ACCEPT Framework

ACCEPT: A Programmer-Guided Compiler Framework for Practical Approximate Computing

Adrian Sampson André Baixo Benjamin Ransford Thierry Moreau
Joshua Yip Luis Ceze Mark Oskin

Group 10: Zhixiang Teoh, Peter Ly, Owen Goebel, Neel Shah

Motivating analogy

Sometimes, we might want to **sacrifice an image's quality to reduce its file size,** provided the image is still sufficiently high quality.

Similarly, we might want to **sacrifice a program's accuracy to boost its speed,** provided the program is still sufficiently accurate.

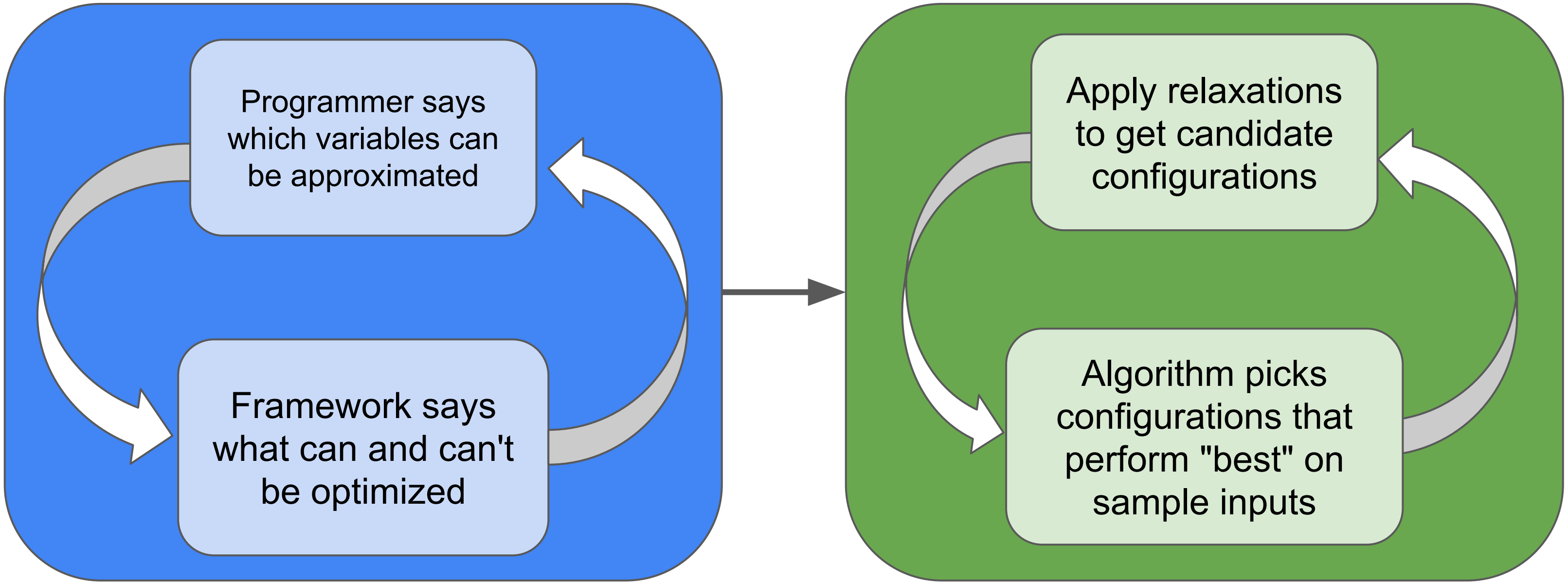
ACCEPT approximates at the compilation step.

Low-compression (high quality) JPEG

High-compression (low quality) JPEG

(image source)

ACCEPT programming model



Programmer says which variables can be approximated

Framework says what can and can't be optimized

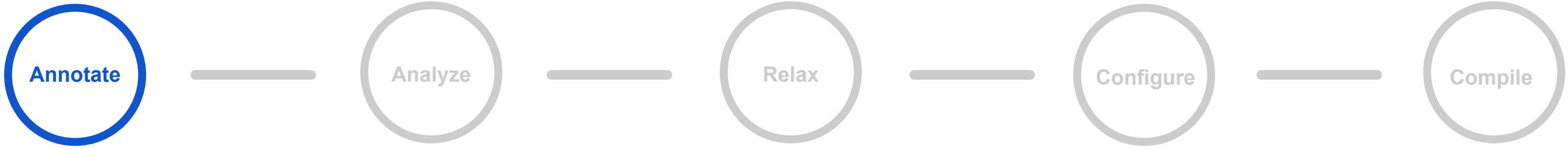
Apply relaxations to get candidate configurations

