Quality of Service Profiling and Autotuning for Energy-Aware Approximate Programming

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

July 11, 2012

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 service 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 also present an OCaml-based prototype of our tool suite, and describe three case studies that we performed with it.

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 [7].

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 [13]. Similarly, we can execute instructions on a low-powered pipeline if we can tolerate occasional logic errors [8].

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.[1] 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. One example of such a language is our EnerJ extension to Java [18]. EnerJ programmers annotate data that can be approximated and the type system ensures that approximate data does not flow into precise data without explicit programmer permission.

Approximate computing represents a tradeoff between energy efficiency and quality of service (QoS). Researchers (and developers) investigating approximate computing need tools to help quantify this tradeoff and understand how much QoS 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 QoS is an application-specific measurement, the profiler takes a QoS-evaluation function as input. Finally,

[1] In fact, one of the sample applications we considered for this paper was a genetic algorithm, but it turned out that when we added approximation the results were a *better* fit for the data than when we ran it precisely. Thus it was not an interesting test case for our profiler and autotuner.

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 [6].

The rest of this paper describes the layers of our architecture, as well as how they were implemented in our prototype. Specifically, Section 2 discusses the approximation layer, Section 3 describes the profiling layer, and Section 4 reviews the autotuning layer. We also discuss three case studies in Section 5.

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 QoS 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 service.

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 EnerCaml 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)
```

Users can also customize the technique used to estimate the amount of energy saved via approximation. During execution, the runtime records a trace of approximable events: those for which the system supports error injection. The trace records the kind of each event and whether it was approximate or precise. To model a given hardware technique for approximation, the user can provide a function that processes a trace and returns a "score": a number between 0.0 and 1.0 proportional to the amount of energy hypothetically saved. The system runs the scoring function after every execution to quantify the gain from approximate execution.

To model a hardware approximation technique in our proposed architecture, the user simply supplies error injection and energy quantification functions written in the application's native source language. This flexibility makes it possible to use our approach to evaluate a wide range of approximation techniques.

2.3 EnerCaml's Approximation Layer

EnerCaml is a prototype implementation of our architecture for the OCaml language. Users create approximation

approximate	(unit->'a) -> 'a approx	Executes its thunked argument approximately, wraps the result in an approximate type, and returns it.
continue_approx	'a approx->('a->'b)->'b approx	Takes an approximate value and a function, and approximately applies the function to the value.
endorse	'a approx -> 'a	Transforms its approximately-typed argument into a precisely-typed return value.
precise	(unit -> 'a) -> 'a	Executes its thunked argument precisely, and returns the thunk's result.
lift	'a approx approx-> 'a approx	Lifts an approx approx type to an approx type.
set_float_approximation	(float->float) -> unit	Specifies the float approximation function.
set_integer_approximation	(int->int) -> unit	Specifies the integer approximation function.
set_load_approximation	(int->int) -> unit	Specifies the integer array load approximation function.
set_load_float_approximation	(float->float) -> unit	Specifies the float array load approximation function.

Table 1: The EnerCaml approximation primitives.

in EnerCaml programs by passing a thunked code block to the function `EnerCaml.approximate`, which has type `(unit -> 'a) -> 'a approx`. The EnerCaml system then executes the thunk approximately and returns the result wrapped inside an approximate type. Before using the approximately-typed result in a precise computation, the user must endorse it with a call to a function `EnerCaml.endorse` of type `'a approx -> 'a`. The use of approximate types and explicit endorsements is modeled after EnerJ. It enforces a boundary between approximate and precise computations and requires users to explicitly acknowledge every location where data crosses the boundary from the approximate realm into the precise realm.

For example, consider the following code snippet from a ray-tracer (downloaded from the website of Flying Frog Consultancy [9]), where the function `intersect` is used to determine where a ray intersects a scene:

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

To execute the intersection approximately, we simply write:

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

The call to `approximate` causes the EnerCaml system to simulate executing the intersection computation on approximate hardware (we discuss this in more depth in Section 2.4). The call to `endorse` allows the values returned from the approximate intersection to be used in future precise computations. Alternatively, approximate values can be passed to future approximate computations via the `continue_approx` primitive. The `continue_approx` primitive has type `'a approx -> ('a->'b) -> 'b approx`. It takes an approximate value and a function and approximately applies the function to the value, returning another approximate value.

Table 1 lists the approximation primitives we used in our prototype . We described the `approximate`, `continue_approx`, and `endorse` primitives above. The `precise` primitive allows programmers to specify that certain code should always be executed precisely, even inside an approximate dynamic context. `precise` takes a thunked block of code as its argument and executes it precisely, returning the return value of the thunk. Outside of an approximate dynamic context (or directly nested inside another precise context), the `precise` primitive is simply a direct application of the thunk. We also provide a `lift` primitive that converts an `approx approx` type into an `approx` type. This is useful when an approximate thunk returns the result

of a nested approximate thunk, resulting in an `'a approx approx` when we would prefer an `'a approx`. This could be handled by the `endorse` primitive, but that would be misleading because we are not really endorsing a flow as much as saying that multiple levels of `approx` are equivalent to a single level.

The EnerCaml approximation layer also provides primitives that allow developers to pass arbitrary approximation routines to the approximation layer. These approximation routines specify how the approximable operations should be approximated, as described previously in Section 2.2. We also list these primitives in Table 1. They all take an approximation routine of the appropriate type (e.g., `int -> int` for the integer approximation routines) and return `unit`. If no approximation functions are specified with these primitives, the EnerCaml system defaults to using random bit-flips for all array loads and integer arithmetic, and to calculating within an error margin for floating point math. We provide additional primitives to set the probability of bit flips and floating point errors (as well as the size of the floating point errors) for the default approximation routines.

Finally, the EnerCaml approximation layer allows users to set the energy scoring function by modifying the mutable reference `EnerCaml.score_callback`. As described in Section 2.2, the scoring function should process a list of approximate and precise events and return a score between 0.0 and 1.0 proportional to the amount of energy saved.

2.4 Implementation of Approximation in EnerCaml

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

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

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

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

The final piece of the approximation layer is the implementation of the approximation primitives. The `precise` primitive simply passes its argument to the callback routines that are provided as part of the OCaml-C interface. For the `approximate` primitive, we create new approximate versions of these callback routines that follow the approxi-

mate code pointer rather than the precise code pointer. The `endorse` primitive does not require a C implementation. The approximate type is implemented as an abstract type (`type 'a approx = 'a`) in the `EnerCaml` module, so `endorse` is simply the identity function:

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

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

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

3 Profiling Layer

The profiling layer is responsible for estimating the energy savings and quality of service 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 *QoS function*). The profiling layer runs the code twice, collects the outputs of both runs, and compares them using the QoS function.

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

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

and compare them using peak signal-to-noise ratio (PSNR):

