Designing a Time-Predictable Memory Hierarchy for Single-Path Code

Bekim Cilku, Peter Puschner
Outline

- Introduction
- Single-path code
- Transformation Rules
- Memory hierarchy for single-path code
- Prefetching
- Summary
Software is written to execute fast – different execution paths for different input data;

Hardware features extend the analysis with state dependencies and mutual interferences;

State-of-the-art WCET tools are using integrated approach considering all interferences;
Single-Path Code

- Converts all input-dependant alternatives of the code into pieces of sequential code;

- Properties of SP code:
  - Every execution has the same instruction trace, i.e., the same sequence of references to instruction memory;
  - Eliminates control-flow induced variations in execution time – forces the execution time to become constant;
  - Path analysis is trivial – there is only one path;

```
if(a)
  x=x+1
else
  y=y+1
beq a,0,L1
add x,x,1
jump L2
pred_eq p,a
add x,x,1 (p)
add y,y,1 (not p)
```
The Single-Path Transformation

<table>
<thead>
<tr>
<th>Construct $S$</th>
<th>Translated Construct $\mathcal{SP}[S] \sigma\delta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S$</td>
<td>if $\sigma = T$ then $S$</td>
</tr>
<tr>
<td></td>
<td>otherwise $(\sigma) S$</td>
</tr>
<tr>
<td>$S_1;S_2$</td>
<td>$\mathcal{SP}[S_1] \sigma\delta$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{SP}[S_2] \sigma\delta$</td>
</tr>
<tr>
<td>if $\text{cond}$ then $S_1$ else $S_2$</td>
<td>if $\text{ID}(\text{cond})$</td>
</tr>
<tr>
<td></td>
<td>$\text{guard}_{\delta} := \text{cond}$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{SP}[S_1] \langle \sigma \land \text{guard}_{\delta} \rangle \langle \delta + 1 \rangle$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{SP}[S_2] \langle \sigma \land \neg\text{guard}_{\delta} \rangle \langle \delta + 1 \rangle$</td>
</tr>
<tr>
<td></td>
<td>otherwise</td>
</tr>
<tr>
<td></td>
<td>if $\text{cond}$ then $\mathcal{SP}[S_1] \sigma\delta$</td>
</tr>
<tr>
<td></td>
<td>else $\mathcal{SP}[S_2] \sigma\delta$</td>
</tr>
<tr>
<td>while $\text{cond}$ max $N$ times do $S$</td>
<td>if $\text{ID}(\text{cond})$</td>
</tr>
<tr>
<td></td>
<td>$\text{end}_{\delta} := \text{false}$</td>
</tr>
<tr>
<td></td>
<td>for $\text{count}_{\delta} := 1$ to $N$ begin</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{SP}[\text{if } \neg\text{cond} \text{ then } \text{end}_{\delta} := \text{true}] \langle \delta + 1 \rangle$</td>
</tr>
<tr>
<td></td>
<td>$\mathcal{SP}[\text{if } \neg\text{end}_{\delta} \text{ then } S] \langle \delta + 1 \rangle$</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>otherwise</td>
</tr>
<tr>
<td></td>
<td>while $\text{cond}$ do $\mathcal{SP}[S] \sigma\delta$</td>
</tr>
</tbody>
</table>

- $\sigma$… inherited precondition from previously transformed code constructs;
- $\delta$… counter, used to generate variable names needed for the transformation;
Compositionality of Single-Path Code

- Large programs can be decomposed into smaller segments – each can be analyzed in separation;

- Traditional segment analysis:
  - Wider interfaces between segments – a lot of information gets lost;
  - High complexity for calculating initial states at segment boundaries;
  - Highly pessimistic results;

- Single-path segment analysis:
  - Narrow interface between segments - single trace of execution;
  - Easy calculation of initial state;
  - Accurate results;
The long latency of memory accesses is one of the key performance bottlenecks;

Use “knowledge of the future” properties of single-path code to control cache content and get a higher benefit from locality principle;

Add prefetch unit to hide/reduce memory latency;
Cache Memory

- Two banks allow to overlap the process of fetching with prefetching;
- C-bit prevents replication of requests issued between CPU and prefetcher;
- State of references generated from CPU:
  - No match with Tag columns;
  - Tag match, C=0;
  - Tag match, C=1;
Prefetching Algorithm

- Sequential and non-sequential prefetching:
  - Sequential: simple algorithm;
  - Non-sequential: needs input for target destination;

- RPT entries: trigger line, destination line, count, type;

- RPT output has precedence over NLP;
Architecture of the Memory

Diagram showing the architecture of the memory system, including components such as the prefetch unit, state machine, RPT, next line prefetching, cache, and connections to the CPU and main memory.
Summary

- Memory hierarchy that increases performance for single-path code;
- Higher efficiency of cache by better exploitation of the principle of locality;
- Hardware approach – no overhead by extra instructions;
- Dual-bank approach pipelines CPU and prefetch accesses;
- Cache pollution and useless memory traffic is almost zero;
Thank you