

Many applications have kernels that are amenable to approximation. For example, applications that work with audio, video, or images are inherently error-tolerant—in fact, common media storage formats involve lossy compression. Any code that involves a randomized or approximate algorithm can also tolerate imprecision. However, even the most approximable applications require some code to execute precisely. For example, memory allocation, control flow, and bounds checking must be precise to avoid faults. Some applications also have certain phases that must execute precisely. For instance, while we can often approximate the pixels of an image, approximation in the image header may be catastrophic.

Energy savings from approximate computing typically come from hardware. However, only the application can determine where approximation is appropriate. Thus, a language with support for approximation must allow programmers to distinguish parts of a program—variables, operations, methods, loops, and so on—that are tolerant to error. Examples of such languages include the EnerJ extension to Java [23] and the Rely programming language [4].

Approximate computing represents a tradeoff between energy efficiency and quality of result (QoR). Researchers (and developers) investigating approximate computing need tools to help quantify this tradeoff and understand how much QoR must be sacrificed to achieve desired efficiency gains. Users can also benefit from understanding which portions of their code should be approximate, and which precise, to optimize this tradeoff.

To address these challenges, we propose an architecture for a tool that prototypes, profiles, and autotunes approximate applications designed for *future approximate* hardware using only readily-available *conventional* hardware. Our architecture consists of *approximation*, *profiling*, and *autotuning* layers. The approximation layer is responsible for simulating the effects of approximate hardware. Both the approximation model and the energy cost model are customizable. The profiling layer uses the approximation layer to monitor both the quality lost and the efficiency gained due to approximation. Because QoR is an application-specific measurement, the profiler takes a QoR-evaluation function as input. Finally, the autotuning layer builds on the previous layers to explore alternate precise–approximate decompositions of user code blocks. It searches for points along the Pareto frontier of optimal quality–efficiency tradeoffs. We have implemented this architecture for OCaml programs by modifying the OCaml implementation. We call our tool EnerCaml. It is available at our website [8].

The rest of this paper describes the layers of our architecture in more depth. We also describe our implementation and three case studies. Further details about EnerCaml can be found in [21].

2. Approximation Layer

To determine how a prototype application can leverage approximate computing, we need a way for programmers to indicate where approximation is acceptable. To determine how the application responds to different kinds of approximate hardware, we need a way to run the application such that approximation occurs according to some model. This section describes our design for both needs.

2.1. Code-Centric Approximation

We assume most code will be written assuming precise execution but that some energy-consuming kernels will leverage approximation. We therefore have explicit markers in the program to indicate that the execution of some code block can be approximate. (In languages like OCaml with higher-order functions, such a marker can just be a function taking a function, so an approximate computation `e` looks like `approximate (fun () -> e)`, but the syntax is not essential.) Conversely, users can indicate that a subcomputation of an approximate computation should be precise (e.g., `precise (fun () -> e)`). This code-centric approach has complementary advantages to data-centric approaches that mark approximate data elements instead of code blocks. In prototyping applications, we often found the code-centric approach avoided unnecessary copying of data into and out of kernels.

Even within approximate computations, many operations would lead to crashes and other bad behavior if executed approximately. Examples include control flow and memory management. Therefore, we take a conservative approach to approximate execution and allow imprecision only in arithmetic operations, comparisons, and loads from numeric arrays. Approximate array loads model approximate memory since whether the load or the storage system introduces the error is irrelevant to the application.

When prototyping how approximation can change application behavior, one simply marks approximate sections of code (and precise subsections within them). This purposely simple approach adds only directly relevant work over implementing the original algorithm.

2.2. Simulating Approximation

For understanding and prototyping approximation, we argue against this natural approach: Build an approximate hardware platform (or simulator), write a compiler, run the program, and measure QoR and energy usage. Such an approach gives little feedback in terms that make sense with respect to the high-level algorithm. Moreover, tweaking low-level parameters (e.g., DRAM refresh rate) will likely have inscrutable effects on quality of result.

Instead, we advocate and have implemented a high-level configurable approximation model directly corresponding to the operations visible in the programming language. For each approximated operation, we apply a transformation to the precise output to produce the approximate output. For example, one simple model is that with probability p a load is correct and with probability $1-p$ it is a uniformly random bit-pattern. A model representing approximate arithmetic functional units could incorporate that low-order bits are more likely to be wrong. We expect users to design these models based on complementary research on approximate hardware, allowing a key separation of concerns between high-level application design and low-level hardware design.