Algorithm picks configurations that perform "best" on sample inputs

"Static" (programming) Phase

"Dynamic" (testing) Phase

ACCEPT workflow



Programmer **annotates** C++ code with `APPROX` and `ENDORSE` keywords

System does **precise-purity analysis** and identifies **relaxation opportunity sites** (i.e., approximate region selection)

System identifies code regions and **safe approximation relaxations**—e.g., loop perforation, synchronization elision

System vets individual relaxations, and conducts **autotuning search** to produce good **composite configurations**

System **applies** program relaxations to generate candidate binaries

APPROX and ENDORSE

The value of this variable is safe to approximate

Variables are "precise" by default

```
APPROX int a = func();

funcPrecise(a); // illegal!
```

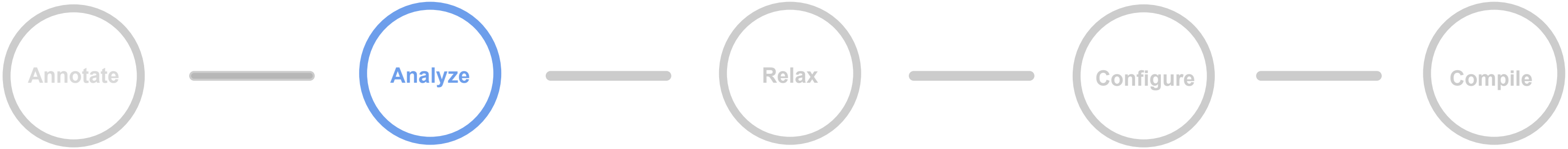
funcPrecise expects a precise value, but is supplied an APPROX value

Pointers can never be APPROX

```
funcPrecise(ENDORSE(a)); // legal!
```

"casts" from APPROX to precise

ACCEPT workflow



Programmer **annotates** C++ code with `APPROX` and `ENDORSE` keywords

System does **approx-only analysis** and identifies **relaxation opportunity sites** (i.e., approximate region selection)

System identifies code regions and **safe approximation relaxations**—e.g., loop perforation, synchronization elision

System vets individual relaxations, and conducts **autotuning search** to produce good **composite configurations**

System **applies** program relaxations to generate candidate binaries

~~Precise-purity~~ Approx-only

An approx-only region:

1. Does **not overwrite precise variables** that may be read outside the region
2. Only **calls functions that are entirely approx-only**
3. Does **not have unbalanced synchronization** (e.g. Lock() with no Unlock())

Approx-only is the **key criterion for whether program relaxations can apply**

ACCEPT determines whether a region of interest is **approx-only** to decide whether to **apply program relaxations**

Approx-only examples

```
int sensitive = 583;
APPROX int c = 483;
sensitive = c;
```

Approx-to-precise flow unsound

```
APPROX double cost = 0;
int *weights = ...;
```

```
for (int i = 0; i < N; i++) {
    cost += (weights[i] - 
weights[0]);
}
```

```
weights[N+1] = cost;
```

Stores to precise variable `weights[N+1]` that may be read outside block

Approx-only analysis

Initially assume not approx-only,
then attempt to prove approx-only.

Checks approx-only conditions

1. No stores to outside precise variables
2. Only calls functions that are entirely approx-only
3. No unbalanced synchronization

Done **conservatively** via LLVM passes:

```
isPure[func] = false;
```

- **SSA definition-use chains**
- **Pointer escape analysis**

Approximate region selection

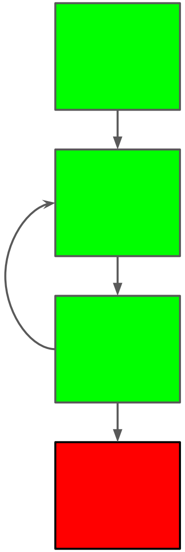
ACCEPT builds larger approx-only blocks with a single entrance and exit point

Why: To identify chunks of code that are amenable to approximation!

