A Low Power Front-End for Embedded Processors Using a Block-Aware Instruction Set

Ahmad Zmily

Computer System Lab
Stanford University
Motivation

• Processor front-end engine
  – Performs control flow prediction & instruction fetch
  – Sets upper limit for performance
    • Cannot execute faster than you can fetch

• Energy and Power efficiency
  – Determines battery life-time
  – Cost of cooling and packaging

• Front-end consumes significant budget of total power
  – Large memory arrays accessed nearly every cycle
    • Instruction cache, predictors, BTB
  – Arrays are sized to achieve good overall performance

• Reduce the size of the front-end structures?
The Problem

- Xscale with small front-end structures
  - 16% decrease in total processor power

- The cost for MediaBench programs
  - 12% performance loss
BLISS

• Focus of this paper
  – Low-power using small front-end structures
  – Eliminate performance degradation through optimizations

• A block-aware instruction set architecture (BLISS)
  – Decouples control-flow prediction from instruction fetching
  – Allows software to help with hardware challenges

• Talk outline
  – BLISS Overview
  – Front-End Optimizations
  – Tools and Methodology
  – Evaluation for Embedded Processors
  – Conclusions
Block-Aware Instruction Set

- BLISS = Block-aware Instruction Set

- Explicit basic block descriptors (BBDs)
  - Stored separately from instructions in the text segment
  - Describe control flow and identify associated instructions

- Execution model
  - PC always points to a BBD, not to instructions
### 32-bit Basic Block Descriptor Format

<table>
<thead>
<tr>
<th>Type</th>
<th>Offset</th>
<th>Length</th>
<th>Instruction Pointer</th>
<th>Hints</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>4</td>
<td>13</td>
<td>3</td>
</tr>
</tbody>
</table>

- **Type**: type of terminating control-flow instruction
  - Fall-through, jump, jump register, branch, call, return

- **Offset**: displacement for PC-relative branches and jumps
  - Offset to target basic block descriptor

- **Length**: number of instruction in the basic block
  - 0 to 15 instructions

- **Instruction pointer**: address of the first instruction in the block
  - Remaining bits from TLB

- **Hints**: optional compiler-generated hints
BLISS Code Example

\[
\text{numeqz}=0;
\]
\[
\text{for } (i=0; i<N; i++)
\]
\[
\text{if } (a[i]==0) \text{ numeqz}++; \]
\[
\text{else foo();}
\]

- Example program in C-source code:
  - Counts the number of zeros in array a
  - Calls foo() for each non-zero element
**BLISS Code Example**

<table>
<thead>
<tr>
<th>BBD1: FT , --- , 1</th>
<th>addu r4,r0,r0</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBD2: B_F , BBD4, 2</td>
<td>L1: lw r6,0(r1)</td>
</tr>
<tr>
<td></td>
<td>bneqz r6,L2</td>
</tr>
<tr>
<td>BBD3: J, BBD5, 1</td>
<td>addui r4,r4,1</td>
</tr>
<tr>
<td></td>
<td>j L3</td>
</tr>
<tr>
<td>BBD4: JAL, FOO, 0</td>
<td>L2: jal FOO</td>
</tr>
<tr>
<td>BBD5: B_B, BBD2, 2</td>
<td>L3: addui r1,r1,4</td>
</tr>
<tr>
<td></td>
<td>bneq r1,r2,L1</td>
</tr>
</tbody>
</table>

- All jump instructions are redundant
- Several branches can be folded in arithmetic instructions
  - Branch offset is encoded in descriptors
BLISS Decoupled Front-End

Basic Block Queue
Descriptor cache replaces BTB

BB-Cache Entry Format

<table>
<thead>
<tr>
<th>tag</th>
<th>type</th>
<th>target</th>
<th>length</th>
<th>instr. pointer</th>
<th>hints</th>
<th>bimod</th>
</tr>
</thead>
<tbody>
<tr>
<td>(4b)</td>
<td>(30b)</td>
<td>(4b)</td>
<td>(13b)</td>
<td>(2b)</td>
<td>(2b)</td>
<td></td>
</tr>
</tbody>
</table>
Agenda

• Motivation
• BLISS Overview

• Front-End Optimizations
• Tools and Methodology
• Evaluation for Embedded Processors
• Conclusions
Instruction Reordering

• Idea: Reorder blocks to improve hit rate and utilization
  – Lay out closely executed blocks in chains using profiling
  – Adjust instruction pointer in block descriptor
**Instruction Reordering**

- Idea: Reorder blocks to improve hit rate and utilization
  - Lay out closely executed blocks in chains using profiling
  - Adjust instruction pointer in block descriptor

**BBB1:** B_F, BBB4, 2

**BBB2:** B_F, BBB_y, 3

**BBB3:** J, BBB_x, 2

**L1:**
- `add r3, r2, r8`
- `bne L2 r3, r12`
- `lw r6, 128(r30)`
- `addu r4, 0, r6`
- `beq L6 r4, r9`

**L2:**
- `addiu r17, r0, 1`
- `add r3, r2, r8`

**Instruction Cache**

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td><strong>bne</strong></td>
<td><strong>lw</strong></td>
<td><strong>addu</strong></td>
</tr>
<tr>
<td><strong>jal</strong></td>
<td><strong>addiu</strong></td>
<td><strong>add</strong></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Prefetching