An essential advantage of this approach is that making the approximation model configurable is easy: We can provide hooks (e.g., a library API) for users to plug in arbitrary functions to replace each approximate result. For example, to configure our system such that approximate integer arithmetic produces the wrong low-order bit with probability 0.1, one would just run this code:

```
let flip p i = (if ((Random.float 1.0) < p)
                then (i lxor 1) else i)
in set_integer_approximation (flip 0.10)
```

We provide similar hooks to customize the amount of simulated energy saved given a trace of execution events.

3. Profiling Layer

The profiling layer is responsible for estimating the energy savings and quality of result for an execution of an approximate application. This may vary between runs due to different inputs and the randomness present in most forms of approximation.

Quality is measured by comparing an approximate execution with a precise execution with identical inputs. It is inherently application-specific, so the profiling layer must provide a way for users to specify how executions should be compared. This involves specifying the outputs to be compared along with an output comparison function (the *QoR function*). The profiling layer runs the code twice, collects the outputs of both runs, and compares them using the QoR function.

For example, an EnerCaml programmer writing a ray tracer might record the final pixel values for comparison:

```
let _ = record_profile_output g
in Printf.fprintf pgm_file "%c" g
```

and compare them using peak signal-to-noise ratio:

```
let mse prc app = (* Compute mean-squared
  error between lists prc and app *) in
let psnr precL appL = 10. *. (log10
  ((255. *. 255.)/.(mse precL appL))) in
eval_qor psnr
```

The `record_profile_output` function appends its argument to a list of output data specific to the current execution. After the precise execution, the profiling layer saves this list and starts a new list for the approximate execution. At the end of the approximate execution, we apply the QoR function (the argument to `eval_qor`) to the two lists. We then output the computed QoR and an estimate of the energy saved. As mentioned in Section 2, we provide hooks that let users customize the estimation of energy savings from approximation. By default, we use a simple metric proportional to the percentage of approximable operations executed approximately.

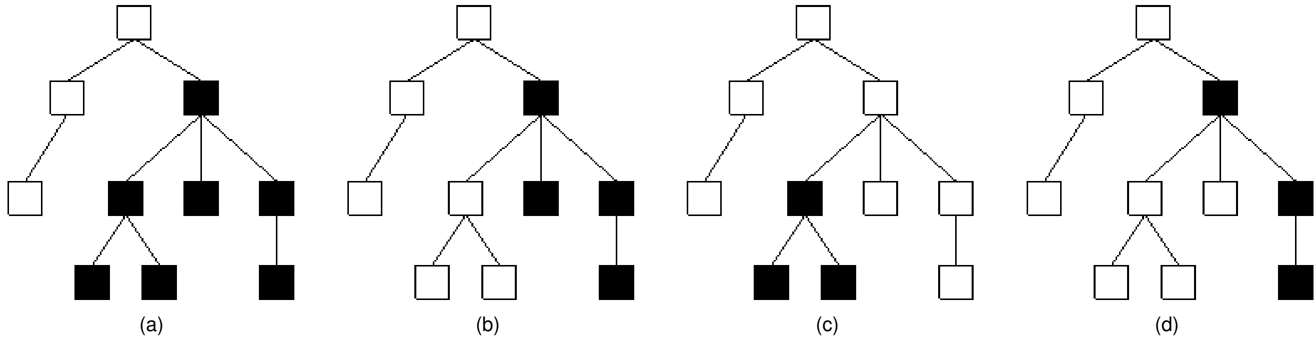

Figure 1: Static call trees illustrating the strategies we propose to search the precise-approximate decompositions of programs for improved quality of service versus efficiency tradeoffs. A black node represents an approximate function application and a white node represents a precise application. Figure (a) shows the originally specified approximation. Figure (b) shows the result of making one call site precise. Figure (c) shows the result of narrowing the approximation to just that same call site. Finally, figure (d) illustrates the result of making two sibling call sites precise.

4. Autotuning Layer

The profiling layer lets programmers investigate the QoR and efficiency implications of their approximate programs. However, to improve QoR–energy tradeoffs, programmers must be able to determine which portions of their code are most amenable to approximation and which should be kept precise. Doing this by hand is tedious and time-consuming due to the number of possible combinations of precise and approximate annotations. The autotuning layer automates part of this process and generates a set of simple code changes that improve QoR and/or efficiency.

The autotuner builds on the profiling system to navigate the search space of alternate precise/approximate decompositions of the original program. The goal is to automatically identify program annotations that offer better efficiency–QoR tradeoffs than an initial annotation provided by the programmer. Using search heuristics, the autotuner generates many alternative program decompositions and profiles each in turn to assess its energy efficiency and QoR. The configurations with the best efficiency and QoR are reported to the programmer.

