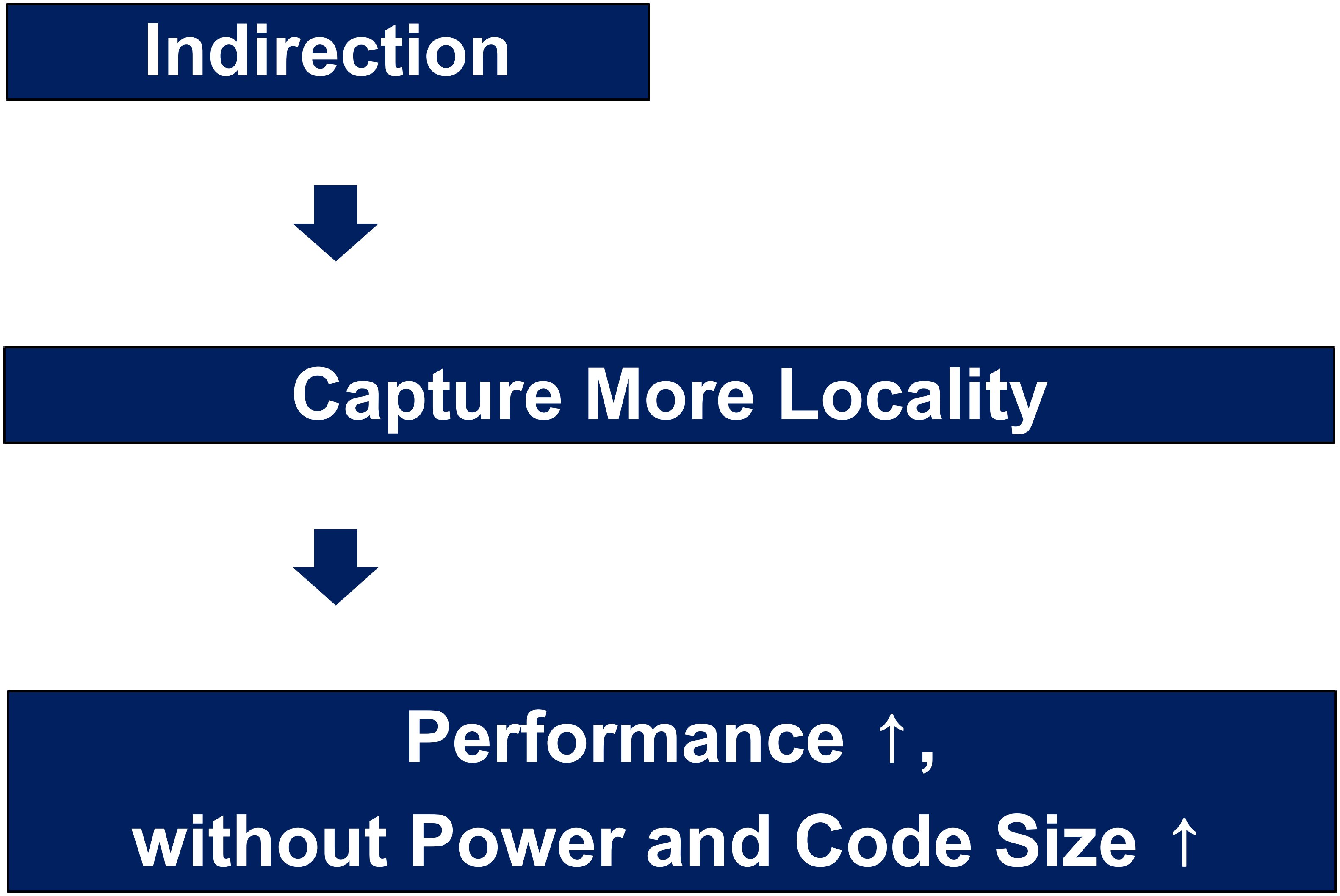
Register Pointer Architecture for Efficient Embedded Processors

JongSoo Park, Sung-Boem Park,
James Balfour, David Black-Schaffer,
Christos Kozyrakis, William Dally
Stanford University

Register Pointer Architecture (RPA)



Indirection

Capture More Locality

Performance ↑,

without Power and Code Size ↑

Embedded Computing

Real time constraint

with

Energy Efficiency

Cost Efficiency

30 frame/sec

voice

Inefficient Microprocessor

	MOPS/mW
Microprocessor	0.13
DSP	7
ASIC	200

1000x

[Broderson, ISSCC 2002]

How to close the gap?

- Efficient Embedded Computing (EEC)
  - http://cva.stanford.edu/projects/eec

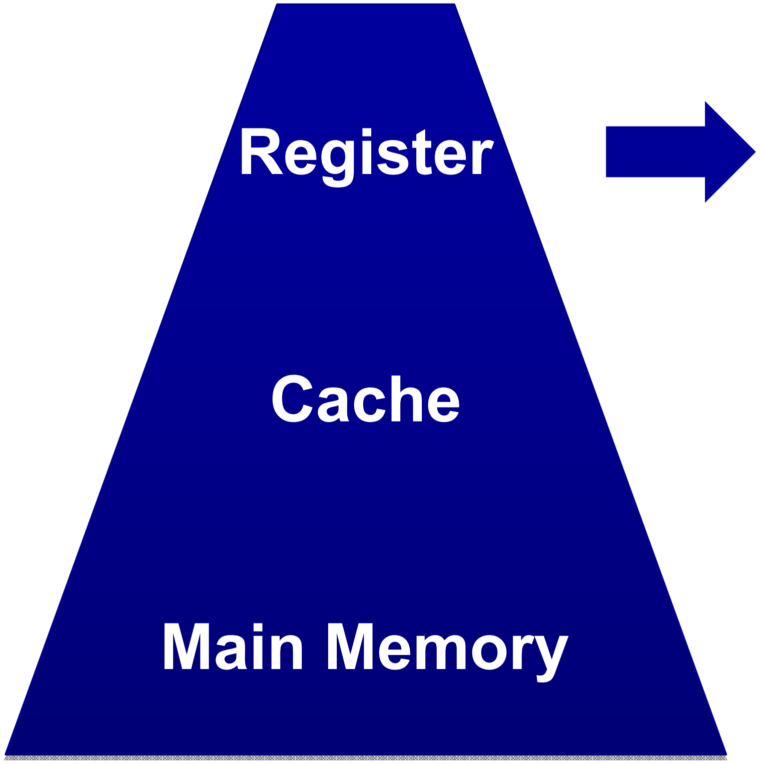
- Large portion of energy spent on data supply
  - 45% energy go to cache [Segars, ISSCC 2001]