- BBQ decouples prediction from instruction fetching
  - Predictor runs ahead even when IQ full or I-cache miss
  - Stalls only on BB-cache miss or BBQ
Instruction Prefetching

- BBQ provides early view into instruction stream
  - Guided instruction prefetch
  - I-cache misses can be tolerated
Instruction Prefetching

- Prefetches initiated for potential misses
  - Prop the cache when read port is idle
- Prefetched data in a buffer to avoid cache pollution
  - Pushed into the I-cache after first access
Unified Instruction Cache and BTB

• Programs exhibit different behavior
  – Susceptible to I-cache organization and size (e.g. rasta)
  – Susceptible to BTB organization and size (e.g. adpcm)

• A unified I-cache and BB-cache
  – Cache line has either BBDs or regular instructions
  – Single port accessed by BBD fetch or instruction fetch
    • Instruction fetch returns multiple instructions per cycle

• Difficult with a conventional front-end
  – Same PC used to access I-cache & BTB
    • More conflict misses
    • Need to store extra information to differentiate the two types
  – Sharing single port is difficult
    • Basic-Block Boundaries are not known before decoding
Tagless Instruction Cache

- Idea: exploit tag checks on descriptor accesses
  - Improves I-cache access time, energy, and area
Tagless Instruction Cache

- Idea: exploit tag checks on descriptor accesses
  - Improves I-cache access time, energy, and area
Cache Hints

- General mechanism to attach compiler generated hints
  - Basic-Block granularity
  - No effect on instruction footprint

- Cache placement hints
  - At what cache level it is profitable to place data
  - Heuristic: exclude infrequent and/or high mis-rates blocks

- Cache redistribute hints
  - Hints used as part of the cache index
Agenda

- Motivation
- BLISS Overview
- Front-End Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions
Evaluation Methodology

• Intel XScale PXA270 processor
  – Single issue in-order execution
  – Simulated with Simplescalar & Wattch toolsets
  – MediaBench benchmark suite

• BLISS code generation
  – Static binary translation from MIPS executables
  – Front-end optimizations performed during translation

• Instruction Reordering
  – Pettis and Hansen block-level positioning
Agenda

• Motivation
• BLISS Overview
• Front-End Optimizations
• Tools and Methodology

→ • Evaluation for Embedded Processors
• Conclusions
Performance Analysis

- Instruction Prefetching & Reordering $\Rightarrow$ consistent performance
- Unified I-cache and BTB $\Rightarrow$ for programs stressing BTB
- Tagless I-cache $\Rightarrow$ for programs with BB size of 4 instructions
- Cache hints $\Rightarrow$ consistent performance
• Tagless I-cache achieves lowest energy
  - Except for vortex due to its large BBs

• Combination leads to 19% total energy savings over base
BLISS Vs. Filter Cache

- Filter cache (tiny cache) proposed by kin et al.
  - Using similar optimizations

- BLISS achieves similar power reduction with
  - 9% performance improvement
  - 19% total energy improvement
Conclusions

• BLISS: a block-aware instruction set
  – Block descriptors separate from instructions
  – Expressive ISA to communicate software info and hints

• Enabled front-end optimizations
  – Efficient instruction reordering
  – Accurate instruction prefetching
  – General mechanism to implement cache hints
  – Unified instruction cache and BTB
  – Tagless instruction cache

• Result: Low-Power + Performance + Energy
  – 9% performance improvement
  – 16% total power improvement
  – 19% total energy improvement
Questions?
Microarchitecture parameters

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>BLISS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch Width</strong></td>
<td>1 inst/cycle</td>
<td>1 BB/cycle</td>
</tr>
<tr>
<td><strong>Regular BTB</strong></td>
<td>64-entry, 4-way</td>
<td>64-set, 4-way</td>
</tr>
<tr>
<td><strong>Small BTB</strong></td>
<td>16-entry, 2-way</td>
<td>16-set, 4-way</td>
</tr>
<tr>
<td><strong>Regular I-cache</strong></td>
<td>32 KBytes, 32-way, 32B Blocks, 2-cycle access</td>
<td></td>
</tr>
<tr>
<td><strong>Small I-cache</strong></td>
<td>2 KBytes, 2-way, 32B Blocks, 2-cycle access</td>
<td></td>
</tr>
<tr>
<td><strong>BBQ</strong></td>
<td>–</td>
<td>4 entries</td>
</tr>
<tr>
<td><strong>Execution</strong></td>
<td>single-issue, in-order with 1 INT &amp; 1 FP unit</td>
<td></td>
</tr>
<tr>
<td><strong>Predictor</strong></td>
<td>256-entry bimod with 8 entry RAS</td>
<td></td>
</tr>
<tr>
<td><strong>IQ/RUU/LSQ</strong></td>
<td>16/32/32 entries</td>
<td></td>
</tr>
<tr>
<td><strong>D-cache</strong></td>
<td>32 KBytes, 4-way, 32B blocks, 1 port, 2-cycle access</td>
<td></td>
</tr>
<tr>
<td><strong>L2-cache</strong></td>
<td>128 KBytes, 4-way, 64B blocks, 1 port, 5-cycle access</td>
<td></td>
</tr>
<tr>
<td><strong>Main memory</strong></td>
<td>30-cycle access</td>
<td></td>
</tr>
</tbody>
</table>