```
let rec se_sum prec_l app_l =
  match prec_l, app_l with
    prc_hd::prc_tl, app_hd::app_tl ->
      (app_hd -. prc_hd) *.
      (app_hd -. prc_hd) +.
      (se_sum prc_tl app_tl)
  | _ -> 0.
in
let mse prec app = (se_sum prec app) /.
  (float_of_int (List.length prec)) in
let psnr precL appL = 10. *. (log10
  ((255. *. 255.)/.(mse precL appL))) in
EnerCaml.eval_qos 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 QoS function (the argument to `eval_qos`) to the two lists. We then output the computed QoS and an estimate of the energy saved. As mentioned in Section 2.2, we provide hooks that let users customize the estimation of energy savings from approximation.[2] By default, we use a simple metric proportional to the percentage of approximable operations executed approximately (i.e., the computed score is the percentage of the recorded events whose approximate flag is set to `true`).

3.1 Implementation of Profiling in EnerCaml

The code changes required to implement the EnerCaml profiler were once again straightforward. We modified the interpreter to run the code multiple times. In standard profiling mode, there are only two runs: one fully precise run and one run with all user-specified approximation enabled. Section 4 describes our autotuning layer, which adds additional runs. During the fully precise run, the interpreter simply follows the precise code pointer at every function application bytecode (including the `_approx` application bytecodes). The only other change required for profiling EnerCaml codes was to implement the `EnerCaml.eval_qos` primitive. For this to work we had to ensure that the list of output data from the precise run did not get moved or deleted by the garbage collector once we started subsequent approximate runs. To accomplish this, we take advantage of the fact that `eval_qos` is called after all of the data is produced. From the user's perspective, `eval_qos` does nothing during a precise run. However, behind the scenes, we modified it to copy the precise output data list from the OCaml heap to the C heap. We store a pointer to the copied list. On subsequent approximate runs, `eval_qos` computes the quality of service by applying the passed-in QOS function to this previously-stored precise output list and the approximate output list collected during the current approximate run.