The autotuner's search heuristics consist of removing approximation from code that was marked as approximate in the original code. We never *add* approximation to code that was originally specified as precise—the programmer's initial annotation bounds approximation to code that can be safely relaxed. Each alternative program decomposition consists of a set of static call sites within an approximate computation that are marked as precise (as if with the `precise` marker). The idea is that programmers can roughly indicate an area where approximation might be appropriate and the autotuner refines the region to improve QoR–efficiency tradeoffs.

Exhaustively considering every subset of the call sites in an approximate computation would create an exponential search space. Thus, the autotuner must use heuristics to choose which call sites to evaluate. We found that the following heuristics (illustrated by the static call trees in Figure 1) worked well in EnerCaml:

- Make a single call site precise.
- Make all call sites in the computation precise except for one, effectively "narrowing" approximation to the chosen site.
- Make a pair of call sites that appear in the same calling function precise. Intuitively, these "adjacent" call pairs are more likely to have a synergistic effect—i.e., the benefit of making them both precise may be more than the sum of the benefits of making them individually precise.

The chosen heuristics represent a tradeoff between autotuning time and the thoroughness of the search. Additional strategies would be easy to add but, in our studies, we found that the above strategies were sufficient.

The autotuner profiles each alternative configuration and collects its QoR and estimated energy savings. If one result has both better QoR and higher energy savings than another result, we say that the former result *dominates* the latter. The tool reports all configurations that are not dominated. This represents the Pareto frontier of the best discovered QoR versus efficiency tradeoffs. Users may also iteratively refine these configurations by rerunning the autotuner. Figure 2 depicts an excerpt of the tool's output, including a textual listing and a graph.

5. The EnerCaml System

We built a prototype of our proposed tool called EnerCaml. EnerCaml allows researchers and developers to prototype approximate applications in OCaml [19] and to customize their approximation and energy models. EnerCaml also contains a QoR–efficiency profiler and autotuner. OCaml is known to be a good tool for prototyping, and its functional style is also a very good fit for our autotuning strategies that vary the approximation at static call sites. This section briefly describes the implementation of EnerCaml and then discusses three case studies.

```
Narrowing approximation to trace.ml, line 16,
character 10:
QOR: 37.644753, Approximation score: 22.282223
...
Narrowing approximation to trace.ml, line 36,
character 13:
QOR: 32.663749, Approximation score: 63.438417
...
Making precise trace.ml, line 55, character 47:
QOR: 28.351986, Approximation score: 94.797524
```

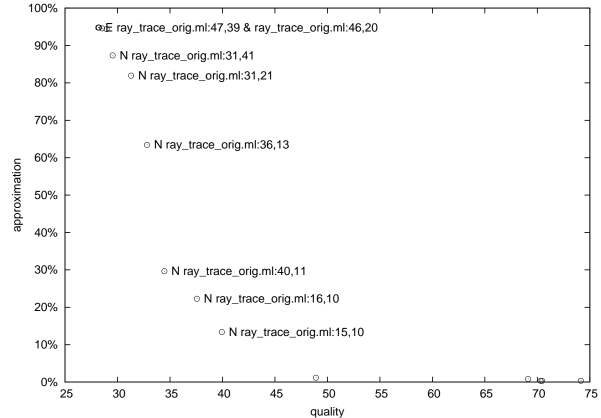

Figure 2: Our autotuner produces a textual and graphical depiction of the best results among the profiled executions.

5.1. EnerCaml Implementation

The EnerCaml implementation tracks precise and approximate execution by compiling two versions of each function: a precise version and an approximate version. The precise version is called whenever we apply the function in a precise context (i.e., inside precise code) or execute the argument of a `precise` call. The approximate version is called whenever we apply the function in an approximate context or execute the argument of an `approximate` call. We track the two versions by adding a second code pointer to each closure. Profiling and autotuning are handled by modifying the interpreter to execute applications multiple times. We track all the approximate function calls (calls that follow the approximate code pointer) and modify them according to our search heuristics in subseqeunt runs. To ensure that output data from previous runs is preserved, we copy it out of the (garbage-collected) OCaml heap and into the C heap used by the runtime.

This approach works well for prototyping and profiling, which is our goal. On real energy-saving approximate hardware, however, it may be less compelling because the extra space required for dual closures would use more energy. Designers of such systems can utilize alternate approaches that transfer execution to approximate cores or track approximate state with hardware bits.