⬇

- This work's focus:

  Energy efficient data supply

Memory Hierarchy



Fast, Close
Less Energy

Inflexible

Example: FIR (1)

```
for (i = 0; i < NUM_IN - 3; i++) {
    acc = 0;
    for (j = 0; j < 3; j++) {
        acc += coeff[j]*in[i + j];
    }
    out[j] = acc;
}
```

Unrolling

Inner-loop unrolling

```
coeff0 = coeff[0]; coeff1 = coeff[1];
coeff2 = coeff[2];
for (i = 0; i < NUM_IN - 3; i++) {
        acc = coeff0*in[i];
        acc += coeff1*in[i+1];
        acc += coeff2*in[i+2];
        out[i] = acc;
}
```

- coeff0~2: allocated in registers
- **3 loads per input**

- code size: **O(# of taps)**

Without unrolling

```
for (i = 0; i < NUM_IN - 3; i++) {
        acc = 0;
        for (j = 0; j < 3; j++) {
                acc += coeff[j]*in[i + j];
        }
        out[j] = acc;
}
```

- **6 loads per input**

Full Unrolling

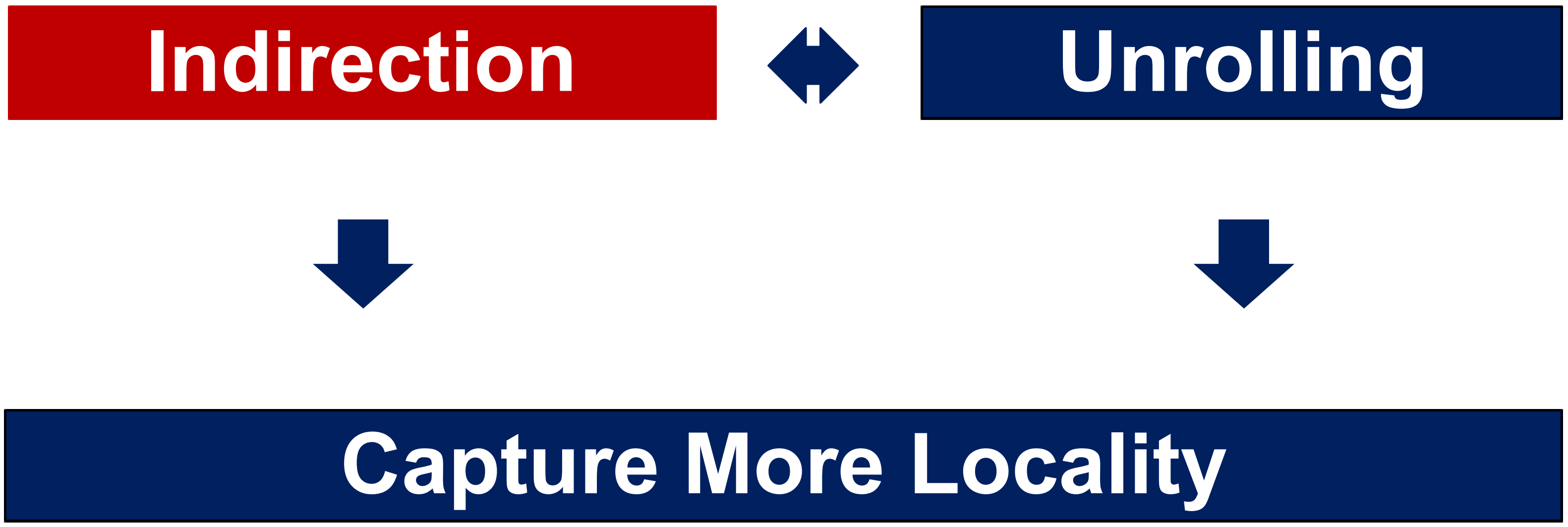
```
in0 = in[0]; in1 = in[1];
for (i = 0; i < NUM_IN - 3; i +=3 ) {
        in2 = in[i + 2];
        acc = coeff0*in0;
        acc += coeff1*in1;
        acc += coeff2*in2;
        out[i] = acc;

        in0 = in[i + 3];
        acc = coeff0*in1;
        acc += coeff1*in2;
        acc += coeff2*in0;
        out[i+1] = acc;

        in1 = in[i + 4];
        acc = coeff0*in2;
        acc += coeff1*in0;
        acc += coeff2*in1;
        out[i + 2] = acc;
}
```

- **1 load per input**
- code size: **O((# of taps)^2)**

Problems of Unrolling

- Code size
  - 35 taps FIR with ARM ISA
    - Inner loop unroll: 14 instruction → 75 instructions (5.4x)
    - Fully unroll: 14 instructions → 1229 instructions (88x)

