Block Aware Instruction Set Architecture

Ahmad Zmily

Computer System Lab
Stanford University
Research Contributions

• **Research focus: Addressing basic challenges with the processor front-end**
  - Performance, energy, power, and code density

• **We describe a block-aware ISA (BLISS)**
  - Defines basic block descriptors separate from instructions
  - Expressive ISA to communicate software info and hints

• **We propose a decoupled front-end for BLISS**
  - Decouples prediction from instruction fetching
  - Replaces branch target buffer with a descriptors’ cache

• **BLISS improves performance, energy consumption, power, and code size**
  - Applies to both low-end and high-end processors
  - Provides a balance between hardware and software front-end features
  - Compares favorably to software-only and hardware-only schemes
Processor Efficiency Metrics

- **High performance**
  - Application and computing demand
  - Image, video, and voice processing common in mobile devices

- **Energy efficiency**
  - Cost for dense servers
  - Determines battery life-time for mobile devices

- **Low power**
  - Cost of cooling and packaging
  - Power delivery and mechanical reliability

- **Dense code size**
  - Impact front-end efficiency
  - Determines cost of program storage devices
Processor Front-End

- Processor front-end engine
  - Delivers instructions to the execution pipeline

- Sets upper limit for performance
  - Cannot execute faster than you can fetch

- Determines energy use
  - What core executes: useful, mispredicted, or no instructions

- Impacts code compression efficiency
Processor Front-End

- PC is used to access I-cache
  - On a miss, stalls until instructions retrieved from lower hierarchy

- Pipeline is empty
  - Execution core is idle
Processor Front-End

- On an instruction cache hit
  - Instructions are pushed into the Instruction Queue (IQ)

```
lw r6, r0(r5)
add r7, r7, r6
add r5, r5, 4
bne r5, r4, L1
lw r6, r0(r5)
add r7, r7, r6
```
Processor Front-End

- What to fetch next?
- Predict if next instruction is a branch or not
  - Direction predictor is accessed each cycle using the PC
Processor Front-End

Instruction cache

lw r6, r0(r5)
add r7, r7, r6
add r5, r5, 4
bne r5, r4, L1

PC

Predictor

Instruction Queue

bne r5, r4, L1
add r5, r5, 4
add r7, r7, r6
lw r6, r0(r5)

• Direction predictor is used also for non-branch PCs
  – Slower training and more interference
  – Not energy efficient

• Skewed address is used (not the branch address)
  – More interference
Processor Front-End

- Predictor predicts a taken branch
  - What is the branch address?

- Need a branch target buffer to predict the branch target
  - Cold misses and conflict misses
Processor Front-End

- What if predictor or BTB is wrong?
  - Pipeline flush after:
    - Decode stage (branch target is wrong or prediction is not correct)
    - Execution of branch

- Multi-cycle instruction cache?
  - Affects prediction accuracy
Front-End Detractors

- **Instruction cache**
  - Misses
  - Multi-cycle cache access

- **Predictors**
  - Target address mispredictions
  - Direction mispredictions

- The cost for a 4-way superscalar processor
  - 48% performance loss
  - 21% increase in total energy consumption
Agenda

- Introduction
- BLISS Overview
- Performance Optimizations
- Energy Optimizations
- Code Size Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions
## Block-Aware Instruction Set

<table>
<thead>
<tr>
<th>Text Segment</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Conventional ISA</strong></td>
<td></td>
</tr>
<tr>
<td><strong>BLISS ISA</strong></td>
<td></td>
</tr>
</tbody>
</table>

- **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

- **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
  - Indicate the encoding size of instructions in the block 16- or 32-bit
  - Branch hints (biased taken/non-taken branches)
BLISS Code Example

numeqz=0;

for (i=0; i<N; i++)
    if (a[i]==0) numeqz++;
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>BBD4: JAL, FOO, 0</td>
<td>j L3</td>
</tr>
<tr>
<td>BBD5: B_B, BBD2, 2</td>
<td>L2: jal FOO</td>
</tr>
<tr>
<td></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 decouples prediction from instruction cache

Extra pipe stage to access BB-cache

L2 Cache

Schedule & Execute

D-Cache

I-cache miss

I-cache prefetch

branch type

mipredicted branch target

PC

Hybrid Predictor

BB-Cache

RAS

basic block target

call return target

BB-cache misses

<basic block>

Basic Block

Descriptor cache replaces BTB

Basic-Block queue decouples prediction from instruction cache

Extra pipe stage to access BB-cache

Cache Entry Format

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>4b</td>
</tr>
<tr>
<td>instr. pointer</td>
<td>13b</td>
</tr>
<tr>
<td>hints</td>
<td>2b</td>
</tr>
<tr>
<td>bimod</td>
<td>2b</td>
</tr>
</tbody>
</table>
**BLISS Front-End**

