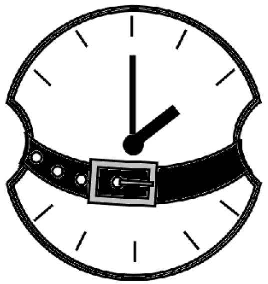
Time Squeezing for Tiny Devices



DAC 2018, ISCA 2019

http://users.cs.northwestern.edu/~simonec/Research_by_project.html

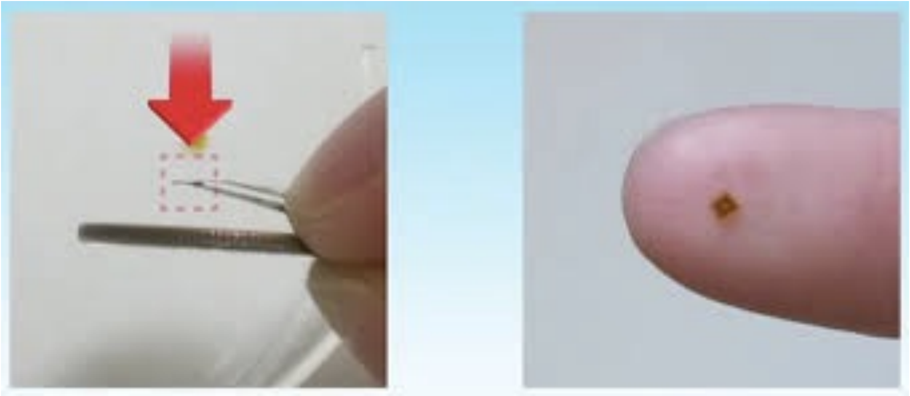
Difficult to achieve energy wins in tiny devices

- Tiny devices include:
  - Energy efficient wearable devices
  - Nano drones
  - Implantable devices
  - Smart city sensors

- Require general purpose CPUs with reasonable performance

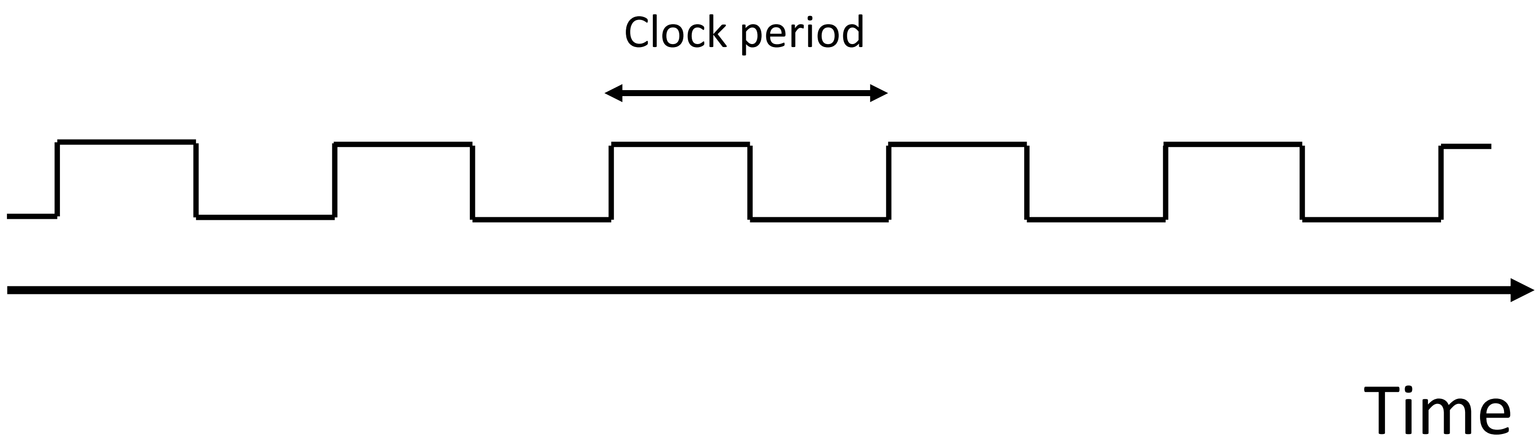
- Difficult to improve efficiency
  - These CPUs are lean and well-optimized already
  - Circuit-level tricks are mostly exhausted
  - End of Moore's Law and Dennard Scaling