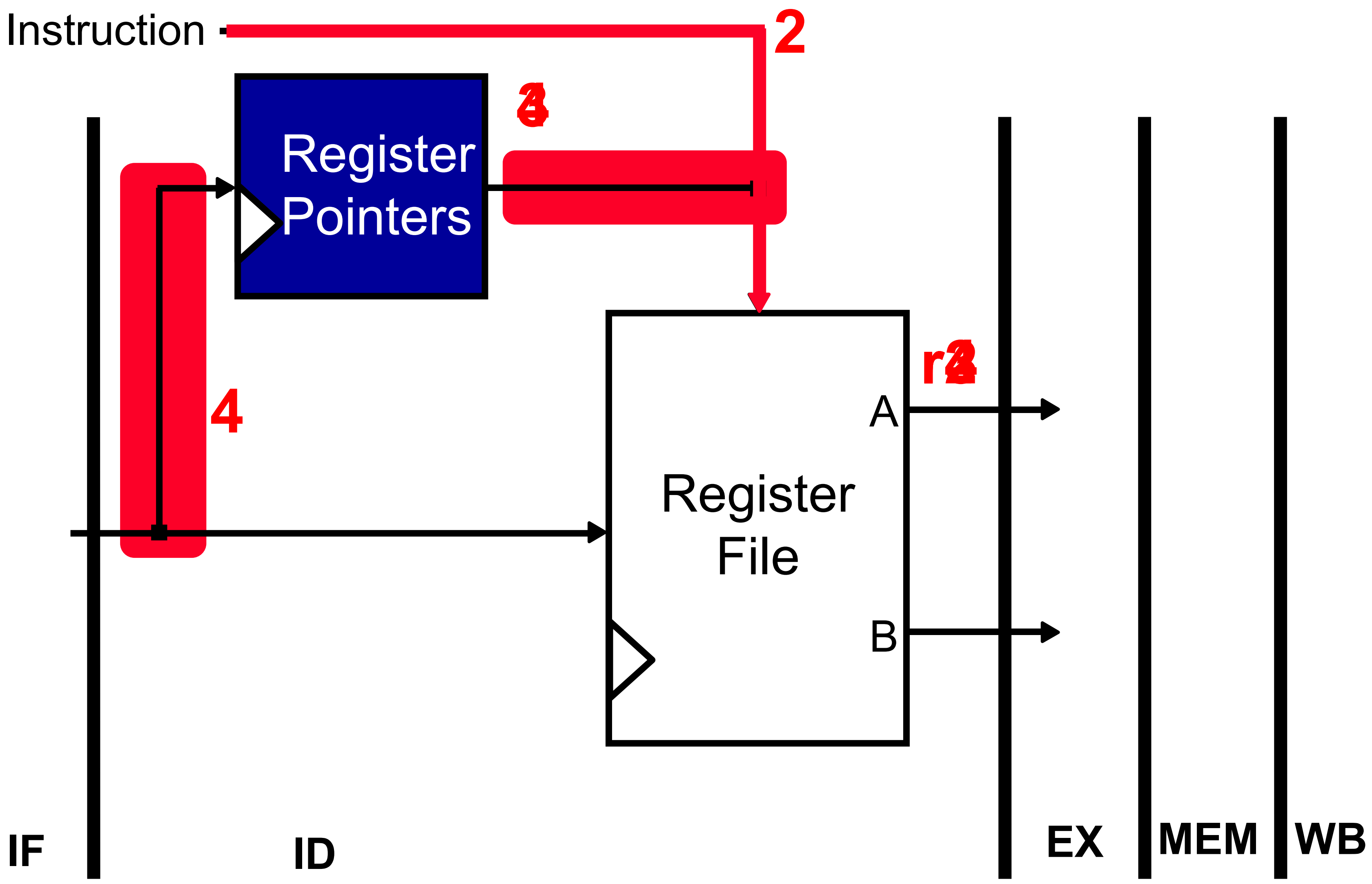
Our Approach



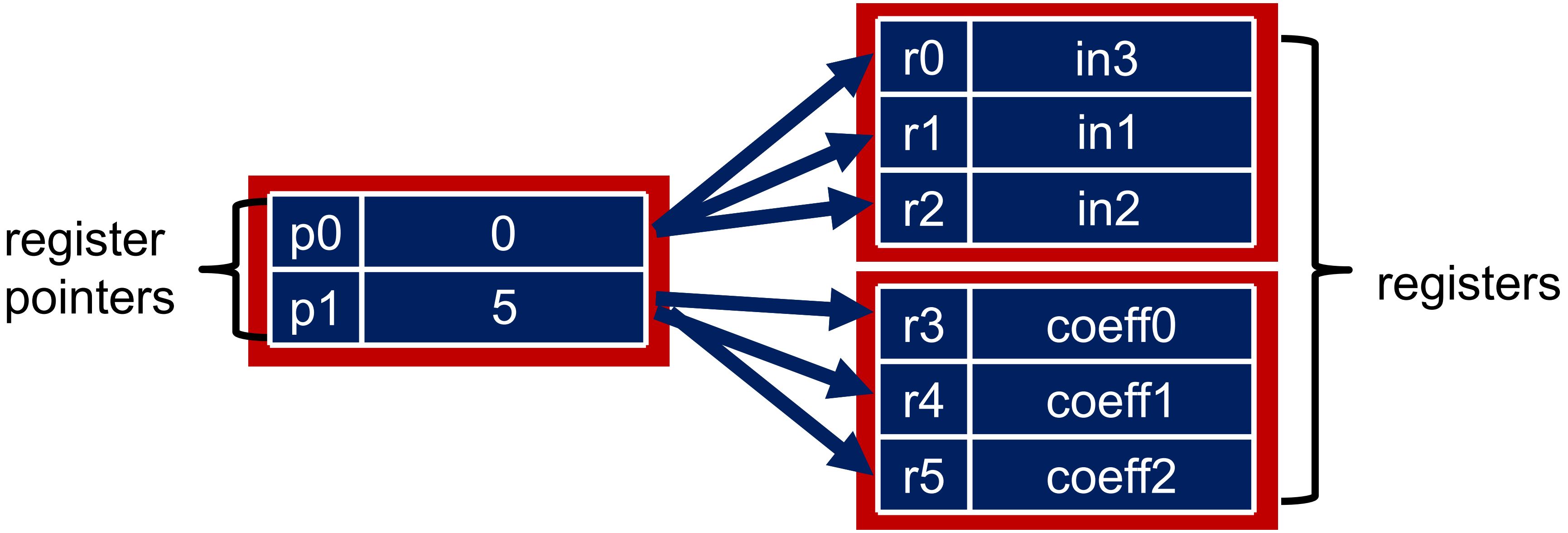
Indirection ⇔ Unrolling

Capture More Locality

Register Pointer Architecture (RPA)