5.2. Case Study: Ray Tracer

Our first EnerCaml case study involved adding approximation to a ray tracer [11]. The ray tracer has two phases: scene creation and ray tracing. First, we approximated scene creation:

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

Next, we approximated ray tracing:

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

We then instrumented the program for profiling and autotuning in order to study the QoR–efficiency tradeoffs. We used the instrumentation shown in Section 3 (record the value of each output pixel and compare with PSNR). We then profiled the ray tracer with our default approximators and cost model (which uses approximation percentage as a proxy for efficiency). Our initial PSNR was 26.9, with 94.8% approximation. This image is shown in Figure 3a. We then ran the autotuner to see if we could improve these results. Figure 2 shows the Pareto frontier found by the autotuner. The first thing that we noticed is that we can obtain better QoR (PSNR of 28.4), while only giving up a very small amount of approximation by making scene creation precise. Intuitively, small changes in the positions of objects can have significant impacts on the errors of some pixels because they can move the boundary between shadowed and non-shadowed pixels. Since most of the computation is in ray-tracing and not scene creation, we made scene creation precise and reran the autotuner.

On our next autotuning run, the most interesting results involved the `ray_sphere` function (which computes the first intersection of a ray and a sphere). We achieved a 29.9 PSNR with 86.3% approximation by narrowing approximation to `ray_sphere`, and a 36.9 PSNR with 22.3% approximation by narrowing approximation to a single dot product in `ray_sphere` (see Figure 3b). We focused our efforts on this code by moving the approximation primitive to the `ray_sphere` call and rerunning the autotuner. We discovered a number of new points along our frontier curve, including 33.6 PSNR with 41.8% approximation (Figure 3c), and 31.5 PSNR with 64.1% approximation. These results were obtained by making either individual calculations or pairs of calculations in `ray_sphere` precise.

It would have required significantly more effort to characterize the effects of approximation without our tools. The autotuner quickly eliminated scene creation due to its large impact on QoR and then pointed us to the importance of `ray_sphere`.

5.3. Case Study: N-Body Simulation

The next application that we looked at was an N-body simulation benchmark [25]. We used a simple QoR metric that calculates the inverse of the average error. The simulation

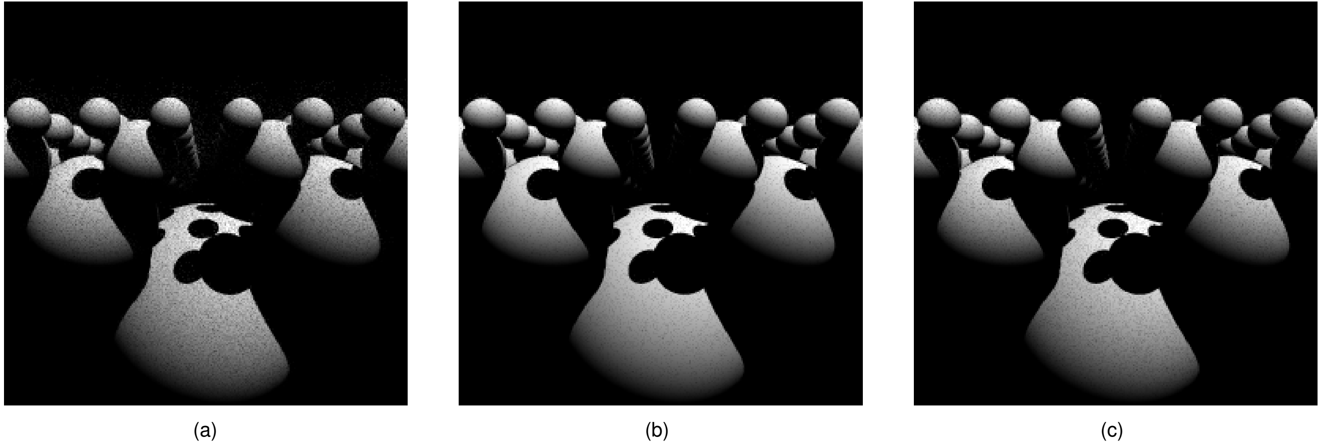

Figure 3: The images generated by our ray tracer with various mixtures of approximate and precise execution. Figure (a) shows the initial approximation, with PSNR 26.9 and 94.8% approximation. Figure (b) shows the image with 36.9 PSNR and 22.3% approximation. Figure (c) shows the image with 33.6 PSNR and 41.8% approximation.

first initializes the system by calling `offset_momentum` and then advances the system one step at a time by calling `advance` in a loop. We made both calls approximate:

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

Initially, our autotuner was not able to tell us very much because the only function applications it identified were the two outer-level calls to `offset_momentum` and `advance`. We looked at the code and discovered that it was written in a very imperative style. We were quickly able to identify various subcomponents of the calculation, and wrap them in function calls. When we reran the autotuner, we found that two of the subcomponents of the calculation could be profitably approximated with very low impact on the quality of result:

```
QOR = 5562.919330
Percent approximated: 24.456460
QOR = 7436.822960
Percent approximated: 24.456460
```

These are significantly better than the original approximation, which had a QOR of roughly 0.01. We also tried approximating both of the identified computations to see if we could get good QoR with a larger fraction of operations approximated. Our results were promising:

```
QOR = 3821.292285
Percent approximated: 48.912917
```

In the case of the N-body simulation, the autotuner allowed us to identify portions of the simulation that could be profitably approximated without significantly impacting the QoR.

5.4. Case Study: Collision Detector

Our third case study was a simple collision detection kernel [20] that checks whether two triangles intersect. Our QoR metric calculates the percentage of correct intersection tests. We added approximation at the call to the intersection check function. We then ran the profiler to get a baseline and achieved 97.81% correctness with 93.96% approximation. We then ran the autotuner to see if we could do better. Most of the proposed changes involved some combination of call sites from four source lines. Two of the lines compute the normals of the planes containing the two triangles. The other two lines use the normals to test whether all three points of one triangle lie on the same side of the plane of the other triangle (indicating no intersection). We experimented with making these computations precise. When we made both of the normal calculations precise, our QoR increased to 98.88% and our approximation rate dropped to 67.1%. When we instead made the no-intersection checks precise, our QoR rose only to 97.94% but our approximation was almost unchanged at 93.0%. When we combined both changes, we were able to detect 98.93% of collisions correctly and still approximate 66.1% of the approximable operations. Compared to the original annotation, we were able to eliminate over 51% of the errors while losing less than 30% of the approximation.

6. Related Work

Many systems have proposed trading off quality to improve performance or save energy using both software [2, 12, 24, 27] and hardware [5, 7, 10, 13, 15, 18] techniques. Several studies have shown that a wide variety of applications can tolerate the resulting imprecision with acceptable results [6, 14, 26]. This work on approximate computing forms the context for tools for managing approximation like the one proposed here.

Some language-level techniques seek to help developers mitigate the effects of approximate semantics. Carbin et al. [3]

propose a proof system for verifying user-specified correctness properties in relaxed programs. The Rely system [4] (which uses a similar error model to ours) bounds the probability that values produced by an approximate computation are correct by examining the static data flow of nondeterministic operations. Misailovic et al. [16] use probabilistic reasoning to prove accuracy bounds on relaxed transformations. EnerJ [23] provides a simple noninterference guarantee. These techniques are static and conservatively bound imprecision. Programmers writing to a relaxed programming model can use them in tandem with dynamic tools like EnerCaml to obtain an empirical picture of quality loss.

The SAGE system [22] implements approximation for CUDA GPU kernels. On of their two phases is a runtime tuning phase that bears some similarities to our autotuning. Their system target a much later phase of the development workflow: runtime tuning of deployed applications in a specific target environment. Thus, they focus on tweaking approximation parameters, rather than exploring which parts of a computation are most amenable to approximation.

The PetaBricks language extensions and compiler features [1] allow developers to auto-tune *variable-accuracy algorithms*, a category which includes approximate algorithms. They focus on performance rather than the quality–energy trade-offs we are looking at, but some of the language extensions they propose may be useful for us to consider.

Quality-of-service profiling [17] identifies code that has little influence on output quality. Programmers can consider relaxing this code to improve performance. In contrast, our tool uses a priori programmer annotations to identify approximate portions of programs that should be made *more* accurate to achieve a desired QoR level. EnerCaml is a closed-loop system that suggests specific code modifications to achieve better energy–quality tradeoffs.

7. Conclusion

This paper proposes an architecture for prototyping, profiling, and autotuning approximate computations. We believe that approximate computing will be a significant factor in improving the energy efficiency of computations in the future. Until now, however, there was a lack of tools to help researchers and developers understand the quality of result versus efficiency tradeoffs that are inherent in approximate computing. This work addresses that pressing need.

Acknowledgements

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.

References

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

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

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

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

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

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

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

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

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

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

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

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

[19] http://caml.inria.fr/ocaml/index.en.html.

[20] Elliott Oti. Collision detection: triangle-triangle intersection. http://www.elliottoti.com/index.php?p=28.

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

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

[23] 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.

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

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

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

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