4 Autotuning Layer

The profiling layer lets programmers investigate the QoS and efficiency implications of their approximate programs. However, to improve QoS–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 QoS 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–QoS 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 QoS. The configurations

[2] The profiler formats the energy score as a percentage (i.e., the score returned by the customized metric is multiplied by 100, so for example 0.948 will be output as as 94.8).

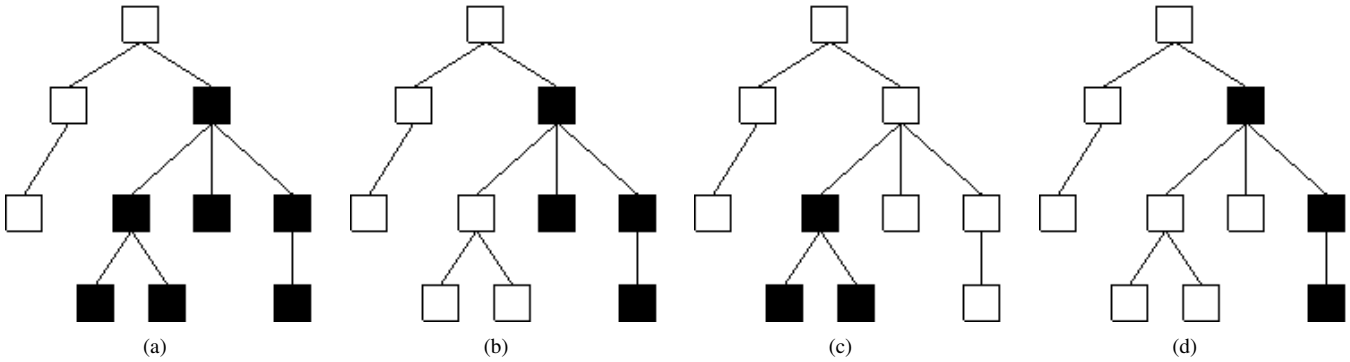


Figure 1: Static call trees illustrating the various strategies we use to search the precise-approximate decompositions of EnerCaml programs for improved quality of service versus efficiency tradeoffs. A black node represents an approximate function application and a white node represents a precise application.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

with the best efficiency and QoS 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 QoS–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:

- Treat a single call site as if it were surrounded by a call to the `precise` primitive (illustrated in Figure 1b). Note that this also causes any calls under it to execute precisely (unless another call to `approximate` is encountered).

- Make all call sites in the computation precise except for one. Note that this includes the call to the code passed to `approximate`, thus this effectively "narrows" approximation to the chosen site (Figure 1c).

- 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 (Figure 1d). For example, the two may pass data from one to the other or both may pass data to a third function.

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 QoS and estimated energy savings. If one result has both better QoS 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 QoS 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.

4.1 Implementation of Autotuning in EnerCaml

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

We also changed the interpreter to implement the autotuner's search strategies. In autotuning mode, our interpreter

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

5 Case Studies

We used the EnerCaml system to profile and tune the approximation properties of three existing OCaml applications, none of which were written by us. This section discusses our experiences with those applications. First, Section 5.1 describes profiling the ray tracer application mentioned previously. Next, Section 5.2 discusses our experiences with profiling an N-body simulation application. Finally, Section 5.3 discusses a collision detection kernel.

5.1 Ray Tracer

Our initial experience with the EnerCaml system involved adding approximation to a ray tracer (downloaded from the website of Flying Frog Consultancy [9]). The ray tracer has two phases: scene creation and ray tracing. The scene creation phase creates a scene consisting of a number of spheres of different sizes. The ray tracing phase then generates an image by sending a series of rays at the scene.

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

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

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

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

We used the default EnerCaml approximation routines, with an error rate set to 0.5%[3] (i.e., one out of every 200 approximable operations returns an incorrect result). Figure 3a shows an image generated by this approximation of the the ray tracer.