FIR with RPA (2)



p0	0
p1	5

r0	in3
r1	in1
r2	in2
r3	coeff0
r4	coeff1
r5	coeff2

acc = in1*coeff0 + in2*coeff1 + in0*coeff2

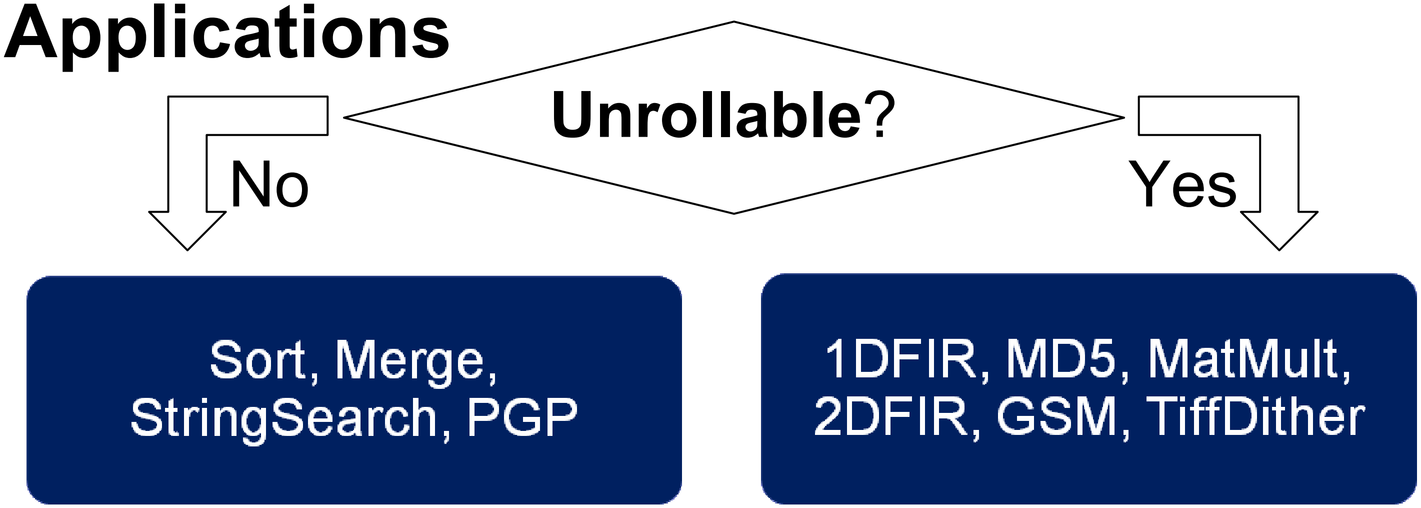
acc = in0*coeff0 + in1*coeff1 + in2*coeff2

Experiment Setup

- Configuration

Baseline 16 Regs vs **RPA** 64 Regs vs **Unrolling** 64 Regs

- Applications



Unrollable?