SKeye mini Quad copter

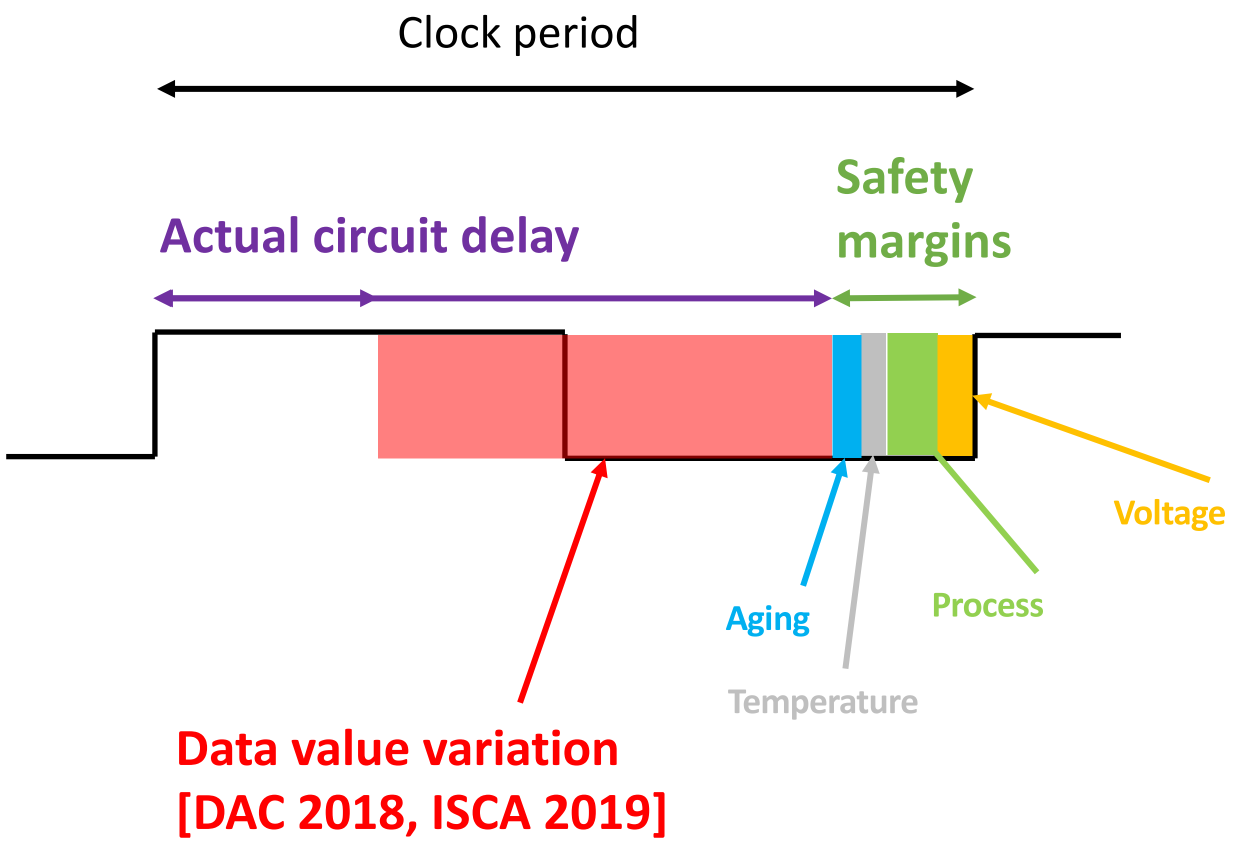


Implantable blood pressure sensor

Looking for efficiency



Overhead:
data-dependent dynamic timing slack (DTS)