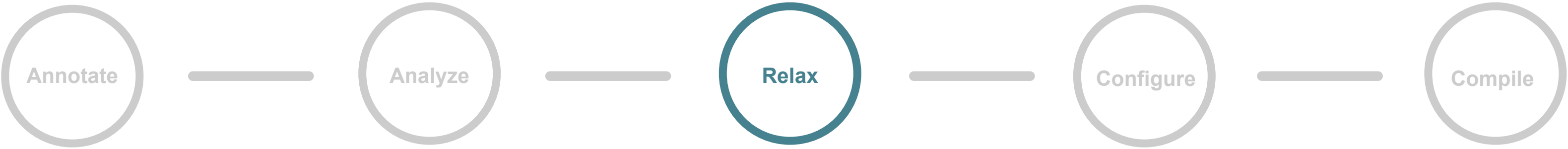
```
APPROX double cost = 0;
int *weights = ...;
```

```
for (int i = 0; i < N; i++) {
    cost += (weights[i] - weights[0]);
}
```

```
weights[N+1] = cost;
```



ACCEPT workflow



Annotate — Analyze — Relax — Configure — Compile

Programmer **annotates** C++ code with `APPROX` and `ENDORSE` keywords

System does **precise-purity analysis** and identifies **relaxation opportunity sites** (i.e., approximate region selection)

System identifies code regions and **safe approximation relaxations**—e.g., loop perforation, synchronization elision

System vets individual relaxations, and conducts **autotuning search** to produce good **composite configurations**

System **applies** program relaxations to generate candidate binaries

ACCEPT implements several relaxations

Only allowed in **approx-only regions!**

Loop perforation: Skip iterations to speed up loops

```
for (int i = 0; i < N; i++)
{
    // approx-only body
}
```

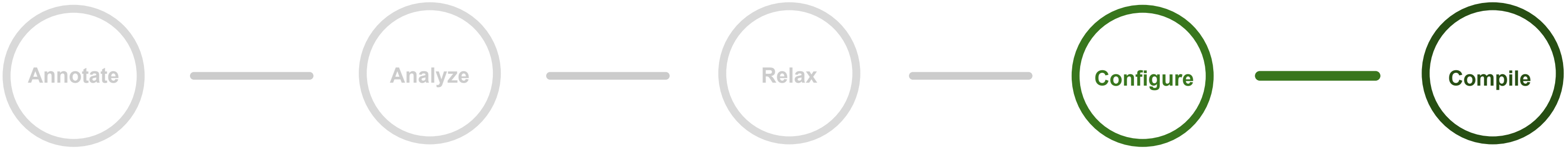
Loop perforation

```
for (int i = 0; i < N; i+=k)
{
    // approx-only body
}
```

Synchronization elision: Remove thread synchronization

Neural acceleration: Replace complex code with a trained neural network approximation

ACCEPT workflow



Annotate — Analyze — Relax — Configure — Compile

Programmer **annotates** C++ code with `APPROX` and `ENDORSE` keywords

System does **precise-purity analysis** and identifies **relaxation opportunity sites** (i.e., approximate region selection)

System identifies code regions and **safe approximation relaxations**—e.g., loop perforation, synchronization elision

System vets individual relaxations, and conducts **autotuning search** to produce good **composite configurations**

System **applies** program relaxations to generate candidate binaries

Autotuning search for configurations

Goal: Find optimal combinations of relaxations (balancing quality and speed)

Programmer supplies quality metric

How?

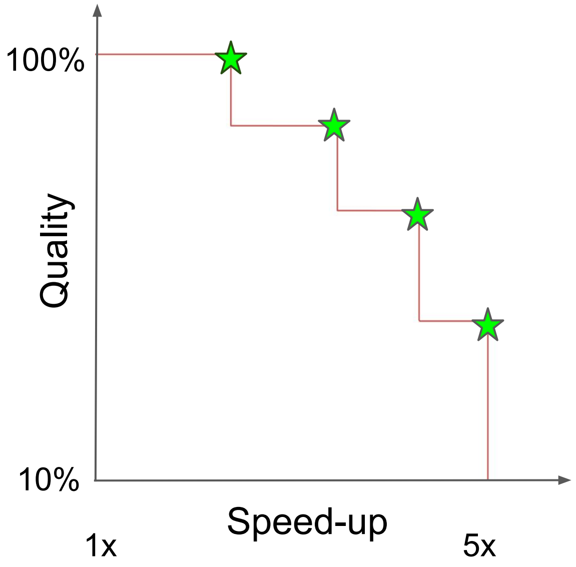
1.) Find good relaxations on test inputs

2.) Compose “good” relaxations

- a.) Uses a knapsack model to compose relaxations
  Maximize speed-up without exceeding error threshold