No

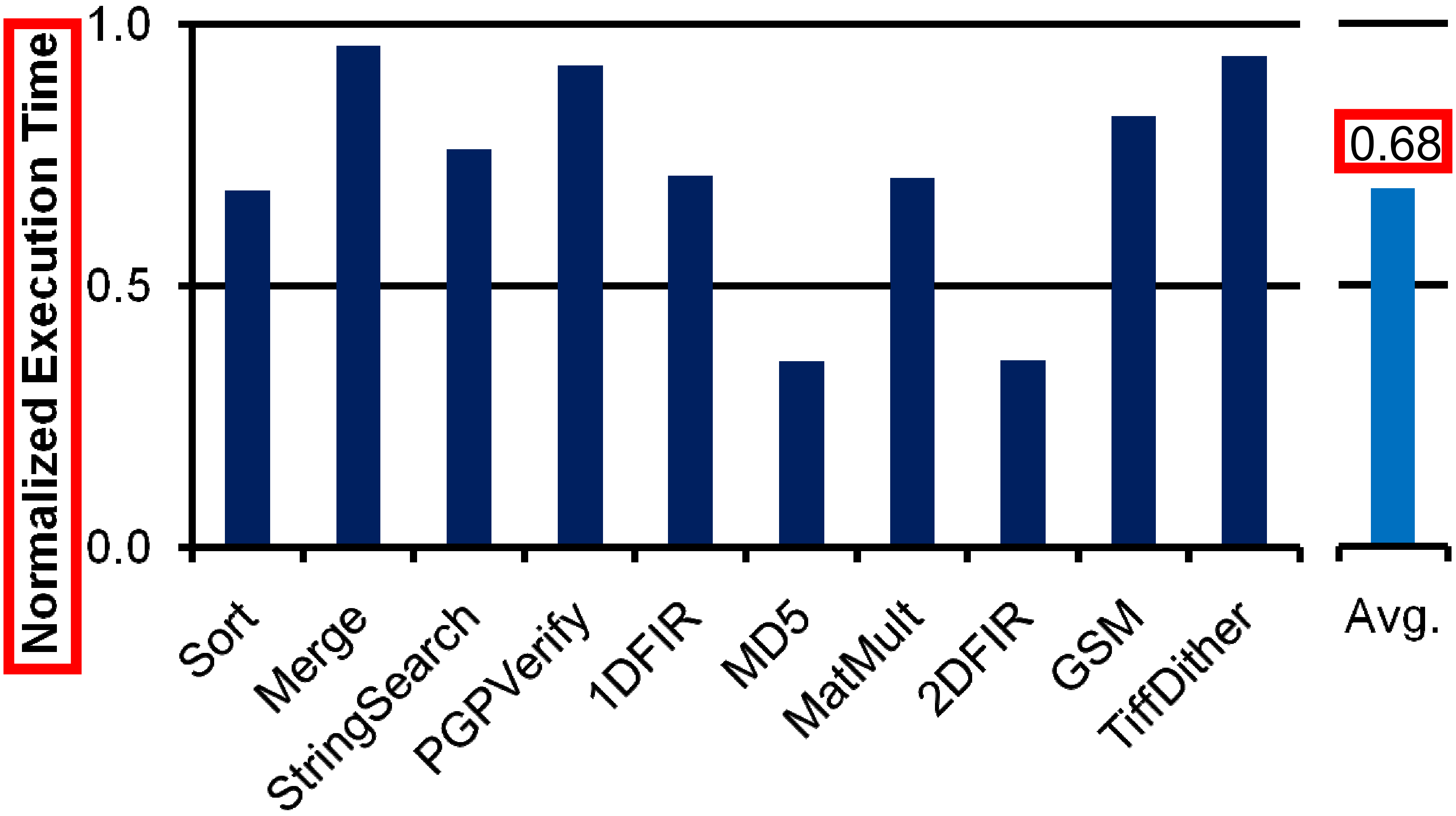
Sort, Merge, StringSearch, PGP

Yes

1DFIR, MD5, MatMult, 2DFIR, GSM, TiffDither

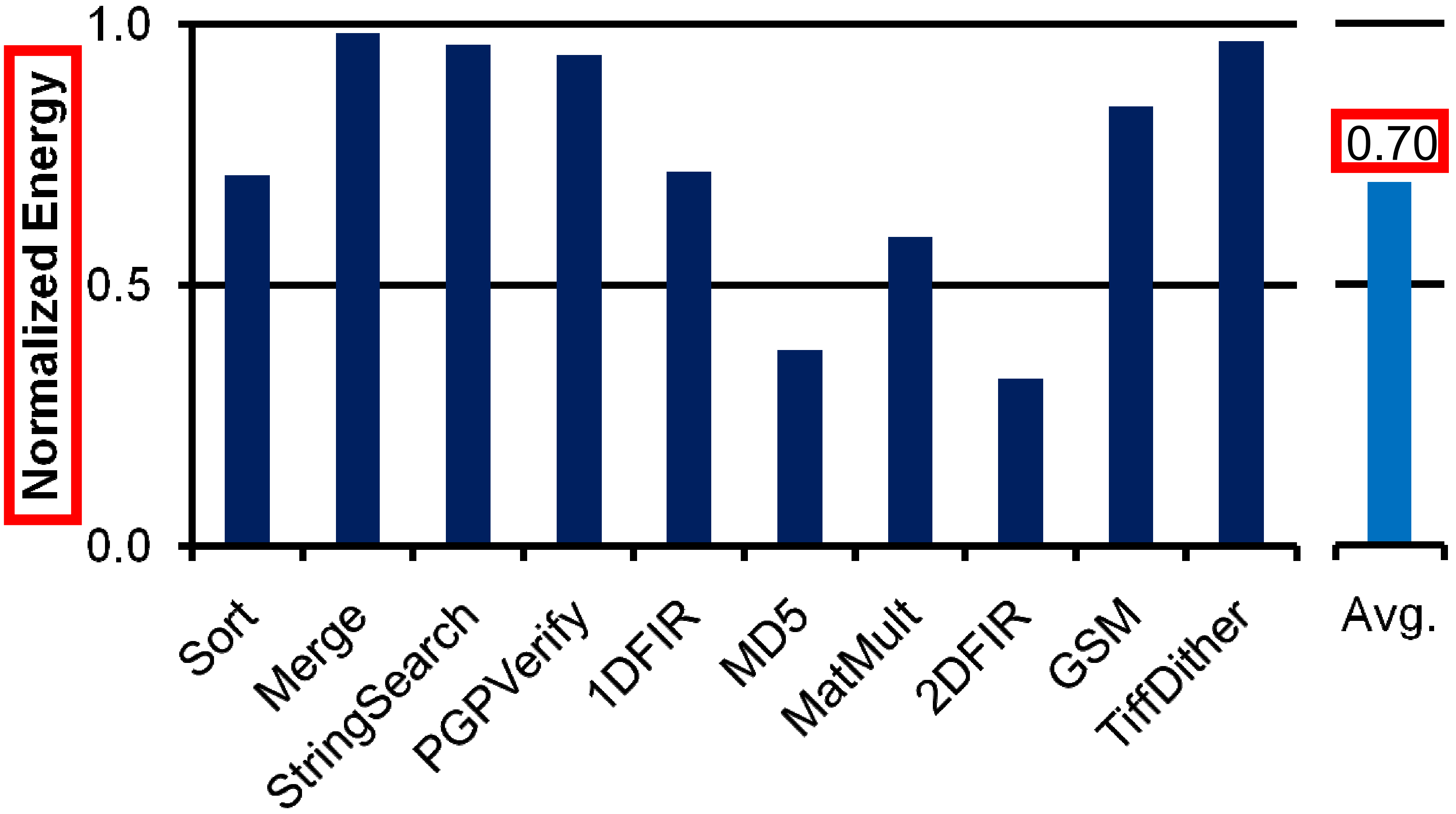