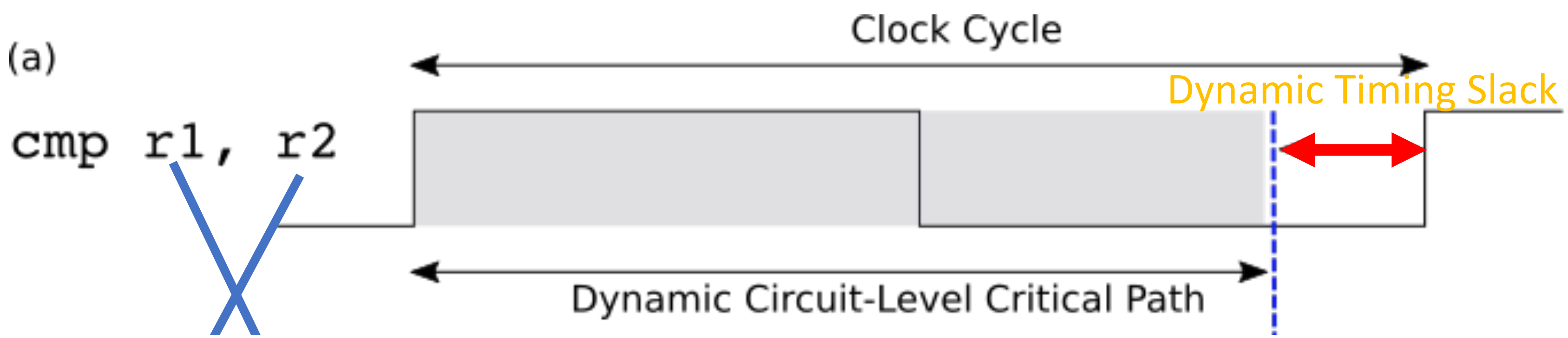
addq rax rbx

10 -1

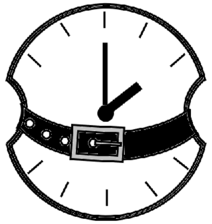
10 1

- The architecture cannot change the data to compute
- but compilers often can

Example of compiler transformation that modifies DTS



Outline

- Data dependent DTS
- Idea behind Time Squeezer 
- Compiler transformations
- Experimental results

Compilers
for Exploiting Data-dependent DTS

Dynamic Timing Slack is limited by combination of code and data

- Introducing Time Squeezer
  - First DTS-aware compiler which considers the impact that data has on timing slack
  - Squeezes operations to expose an additional amount of dynamic timing slack to the hardware
- Placement of data and ways of accessing the data (EA) impact critical paths
- Coupling DTS-aware compilers and architecture saves energy in tiny devices

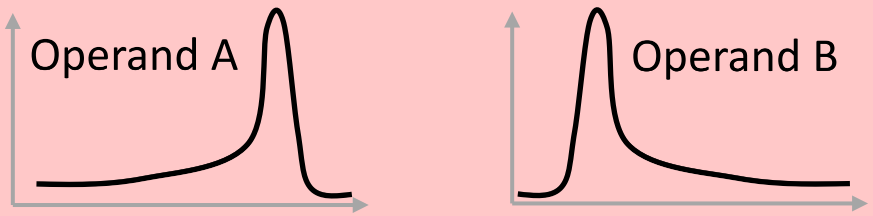
Adders are the workhorses

Adders are used for

A. Adding/subtracting program values

```
000094e0 <susan_principle>:
  94e0:     e92d4ff0     push   {r4, r5, r6, r7, r8, r9, sl, fp, lr}
  ...
  9504:     e50b0020     str    r0, [fp, #-32]  ; 0xffffffe0
  ...
  9510:     e50b2024     str    r2, [fp, #-36]  ; 0xffffffdc
```

B. Computing stack and heap addresses



C. Comparing values

```
if (x_size <= MAX){
   ...
}
```

clang

```
...
cmp r1, r2
...
```

1. Inverting bits of r2
2. Adding 1
3. Adding r1 to the new r2
4. Set the flags

```
susan_principle(...) {
...
  int x_size, y_size;
...
}
```