We next instrumented the program for profiling and autotuning so that we could search for ways to improve the quality of the initial image (Figure 3a). Recall that profiling in EnerCaml involves specifying a quality of service evaluation function and adding calls to collect the data required for the evaluation. We chose peak signal-to-noise ratio (PSNR) for our quality of service. We pass the quality of service function to the `eval_qos` routine, which in turn passes it lists of data from precise and approximate runs:

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

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

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

After instrumenting the code, we ran it through the simple profiler to determine the quality of service and efficiency of our initial attempt at approximation. Our initial PSNR was 26.9, with 94.8% approximation. We next ran our autotuner to see if we could improve on these results. Figure 2 shows a plot of the best results (i.e., the quality of service/efficiency frontier curve), as well as the textual output for a selection of these results. The first thing that jumps out of these results is that we can obtain better quality (PSNR of 28.4), while only giving up a very small amount of approximation by making the scene creation precise. Intuitively, small changes in the positions of spheres can have significant impacts on the errors of some pixels because they can move the boundary between shadowed (dark) and non-shadowed (bright) pixels. These errors may not be as noticeable to a human viewer as the random errors generated by approximating rays, but they have a significant impact on our chosen metric, PSNR. These types of errors would also be more noticeable to humans in a video setting, where small shifts in the positions of objects could create inter-frame jitter. Since approximating scene creation also had a negligible impact on efficiency (most of the energy is spent on tracing the scene, not creating it), we removed it and reran the autotuner.

On our second autotuning run, the most interesting results consisted of a PSNR of 29.9 with 86.3% of approximable operations approximated, and of a PSNR of 36.9 with 22.3% of approximable operations approximated (Figure 3b). The 29.9 PSNR result was obtained by narrowing the approximation to just the call to the `ray_sphere` function (which

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

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

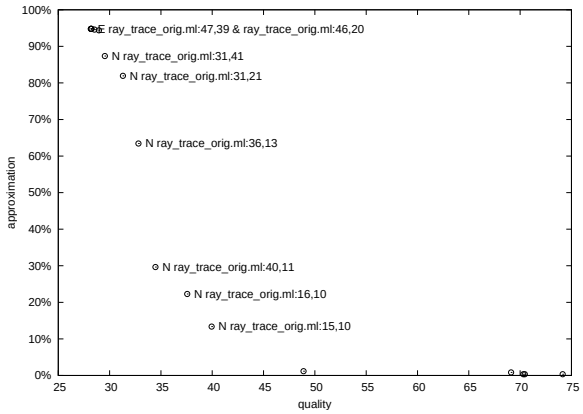


Figure 2: Our autotuner produces a textual and graphical depiction of the best results among the profiled executions.

computes the first intersection of a ray and a sphere). The 36.9 PSNR result was obtained by narrowing the approximation to a particular dot product computation inside of the `ray_sphere` function. These results led us to focus our approximation efforts on the ray-sphere intersection code. We moved the approximation primitive in the ray tracer code to just the `ray_sphere` call site, and reran the profiler. This gave us a number of new interesting points along our frontier curve, including a PSNR of 33.6 with 41.8% approximation, a PSNR of 32.9 with 50.7% approximation, and a PSNR of 31.5 with 64.1% approximation. All three of these results were obtained by making either individual calculations or pairs of calculations inside `ray_sphere` precise. Our favorite result, with PSNR 33.6 and 41.8% approximation, is shown in Figure 3c. It has slightly lower quality than the PSNR 36.9 result in Figure 3b, but nearly twice the number of approximated operations.

It would have required significantly more effort to characterize the effects of approximation without the assistance of our profiling and autotuning tool. The autotuner allowed us to quickly remove scene creation from consideration due to its poor tradeoff between quality and efficiency. It then pointed us to the importance of the `ray_sphere` function and allowed us to focus our efforts there.

