Profiling and Autotuning for Energy-Aware Approximate Programming

Michael Ringenburg, Adrian Sampson, Luis Ceze, and Dan Grossman

Motivation

- Approximation has well-known benefits
  - Energy saving, performance, etc.
  - Thus this workshop
- But, as a developer, *how do we write an approximate application?*
  - How do we understand/manage tradeoffs between energy/performance and quality/precision?
- Key to adoption: easy-to-use, configurable tools that assist developers

sampa

This talk: Prototyping tools

- Development often starts with prototyping
- What should an approximation prototyper look like?
  - What tools are needed?
- We propose a three layered system
  - **Approximation layer:** Provide *simple, coarse-grained* approximate semantics and simulation.
  - **Profiling layer:** Determine quality (QoR) impacts, and energy/performance benefits
    - Allow customization of approximate semantics, benefits
  - **Autotuning layer:** Suggest refinements to approximation that may improve tradeoffs

EnerCaml

- EnerCaml: our implementation of this design
- Built on top of OCaml
  - An ML variant with object-oriented extensions
  - Often used for prototyping
  - Functional style great for coarse-grained approximation
- Contains the three layers described earlier
  - Code-centric approximation via primitive call
  - Profiling with customizable quality metrics
  - Autotuning by searching for alternate precise-approximate decompositions

Approximation Layer

- Key primitive for *code-centric* approximation
  - `EnerCaml.approximate : (unit->’a)->’a approx`
    - Takes a (thunked) code block (think C++ functor), executes it approximately, and returns an approximately-typed result.
    - Also provide endorsement, precise, continue primitives.
- Convenient model for prototyping – just specify approximate kernels
- Natural fit for a functional language
  - Everything is a function
- Simulation: simply create precise and approximate versions of each function
  - Approximate versions execute approximate operations
  - Call sites in approximate functions call approximate versions

Ray Tracer Approximation Example

```
(* Compute a pixel by sending rays in
   every direction *)
for dx = 0 to ss - 1 do
  for dy = 0 to ss - 1 do
    (* Compute direction vector *)
    ...
    (* Trace ray *)
    let next_ray = ray_trace dir scene in


    g := !g +. next_ray;
  done;
done;
```

Ray Tracer Approximation Example

```
(* Compute a pixel by sending rays in
   every direction *)
for dx = 0 to ss - 1 do
  for dy = 0 to ss - 1 do
    (* Compute direction vector *)
    ...
    (* Trace ray approximately *)
    let next_ray = EnerCaml.approximate (
      fun () -> ray_trace dir scene)
    in
    g := !g +. EnerCaml.endorse(next_ray);
  done;
done;
```

sampa

Next layer: Profiling

- Profiling layer lets users investigate the effects of approximation on their code
- Two key features:
  - Measure the quality of result and efficiency impacts of approximation.
  - Let users customize (defaults provided):
    - how operations are approximated (via custom error functions)
    - relative energy savings of approximate operations (via custom scoring function)

Measuring QoR impacts

- Profiling layer lets users define a quality function that compares data from precise and approximate executions.
- User also specifies data to collect to use as input to the QoR function.
    - Stored as a temporally ordered list.
- Profiler executes the application precisely and then approximately, and compares the data lists collected in the two executions using the QoR function.

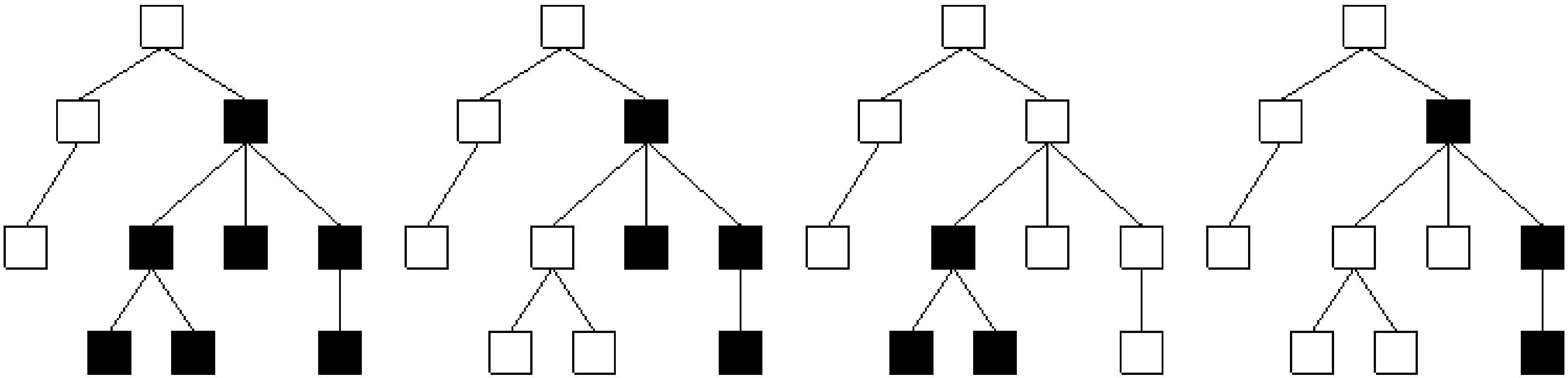
Example: Ray Tracer Profiling

```
(* loop over pixels *)
for (...)
   (* compute brightness g of current pixel *)
   ...
   (* add g to list of profile output for
      current execution *)
  EnerCaml.record_profile_output g;
done;
let psnr prec_lst app_list =
   (* compute PSNR of pixels in app_list
      relative to pixels in prec_list *)
   ...
in EnerCaml.eval_qor psnr
```