- ARM ISA, SimpleScalar, Panalyzer

Execution Time



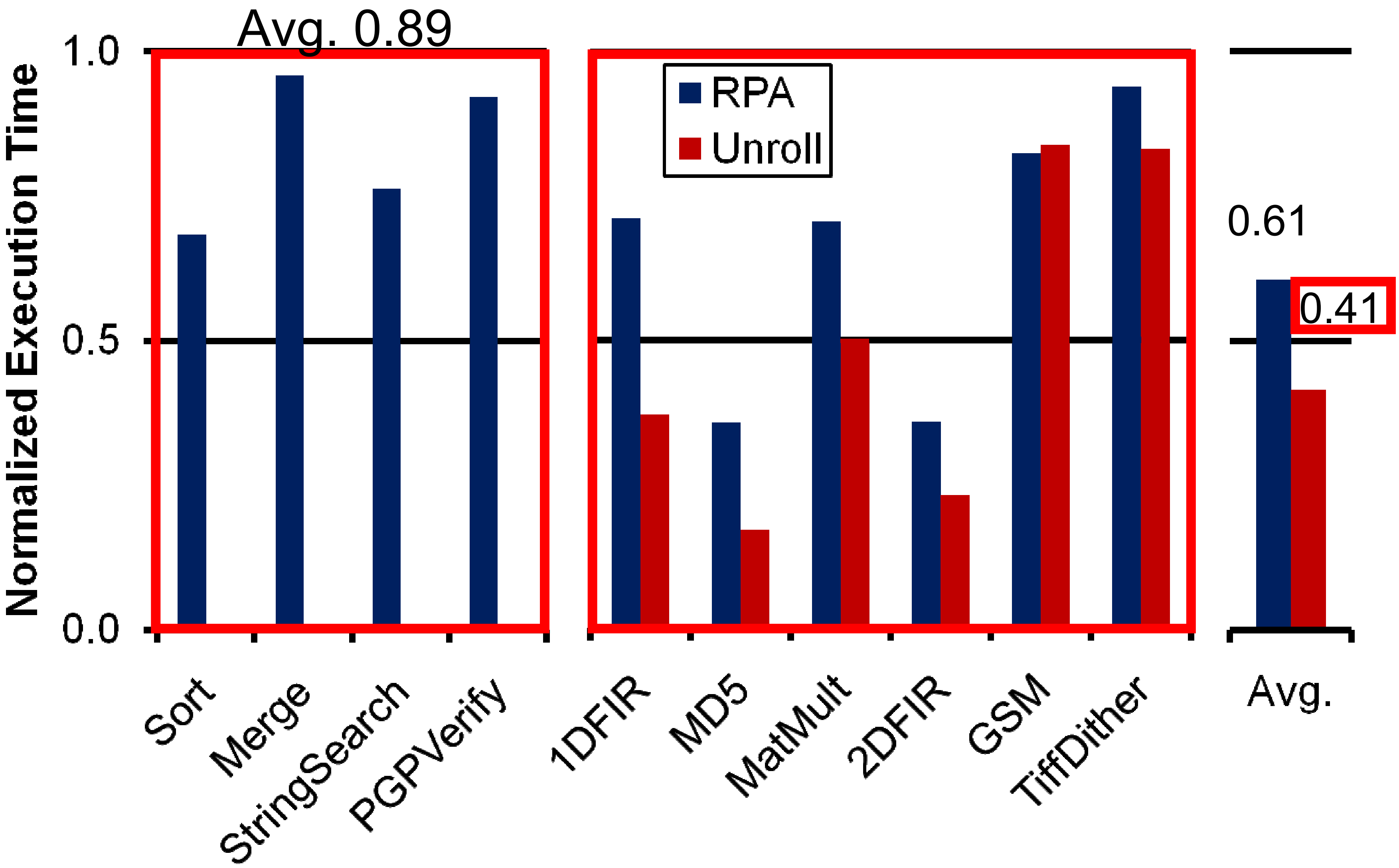
replaces memory accesses with register accesses

