CS 4110

Approximate Computing
free lunch

exponential single-threaded performance scaling!

(time immemorial 2005 2015) (not to scale)
SUMMARY

FIGURE S.1

Processor performance from 1986 to 2008 as measured by the benchmark suite SPECint2000 and consensus targets from the International Technology Roadmap for Semiconductors for 2009 to 2020. The vertical scale is logarithmic. A break in the growth rate at around 2004 can be seen. Before 2004, processor performance was growing by a factor of about 100 per decade; since 2004, processor performance has been growing and is forecasted to grow by a factor of only about 2 per decade. An expectation gap is apparent. In 2010, this expectation gap for single-processor performance is about a factor of 10; by 2020, it will have grown to a factor of 1,000. Most sectors of the economy and society implicitly or explicitly expect computing to deliver steady, exponentially increasing performance, but as these graphs illustrate, traditional single-processor computing systems will not match expectations. Note that the SPEC benchmarks are a set of artificial workloads intended to measure a computer system's speed. A machine that achieves a SPEC benchmark score that is 30 percent faster than that of another machine should feel about 30 percent faster than the other machine on real workloads.
Free lunch

Time

Immemorial

2005

Multicore era

2015

We'll scale the number of cores instead
The multicore transition was a stopgap, not a panacea.
free lunch  multicore era  who knows?

- time
  - immemorial
- 2005
- 2015

?
Application
Language
Architecture
Circuits

hardware–software abstraction boundary

parallelism
data movement

Architecture

guard bands
energy costs
Application

Language

Architecture

Circuits

parallelism
data
movement

hardware-software abstraction boundary

guard
bands

energy
costs
surgical robotics

ACCURACY IS ESSENTIAL

airplane autopilots

C compilers

theorem provers

kernels

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C compilers

encryption

not so much

C com
Hardware support for disciplined approximate programming

```java
int p = 5;
@Approx int a = 7;
for (int x = 0..) {
    a += func(2);
    @Approx int z;
    z = p * 2;
    p += 4;
}
```
Approximation-aware ISA

```
ld 0x04 r1
ld 0x08 r2
add r1 r2 r3
st 0x0c r3
```
Approximation-aware ISA

```
ld    0x04 r1
ld    0x08 r2
add.a r1  r2   r3
st.a  0x0c r3
```
Precise

- references
- jump targets
- JPEG header

Approximate
- pixel data
- neuron weights
- audio samples
- video frames

Safety by isolation
Type qualifiers

```c
@Approx int a = ...;
@Precise int p = ...;
```

```c
p = a;  // Incorrect
```

```c
a = p;  // Correct
```

Type qualifiers

```java
@Approx int a = ...;

@Precise int p = ...;

p = a;  // ✗

a = p;  // ✓
```
@Approx int a = expensive();

@Precise int p;

// Incorrectly assigning `a` to `p`.
P = a;
quickChecksum(p);
output(p);
Endorsement: escape hatch

```java
@Approx int a = expensive();

@Precise int p;

✓ p = endorse(a);
✓ quickChecksum(p);
output(p);
```
Logic approximation: overloading

```java
@Approx int a = ...;

@Precise int p = ...;

p + p;

p + a;

a + a;
```
Control flow: implicit flows

```java
@Approx int a = ...;

@Precise int p = ...;

if (a == 10) {
    p = 2;
}
```
Control flow: implicit flows

@Approx int a = ...;

@Precise int p = ...;

if (endorse(a == 10)) {
    p = 2;
}
class FloatSet {
    float[] nums = ...;
    float mean() {
        calculate mean
    }
}

new @Approx FloatSet()
new @Precise FloatSet()
class FloatSet {
    @Context float[][] nums = ...;
    float mean() {
        calculate mean
    }
}
class FloatSet {

    @Context float[] nums = ...;
    float mean() {
        calculate mean
    }

    @Approx float mean_APPROX() {
        take mean of first \( \frac{1}{2} \)
    }

}

@Approx FloatSet someSet = ...;
someSet.mean();
EnerJ type system

\[ P ::= \text{int} | \text{float} \]
\[ q ::= \text{precise} | \text{approx} \]
\[ T ::= q\ C | q\ P \]

\[
\text{precise } P \leq \text{approx } P
\]
We prove two properties about FEnerJ: type soundness and non-interference. The proof is by rule induction over the operational semantics; in particular, if

\[
\Gamma \vdash h, e \leadsto h', v
\]

then

\[
h' \equiv \tilde{h}' \quad v \equiv \tilde{v}
\]

This rule reflects EnerJ's lack of guarantees for approximate computations. To model computation on an execution substrate that allows approximate instructions, the following rule could be introduced:

\[
\Gamma \vdash h, e \leadsto \tilde{h}', \tilde{v}
\]
We prove two properties about FEnerJ: type soundness and non-interference.

We use special functional units that perform approximate operations. Approximate data stored in memory is distinguished from precise instructions. Precise instructions have the same guarantees as exact logical instructions. Approximate data may use functional units that are not guaranteed to be exact. Approximate data based on address; regions of physical memory are marked as approximate and, when accessed, are stored in approximate portions of memory. Approximate and precise registers are distinguished based on the register number.

Noninterference is a fundamental property of a virtual machine or compiler. It states that approximate computations do not influence precise values. Specifically, for a computation 

\[ \vdash Prg \text{ OK} \land \vdash h, r\Gamma : s\Gamma \]

\[ s\Gamma \vdash e : T \]

\[ r\Gamma \vdash h, e \sim h', v \]

\[ h \equiv \tilde{h} \land r\Gamma \equiv r\tilde{\Gamma} \]

\[ \vdash \tilde{h}, r\tilde{\Gamma} : s\Gamma \]

...and ending heap & value

\[ \begin{cases} r\tilde{\Gamma} \vdash \tilde{h}, e \rightarrow \tilde{h}', \tilde{v} \\ h' \equiv \tilde{h}' \\ v \equiv \tilde{v} \end{cases} \]

...and ending heap & value

A new precise-equivalent starting heap

Steps to a heap (store) & value