5.2 N-Body Simulation

The next application that we looked at was an N-body simulation (downloaded from the Computer Language Benchmarks Game website [20]). We started by adding a simple quality of service metric that calculates the inverse of the average error:

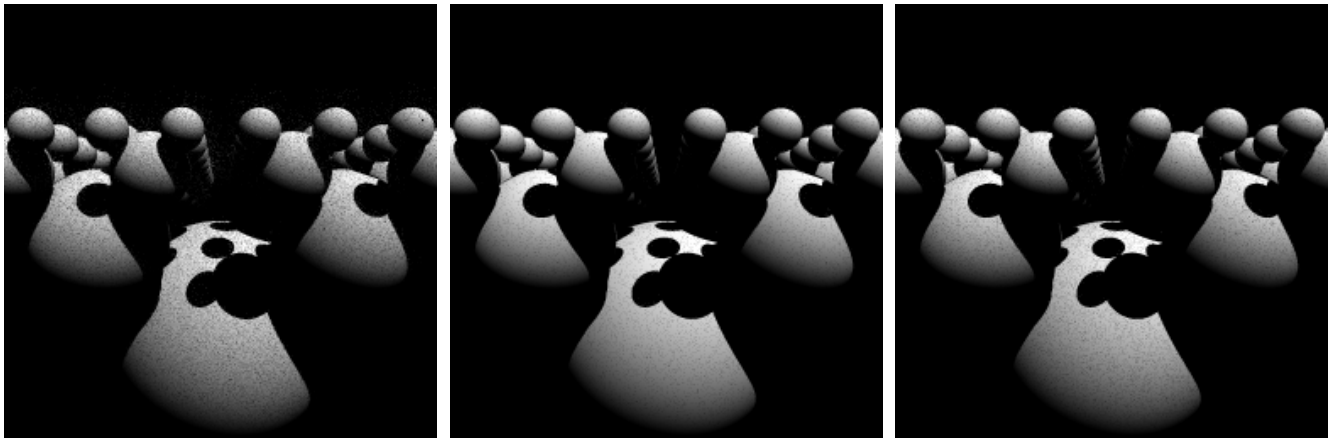
```
let inv_err pl al =
  (* Calculate inverse of average difference
     between elements of pl and al lists *)
  ...
in
EnerCaml.eval_qos inv_err
```

The simulation first initializes the N-body system with a call to `offset_momentum` and then calls `advance` in a loop to advance the state of the simulation one step at a time. We wrapped both calls in approximate thunks:

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

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

Initially, our autotuner was not able to tell us very much because the only function applications it identified were the two outer-level calls to `offset_momentum` and `advance` that we had wrapped in approximate thunks. We looked at the code and discovered that the simulation code was written in a very imperative style, whereas EnerCaml's autotuner is designed for the functional style more commonly used in OCaml applications. A doubly-nested loop over the bodies calculates the effect of each body on every other body. 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 service:



(a) (b) (c)

Figure 3: The images generated by our ray tracer with various mixtures of approximate and precise execution. Image **a** (PSNR 26.9) represents the result of approximating the entire ray tracing and scene creation computations. Image **b** (PSNR 36.9) limits the approximation to a single dot product inside the `ray_sphere` function. Image **c** (PSNR 33.6) approximates the `ray_sphere` function, but executes two of its dot products precisely. It has slightly lower quality than image **c**, but almost twice as much approximation.

```
QOS = 5562.919330
Approximation score: 24.456460
...
QOS = 7436.822960
Approximation score: 24.456460
```

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

```
QOS = 3821.292285
Approximation score: 48.912917
```

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

5.3 Collision Detector

Our final example is a simple collision detection kernel (downloaded from [17]) that checks whether or not two triangles in 3D space intersect each other. Our quality of service metric calculates the percentage of the intersection tests where we correctly detect whether or not the triangles intersect. Based on our experiences with the previous examples, we did not attempt to approximate the initialization. Instead, we just surrounded the call to the intersection routine with an approximate thunk. We then passed the result of the intersection routine to a function that parses the result and records it for the profiler:

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

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

```
Percent correct = 97.810000
Approximation score: 93.960868
```

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

```
if ((sign da1) = (sign da2) &&
    (sign da2) = (sign da3)) then
  NoIntersection
```

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

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

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

We experimented with making these computations precise. When we made both of the plane normal calculations precise, our quality of service increased to 98.88% correct, but our approximation percentage dropped a bit, to 67.1%. When we instead made the no-intersection checks precise, our quality of service did not increase by as much, only rising to 97.94%. However, our approximation was almost unchanged at 93.0%. When we combined both changes we were able detect 98.93% of collisions correctly and still approximate 66.1% of the approximable operations. Compared to the original annotation, we were able to eliminate over 51% of the errors while losing less than 30% of the approximation. This whole process took under an hour for one author unfamiliar with the code.