Energy

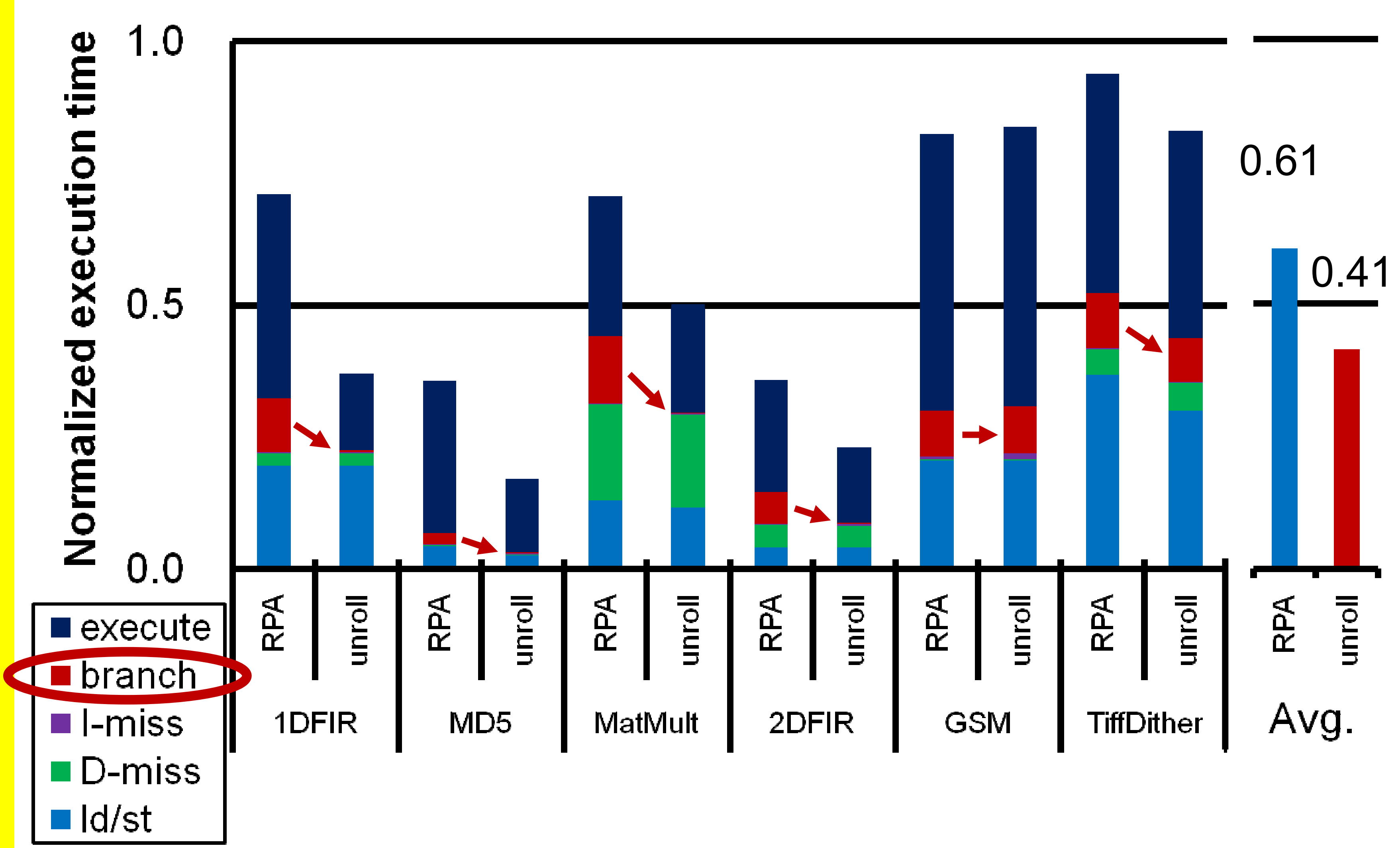


fewer cache accesses compensate larger register file's energy consumption

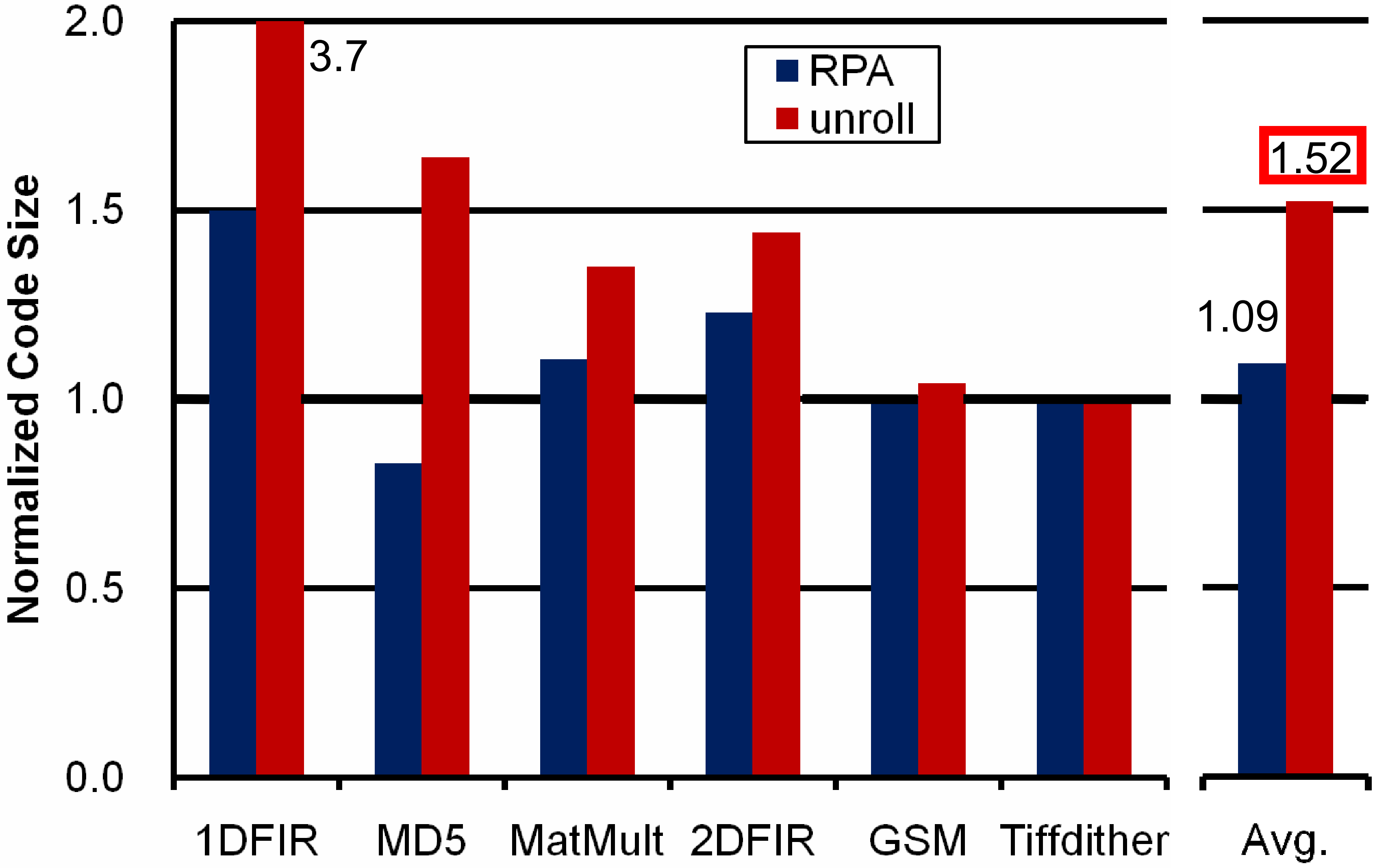
Execution Time: RPA vs. Unrolling



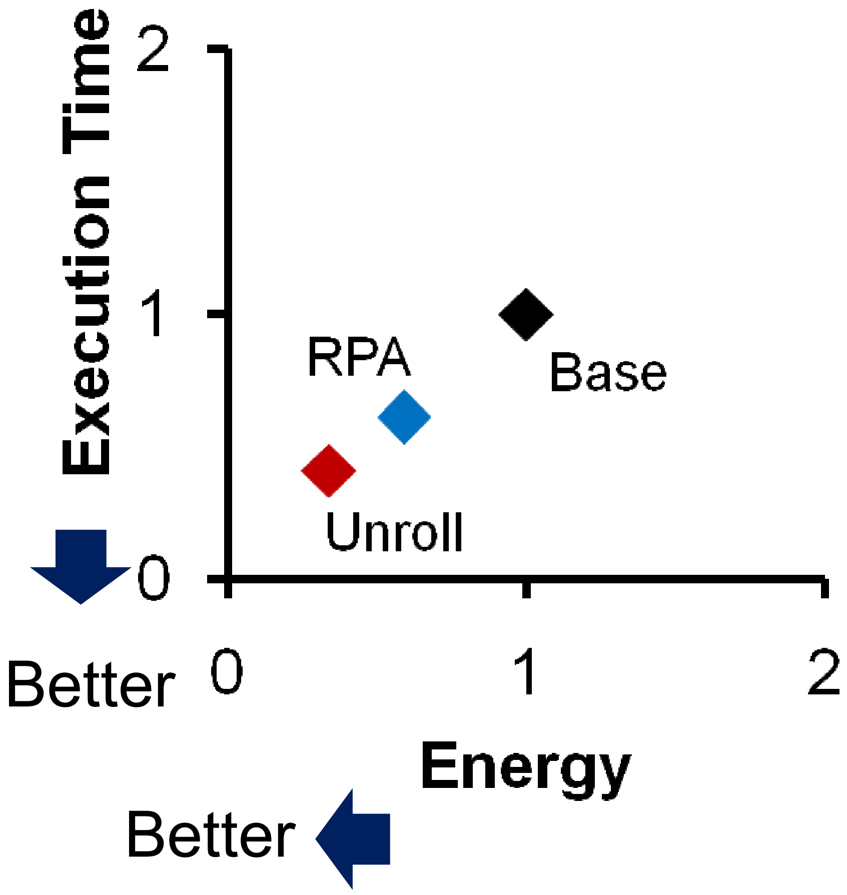
Comparison with Unrolling