EnerCaml Autotuning Layer

- Searches for alternate precise/approximate decompositions of programs that improve the quality and/or energy efficiency.
- Starts with the original approximation specified by the programmer.
  - Idea: specify coarsely, let autotuner refine
- Performs additional runs that remove part of the approximation.
  - Varies which function call sites call the precise versus the approximate versions of functions
- Never *add* approximation – may be unsafe

Autotuning Search Strategies



- Can't try every possible combination: exponential
- So, use some heuristics:
  - Remove approximation at a single static call site
  - Narrow approximation to a single static call site
  - Remove approximation from two "sibling" static call sites (call sites in the same calling function).

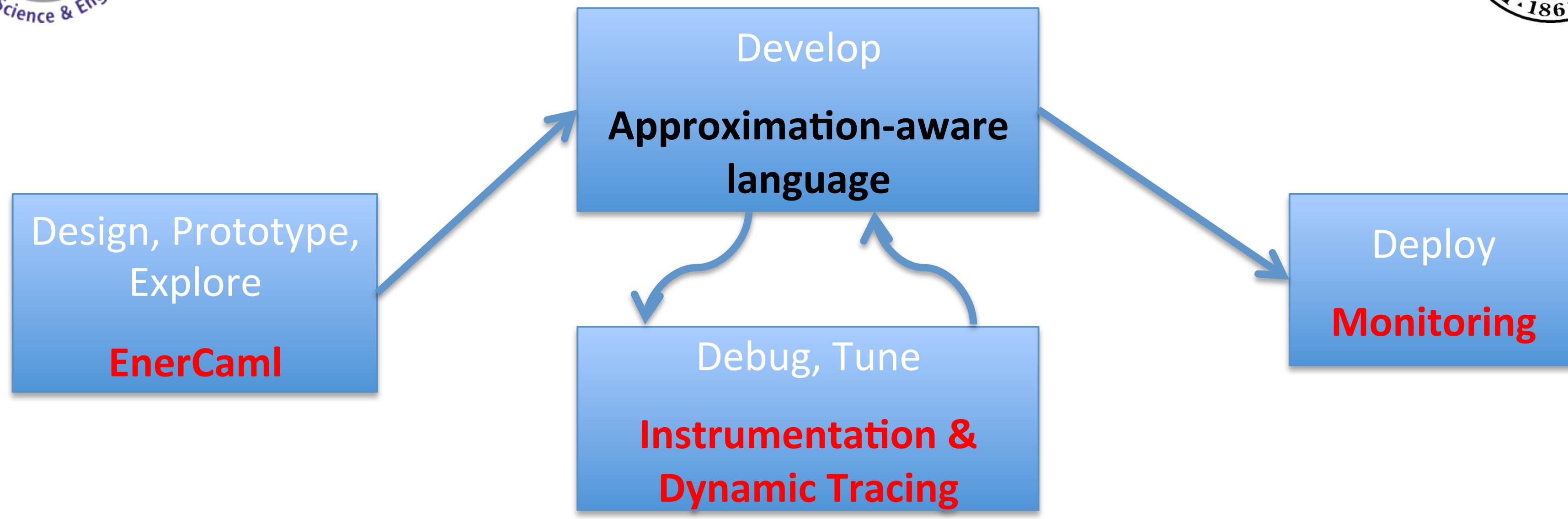
Autotuning Output

- Autotuner outputs the QoR and approximate operation counts for every trial.
- A trial *dominates* another trial if it has better QoR and more approximated operations.
- Non-dominated trials form quality-efficiency Pareto curve.
  - We output these trials with the code changes that produce them.
  - And plot these results.

Case Studies

- Ray tracer:
  - Improved PSNR from 26.9 to 33.6, while maintaining nearly half of energy savings
- N-body simulation:
  - Improved QoR (average error^{-1}) from 0.01 to nearly 4000, and maintained over half of the energy savings.
- Collision detection:
  - Reduced errors by 51% at expense of 30% approximation reduction.