6 Related Work

Many systems have proposed trading off quality to improve performance or save energy using both software [1, 19, 22, 10] and hardware [4, 13, 8, 11, 16, 3] techniques. Several studies have shown that a wide variety of applications can tolerate the resulting imprecision with acceptable results [12, 5, 21]. 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. [2] propose a proof system for verifying user-specified correctness properties in relaxed programs. Misailovic et al. [14] use probabilistic reasoning to prove accuracy bounds on relaxed transformations. EnerJ [18] 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.

Quality-of-service profiling [15] 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 QoS 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 QoS–efficiency tradeoffs that are inherent in approximate computing. This work addresses that pressing need.

References

[1] BAEK, W., AND CHILIMBI, T. M. Green: a framework for supporting energy-conscious programming using controlled approximation. In *PLDI* (2010).

[2] CARBIN, M., KIM, D., MISAILOVIC, S., AND RINARD, M. C. Reasoning about relaxed programs. In *PLDI* (June 2012).

[3] CHAKRAPANI, L. N., AKGUL, B. E. S., CHEEMALAVAGU, S., KORKMAZ, P., PALEM, K. V., AND SESHASAYEE, B. Ultra-efficient (embedded) SOC architectures based on probabilistic CMOS (PCMOS) technology. In *DATE* (2006).

[4] DE KRUIJF, M., NOMURA, S., AND SANKARALINGAM, K. Relax: an architectural framework for software recovery of hardware faults. In *ISCA* (2010).

[5] DE KRUIJF, M., AND SANKARALINGAM, K. Exploring the synergy of emerging workloads and silicon reliability trends. In *Silicon Errors in Logic—System Effects* (2009).

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

[7] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., AND BURGER, D. Dark silicon and the end of multicore scaling. In *ISCA* (2011).

[8] ESMAEILZADEH, H., SAMPSON, A., CEZE, L., AND BURGER, D. Architecture support for disciplined approximate programming. In *ASPLOS* (2012).

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

[10] HOFFMANN, H., SIDIROGLOU, S., CARBIN, M., MISAILOVIC, S., AGARWAL, A., AND RINARD, M. Dynamic knobs for responsive power-aware computing. In *ASPLOS* (2011).

[11] LEEM, L., CHO, H., BAU, J., JACOBSON, Q. A., AND MITRA, S. ERSA: Error resilient system architecture for probabilistic applications. In *DATE* (2010).

[12] LI, X., AND YEUNG, D. Exploiting soft computing for increased fault tolerance. In *Workshop on Architectural Support for Gigascale Integration* (2006).

[13] LIU, S., PATTABIRAMAN, K., MOSCIBRODA, T., AND ZORN, B. G. Flikker: Saving refresh-power in mobile devices through critical data partitioning. In *ASPLOS* (2011).

[14] MISAILOVIC, S., ROY, D. M., AND RINARD, M. C. Probabilistically accurate program transformations. In *SAS* (2011).

[15] MISAILOVIC, S., SIDIROGLOU, S., HOFFMAN, H., AND RINARD, M. Quality of service profiling. In *ICSE* (2010).

[16] NARAYANAN, S., SARTORI, J., KUMAR, R., AND JONES, D. L. Scalable stochastic processors. In *DATE* (2010).

[17] OTI, E. Collision detection: triangle-triangle intersection. `http://www.elliottoti.com/index.php?p=28`.

[18] SAMPSON, A., DIETL, W., FORTUNA, E., GNANAPRAGASAM, D., CEZE, L., AND GROSSMAN, D. EnerJ: Approximate data types for safe and general low-power computation. In *PLDI* (2011).

[19] SIDIROGLOU, S., MISAILOVIC, S., HOFFMAN, H., AND RINARD, M. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE* (2011).

[20] TROESTLER, C. n-body OCaml program: Computer Language Benchmarks Game. `http://shootout.alioth.debian.org/u32/program.php?test=nbody&lang=ocaml%&id=1`, Jan. 2012.

[21] WONG, V., AND HOROWITZ, M. Soft error resilience of probabilistic inference applications. In *Silicon Errors in Logic—System Effects* (2006).

[22] ZHU, Z. A., MISAILOVIC, S., KELNER, J. A., AND RINARD, M. Randomized accuracy-aware program transformations for efficient approximate computations. In *POPL* (2012).