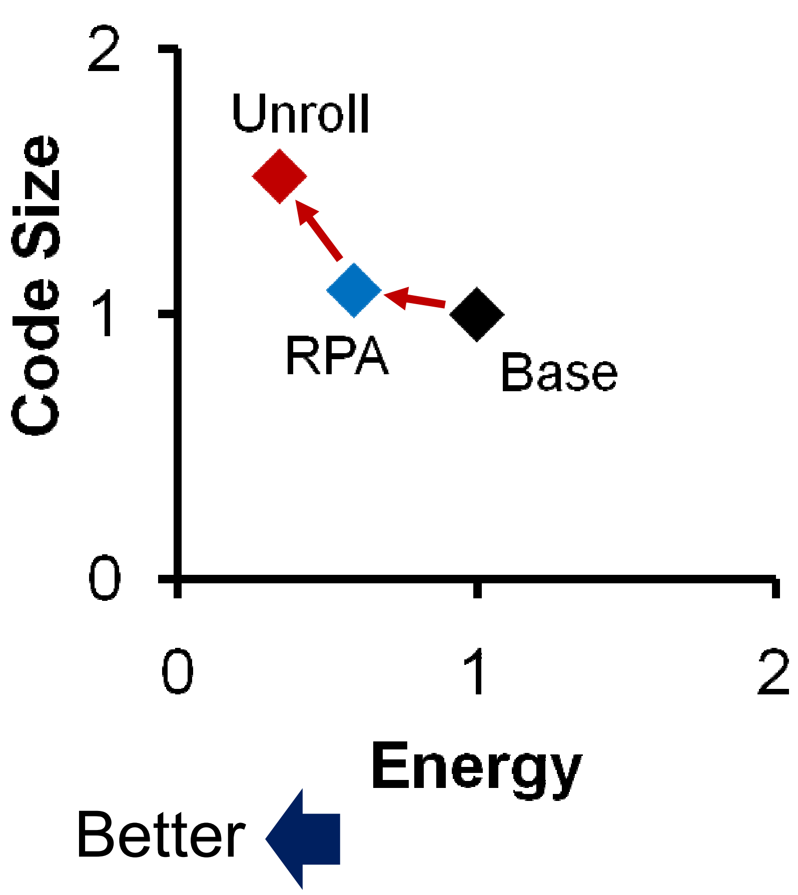


Total Code Size

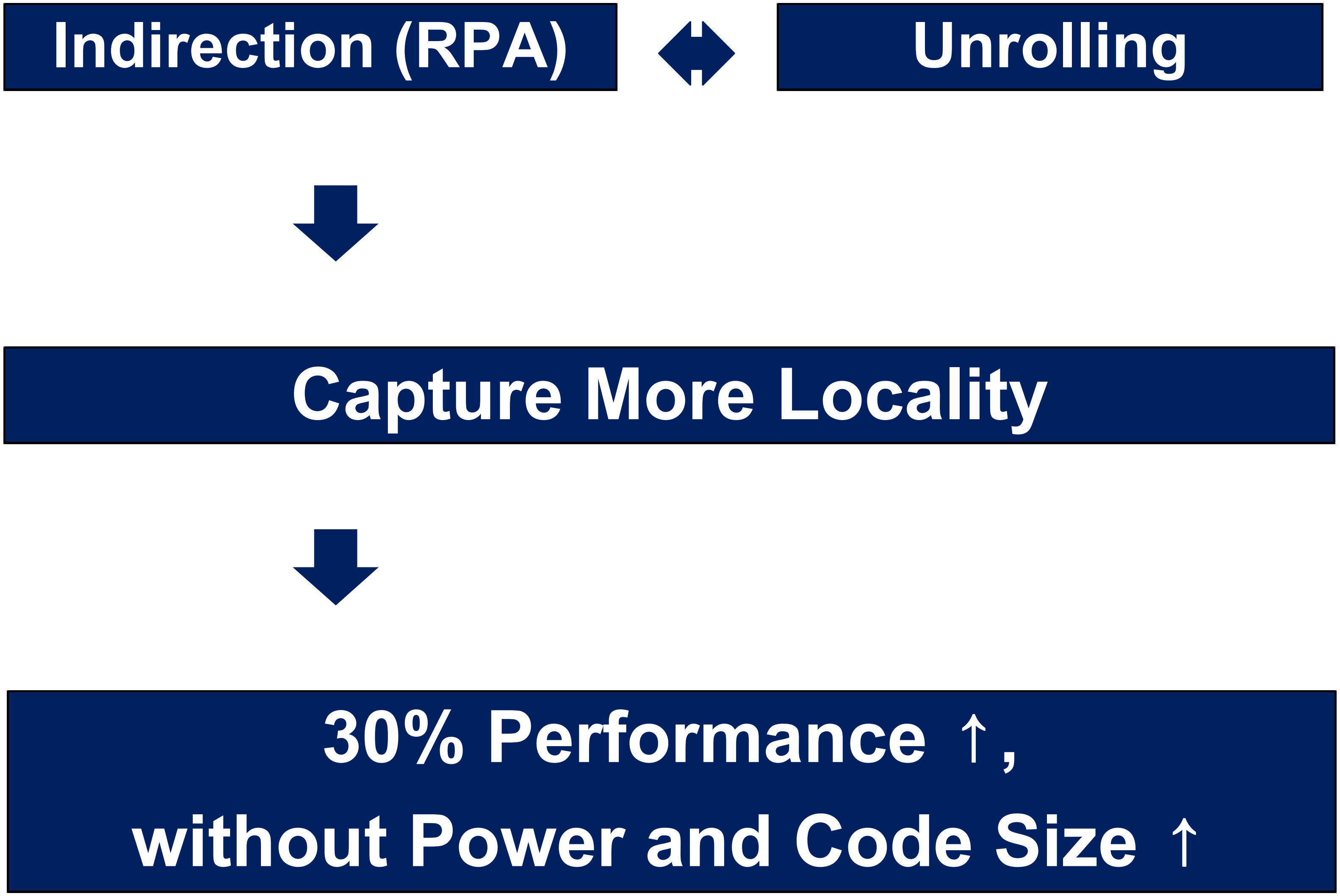


Summary of Comparison





Conclusion



Indirection (RPA) ⇔ Unrolling

Capture More Locality

30% Performance ↑,

without Power and Code Size ↑