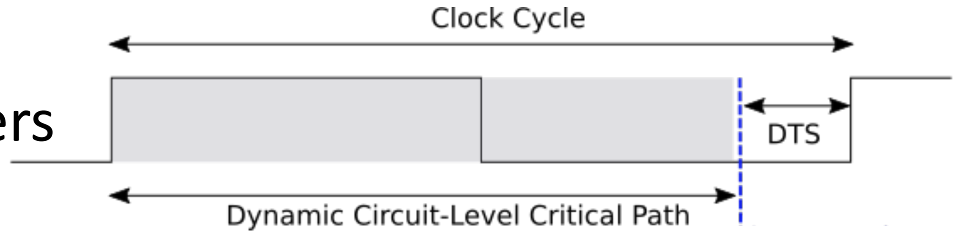
Idea behind Time Squeezer: avoid subtracting low values

- Charry chains in adders lead to long circuit-level latencies

1011 1110 1111 1111 1111 1100 1011 1000 carry chain

0xBEFFFCB8 –
32

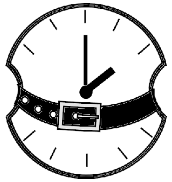
Current compilers

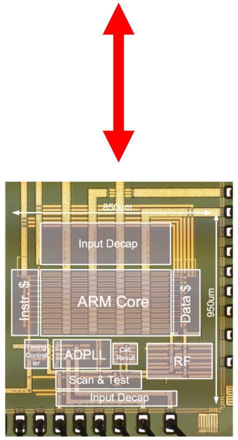


- The idea: a **compiler** that reduces carry chain lengths and an **architecture** to aggressively shrink clock cycles

The Time Squeezer Approach

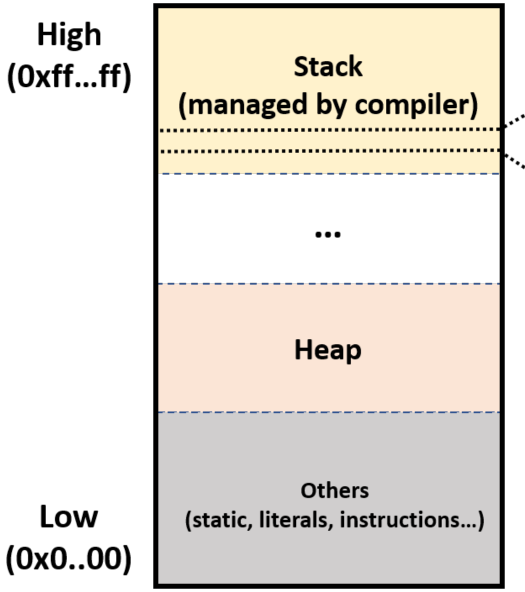
Time Squeezer

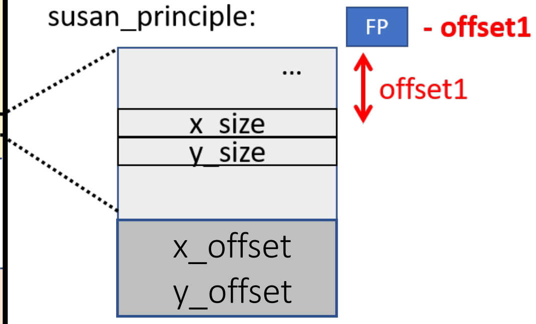




The core uses **40.5%** less energy with Time Squeezer!
(on average among 13 workloads)

Long circuit-level critical path: stack address computation

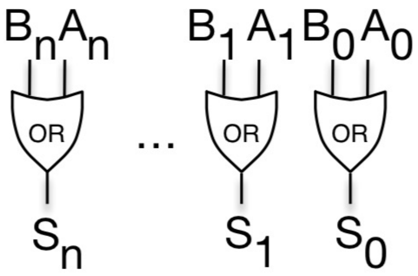





```
susan_principle(…) {
…
  int x_size, y_size;
…
}
```

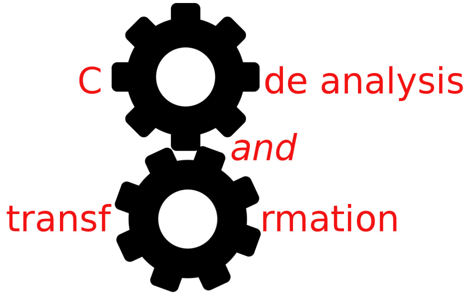
```
000094e0 <susan_principle>:
  94e0:     e92d4ff0     push   {r4, r5, r6, r7, r8, r9, sl, fp, lr}
  …
  9504:     e50b0020     str    r0, [fp, #-32]  ; 0xffffffe0
  …
  9510:     e50b2024     str    r2, [fp, #-36]  ; 0xffffffdc
  …
```