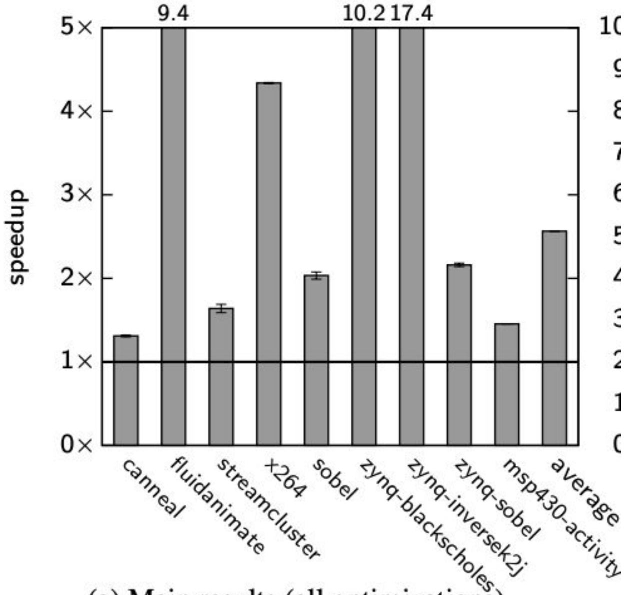
3.) Return optimal relaxations

Programmer chooses best relaxation composition



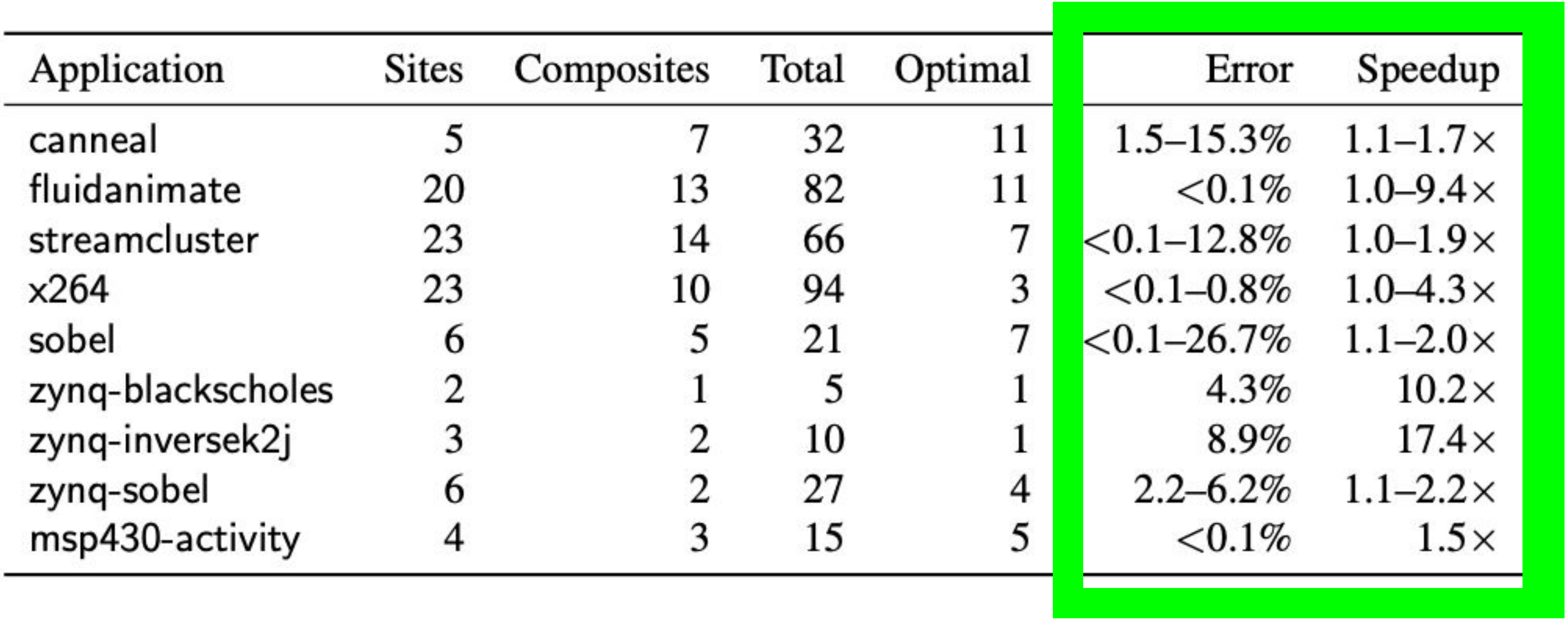
Results: Average 2.5x speedup with under 15%* error

Benchmarks include video encoding, financial algorithms, and simulation algorithms (and more) with manually added APPROX and ENDORSE. Multiple runs are averaged