Part of a Larger Ecosystem

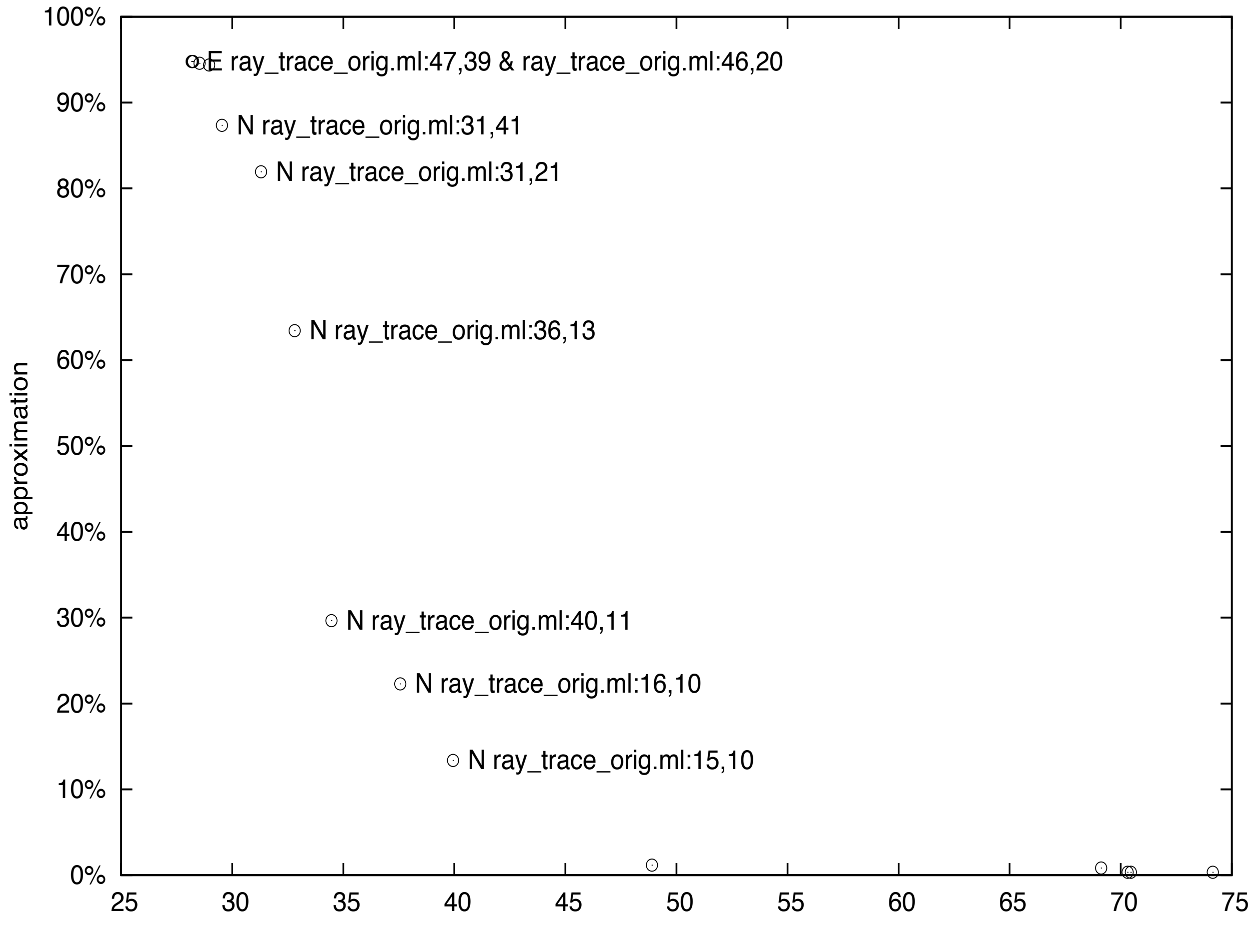


- Part of suite of dynamic tools for managing QoR of approximate applications – see my thesis!
- Aimed at different phases of the software lifecycle:
  - **EnerCaml** for design and prototyping
  - **Instrumentation & Tracing** for debugging and tuning
  - **Monitoring** for real-time, post-deployment response to QoR issues

Questions?

Backup

Autotuning Example



Tracking Approximation

- To track our two function versions, the compiler creates *dual function closures*
  - Closures typically used to represent functions in languages where they are first-class values. Contain pointers to a function and an environment.
  - Our dual closures replace the single function pointer in the closure with two: one for a precise version, and one for the approximate version.
  - Call the approximate version of function passed to approximate primitive call (and precise version in precise primitive)
  - All other calls are determined statically by context
    - If we are in a precise caller, calls go to the precise callee.
    - If we are in an approximate caller, call the approximate callee.

Specifying Approximation

- EnerCaml's approximable operations:
  - Integer arithmetic
  - Floating point arithmetic
  - Integer and floating point array loads
- Approximation function for each of these: replaces result of the operation with another (possibly identical) result of the same type.
  - E.g., introduce a bit flip 0.1% of the time.
  - **`set_float_approximation : (float->float) -> unit`**
  - **`set_integer_approximation : (int->int) -> unit`**
  - **`set_load_approximation : (int->int) -> unit`**
  - **`set_load_float_approximation : (float->float) -> unit`**
- Also, log approximate and precise operations, and let users create a customized *energy score*.
  - Default scorer is just percentage approximated.