- Optimization 1: access stack locations from the stack pointer (SP)
  - Complexity increases when alloca() is invoked
- Optimization 2: align the SP to a power of 2
  - Instead of an adder, we use OR gates



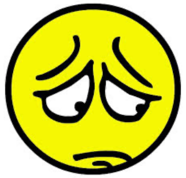
Long circuit-level critical path: heap address computation

... = myObject->field1 ...



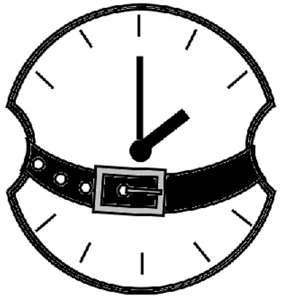

```
p = &(myObject->field1)
for (...){
  p--;
}
```

r1 - 8



- Loop rotation
- Common sub-expression elimination + code scheduling

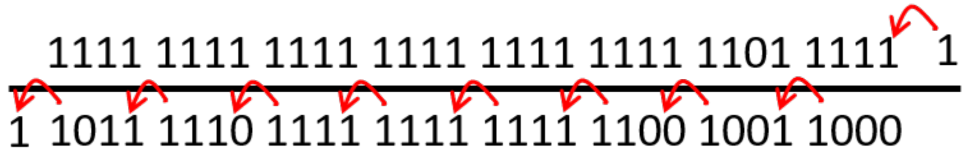
... = myStruct->field1 ...

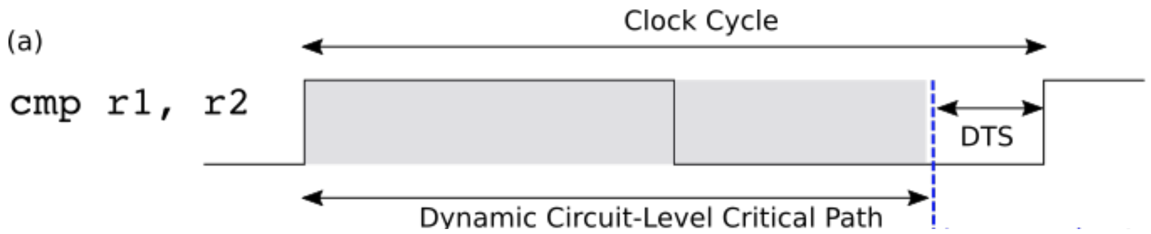


1. Forces field address computation to use object pointer
2. Align object pointer to be a power of 2 for small objects

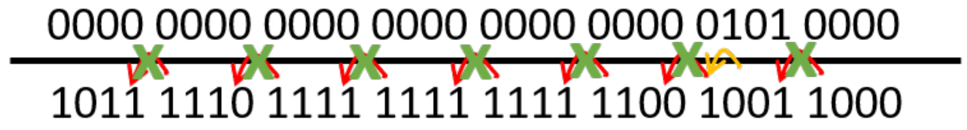
Long circuit-level critical path: values comparison

Inverting a small value (e.g., r2)





Inverting a high value (e.g., r1)



- We run a profiler to understand the likelihood of each bit to be one
- We run a model to compare the two orders (e.g., cmp r1, r2 vs. cmp r2, r1)
- We modify the subsequent branch accordingly
  (like for the translation of “<=“ from L1 to x86_64)

TimeSqueezer: the 1st data-dependent DTS aware compiler

Optimization target:
inversion of small values encoded using the 2-complement representation

The TimeSqueezer compiler

1. Generate comparison instructions
   decreasing the likelihood of inverting small values
2. Layout the stack to avoid the need for inverting small values
3. Layout heap objects to avoid the need for inverting small values
4. Generate code to tune the clock cycle period at run-time ← **Squeeze out DTS**

(Items 1–3:) **Boost DTS**

TimeSqueezer: the 1st data-dependent DTS aware compiler

Optimization target:
inversion of small values encoded using the 2-complement representation