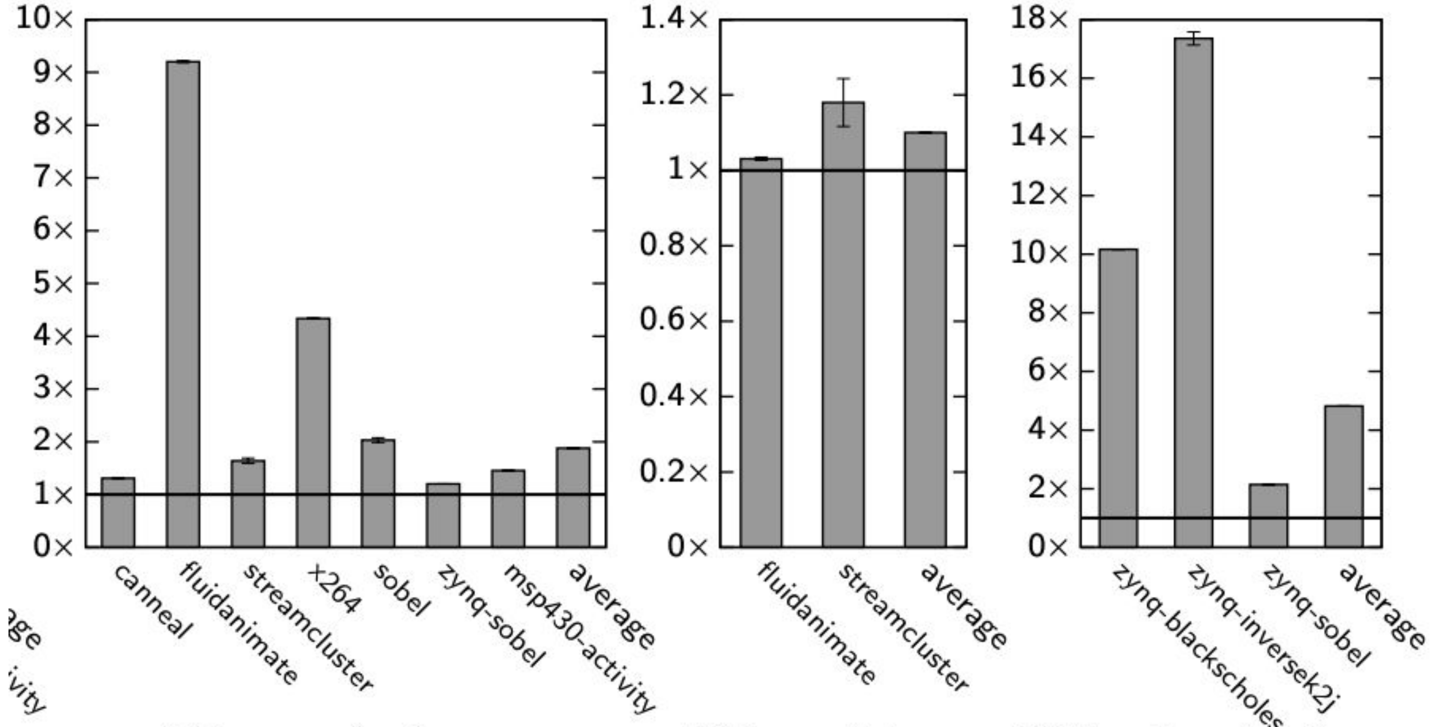
(a) Main results (all optimizations)

Average 2.5x speedup with under 15%* error (continued)



Application	Sites	Composites	Total	Optimal	Error	Speedup
canneal	5	7	32	11	1.5–15.3%	1.1–1.7×
fluidanimate	20	13	82	11	<0.1%	1.0–9.4×
streamcluster	23	14	66	7	<0.1–12.8%	1.0–1.9×
x264	23	10	94	3	<0.1–0.8%	1.0–4.3×
sobel	6	5	21	7	<0.1–26.7%	1.1–2.0×
zynq-blackscholes	2	1	5	1	4.3%	10.2×
zynq-inversek2j	3	2	10	1	8.9%	17.4×
zynq-sobel	6	2	27	4	2.2–6.2%	1.1–2.2×
msp430-activity	4	3	15	5	<0.1%	1.5×

Some benchmarks are more robust to certain techniques



(b) Loop perforation (c) Sync. elision (d) Neural acceleration

Future improvements

Measuring power consumption vs. accuracy

Introducing more relaxations to the framework (for example, width reduction)

Improve on the autotuning algorithm

- REACT: A Framework for Rapid Exploration of Approximate Computing Techniques