Time Squeezing for Tiny Devices

DAC 2018, ISCA 2019

www.cs.northwestern.edu/~simonec/Research_by_project.html#RES
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
Looking for efficiency

Clock period

Time
Overhead: data-dependent dynamic timing slack (DTS)

Clock period

Actual circuit delay

Safety margins

Data value variation [DAC 2018, ISCA 2019]

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

B. Computing stack and heap addresses

C. Comparing values

if (x_size <= MAX) {
    ... clamp
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 \textbullet{Time Squeezer:} avoid subtracting low values

\begin{itemize}
\item Charry chains in adders lead to long circuit-level latencies
\end{itemize}

\begin{center}
1011 1110 1111 1111 1111 1100 1011 1000 \text{ carry chain} 0xBEFFFCB8 – 32
\end{center}

Current compilers

\begin{center}
\text{Dynamic Circuit-Level Critical Path}
\end{center}

\begin{itemize}
\item The idea: a \textbf{compiler} that reduces carry chain lengths and an \textbf{architecture} to aggressively shrink clock cycles
\end{itemize}
The Time Squeezer Approach

The core uses 40.5% less energy with Time Squeezer!
(on average among 13 workloads)
Long circuit-level critical path: stack address computation

- Optimization 1: access stack locations from the stack pointer (SP)
  - Complexity increases when \texttt{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--;  \( \rightarrow r1 - 8 \)
}

... = myStruct->field1 ...

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

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 1\textsuperscript{st} 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
TimeSqueezer: the 1\textsuperscript{st} data-dependent DTS aware compiler

\textbf{Optimization target:}
inversion of small values encoded using the 2-complement representation

\textbf{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!

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Cache Miss Rate</th>
<th>Memory Overhead</th>
<th>Binary Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>basicmath</td>
<td>0.25%</td>
<td>7.19%</td>
<td>3.09%</td>
</tr>
<tr>
<td>bitcnt</td>
<td>0.16%</td>
<td>5.11%</td>
<td>3.14%</td>
</tr>
<tr>
<td>crc</td>
<td>0.45%</td>
<td>3.41%</td>
<td>8.16%</td>
</tr>
<tr>
<td>dijkstra</td>
<td>0.30%</td>
<td>4.40%</td>
<td>9.80%</td>
</tr>
<tr>
<td>fft</td>
<td>0.41%</td>
<td>11.9%</td>
<td>9.59%</td>
</tr>
<tr>
<td>qsort</td>
<td>0.35%</td>
<td>7.16%</td>
<td>11.86%</td>
</tr>
<tr>
<td>susan</td>
<td>0.30%</td>
<td>6.85%</td>
<td>11.39%</td>
</tr>
<tr>
<td>rijndael</td>
<td>0.59%</td>
<td>10.3%</td>
<td>5.88%</td>
</tr>
<tr>
<td>sha</td>
<td>0.41%</td>
<td>12.6%</td>
<td>14.06%</td>
</tr>
<tr>
<td>stringsearch</td>
<td>0.24%</td>
<td>4.42%</td>
<td>5.17%</td>
</tr>
<tr>
<td>iiof</td>
<td>0.34%</td>
<td>6.10%</td>
<td>11.27%</td>
</tr>
<tr>
<td>hsof</td>
<td>0.28%</td>
<td>7.19%</td>
<td>6.02%</td>
</tr>
<tr>
<td>lkof</td>
<td>0.37%</td>
<td>11.5%</td>
<td>9.45%</td>
</tr>
<tr>
<td><strong>Mean</strong></td>
<td><strong>0.35%</strong></td>
<td><strong>6.14%</strong></td>
<td><strong>8.38%</strong></td>
</tr>
</tbody>
</table>
Timing slack depends on data

• Computing stack and heap addresses

• Comparing values

```c
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