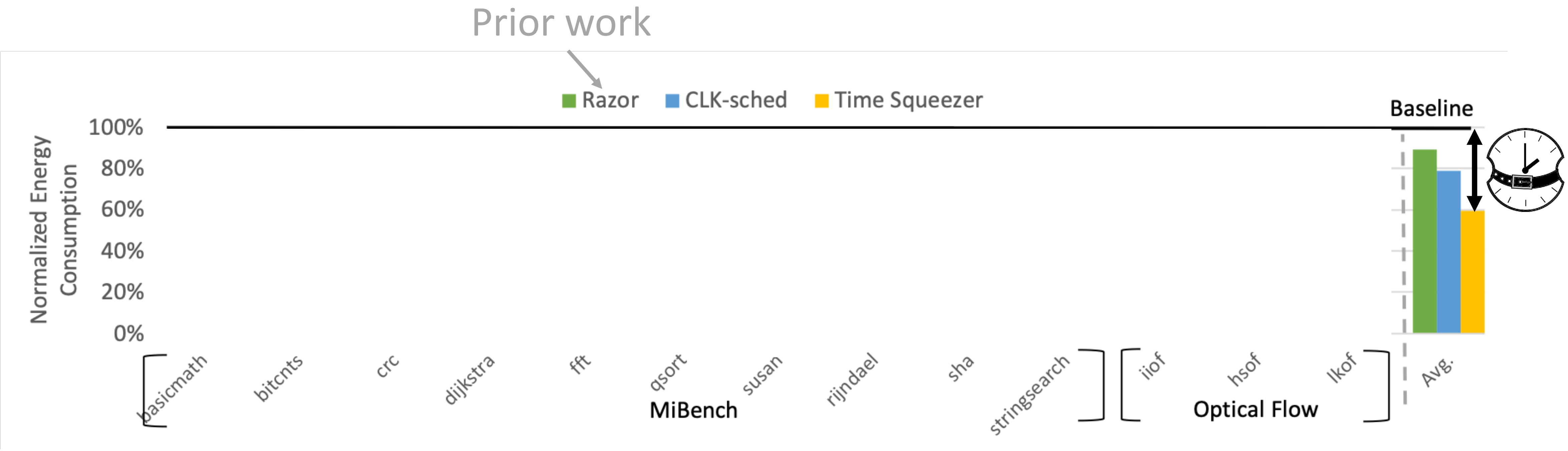
The TimeSqueezer architecture

1. Tune the clock cycle period at run-time
2. Detect timing speculative errors
3. Guarantee correctness thanks to existing recovering mechanisms

TimeSqueezer: the 1st data-dependent DTS aware compiler

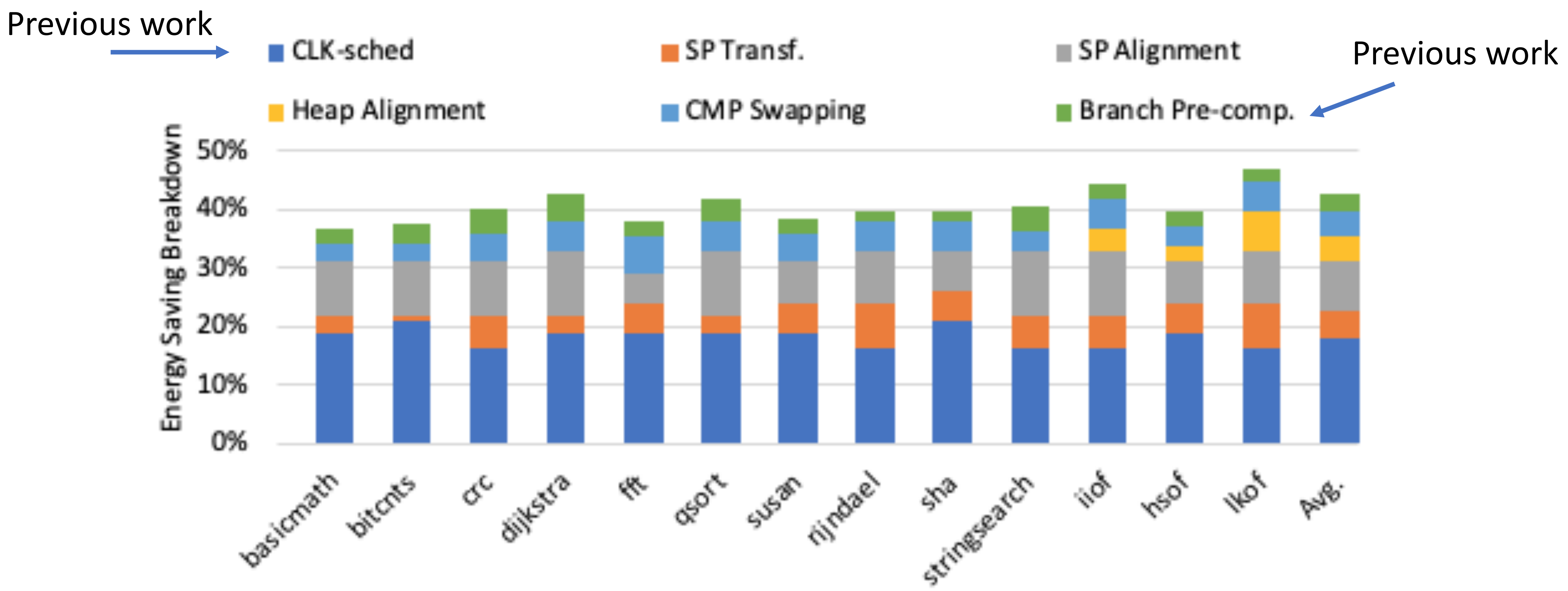
Optimization target:

inversion of small values encoded using the 2-complement representation



Breaking Down Energy Savings

- All of the proposed DTS optimizations contribute to benefits
- Stack alignment has biggest impact on average



Understanding Overheads

- Memory alignment creates some overhead
- Leads to slight increase in cache miss rate
- But there is no tangible performance impact!

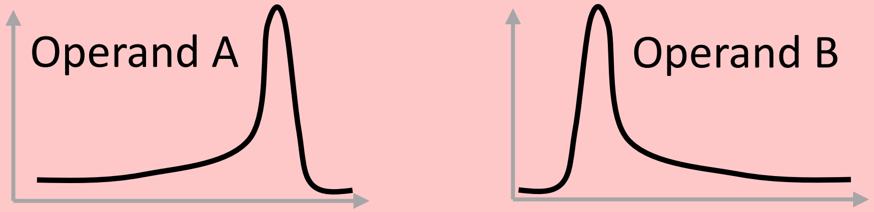
Benchmark	Cache Miss Rate	Memory Overhead	Binary Overhead
basicmath	0.25%	7.19%	3.09%
bitcnt	0.16%	5.11%	3.14%
crc	0.45%	3.41%	8.16%
dijkstra	0.30%	4.40%	9.80%
fft	0.41%	11.9%	9.59%
qsort	0.35%	7.16%	11.86%
susan	0.30%	6.85%	11.39%
rijndael	0.59%	10.3%	5.88%
sha	0.41%	12.6%	14.06%
stringsearch	0.24%	4.42%	5.17%
iiof	0.34%	6.10%	11.27%
hsof	0.28%	7.19%	6.02%
lkof	0.37%	11.5%	9.45%
Mean	**0.35%**	**6.14%**	**8.38%**

Thank you!

Timing slack depends on data

```
000094e0 <susan_principle>:
  94e0:    e92d4ff0    push   {r4, r5, r6, r7, r8, r9, sl, fp, lr}
  ...
  9504:    e50b0020    str    r0, [fp, #-32]  ; 0xffffffe0
  ...
  9510:    e50b2024    str    r2, [fp, #-36]  ; 0xffffffdc
```

- Computing stack and heap addresses



- Comparing values

```
if (x_size <= MAX){
   ...
}
```

clang

```
...
cmp r1, r2
...
```

1. Inverting bits of r2
2. Adding 1
3. Adding r1 to the new r2
4. Set the flags

Always have faith in your ability

Success will come your way eventually

Best of luck!