- **BB-Cache**
- **BBQ**

### BB-Cache
- **d1**
- **d2**
- **d3**
- **d4**
- **d5**

### BBQ

- **PC**
- **Predictor**

- **Single PC**
  - Points always to descriptors

- **BB-cache miss**
  - Wait for refill from L2 cache
  - Back-end only stalls when BBQ and IQ are drained
**BLISS Front-End**

### BB-Cache

<table>
<thead>
<tr>
<th>Type</th>
<th>Target</th>
<th>Length</th>
<th>IP</th>
<th>Pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>FT</td>
<td>--</td>
<td>a1</td>
<td>-</td>
</tr>
<tr>
<td>d2</td>
<td>B_F</td>
<td>d4</td>
<td>a2</td>
<td>0</td>
</tr>
<tr>
<td>d3</td>
<td>J</td>
<td>d5</td>
<td>a4</td>
<td>-</td>
</tr>
<tr>
<td>d4</td>
<td>JAL</td>
<td>FOO</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>d5</td>
<td>B_B</td>
<td>d2</td>
<td>a5</td>
<td>1</td>
</tr>
</tbody>
</table>

### BBQ

- **BB-cache hit**
  - Push descriptor & predicted target in BBQ
    - Instructions fetched and executed later (decoupling)
  - Continue fetching from predicted BBD address
BLISS Front-End

BB-Cache

<table>
<thead>
<tr>
<th>Type</th>
<th>Target</th>
<th>Length</th>
<th>IP</th>
<th>Pred</th>
</tr>
</thead>
<tbody>
<tr>
<td>d1</td>
<td>FT</td>
<td>--</td>
<td>a1</td>
<td>-</td>
</tr>
<tr>
<td>d2</td>
<td>B_F</td>
<td>d4</td>
<td>a2</td>
<td>0</td>
</tr>
<tr>
<td>d3</td>
<td>J</td>
<td>d5</td>
<td>a4</td>
<td>-</td>
</tr>
<tr>
<td>d4</td>
<td>JAL</td>
<td>FOO</td>
<td>0</td>
<td>--</td>
</tr>
<tr>
<td>d5</td>
<td>B_B</td>
<td>d2</td>
<td>a5</td>
<td>1</td>
</tr>
</tbody>
</table>

BBQ

- Control-flow misprediction
  - Flush pipeline including BBQ and IQ
  - Restart from correct BBD address
Agenda

- Introduction
- BLISS Overview
- Performance Optimizations
- Energy Optimizations
- Code Size Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions
Minimizing Instruction Cache Misses

BBQ

- Instruction cache
  - addu r4,r0,r0
  - lw r6,0(r1)
  - bneqz r6,L2
  - addui r4,r4,1

- Instruction Queue
  - addui r4,r4,1
  - bneqz r6,L2
  - lw r6,0(r1)
  - addu r4,r0,r0

• I-cache misses can be tolerated
  - BBQ provides early view into instruction stream
  - Guided instruction prefetch
Tolerating Instruction Cache Latency

- I-cache is not in the critical path for speculation
  - BBDs provide branch type and offsets
  - Multi-cycle I-cache does not affect prediction accuracy
  - BBQ decouples predictions from instruction fetching
- Latency only visible on mispredictions
Accurate Target Prediction

- Full target is stored in the cache
  - Calculate 32-bit instruction target on refill

- Better target prediction
  - No cold-misses for PC-relative branch targets
Efficient Direction Prediction

- Judicious use and training of predictor
  - All PCs refer to basic block boundaries
  - No predictor access for fall-through or jump blocks
Agenda

- Introduction
- BLISS Overview
- Performance Optimizations
- Energy Optimizations
- Code Size Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions
Energy Optimizations

• Front-end engine
  – Determines what the core executes
    • useful, mispredicted, or no instructions at all
  – Consumes a significant percentage of the processor total energy
    • Access large arrays nearly every cycle

• BLISS reduces wasted energy
  – On mispredicted instructions
    • Due to better target and direction prediction
  – On clock tree and processor resources when idling
    • Due to reduced execution time

• BLISS reduces the front-end energy
  – Efficient use and training of the predictor
    • No predictor access for fall-through or jump blocks
  – Efficient Instruction cache access
    • Serial cache access
    • Selective words access
Conventional I-Cache – Parallel Access

- Cache access
  - On the critical path
- Access all arrays
  - Set-Associative cache
  - Not energy efficient
### BLISS I-Cache – Serial Access

- **Cache access**
  - Not on the critical path

- **Serial access**
  - Tag arrays first
  - Data array that hits

- **Increased latency**
  - Tolerated by decoupling
Conventional I-Cache – Full Line Access

- Whole cache line is accessed
  - Not energy efficient
BLISS I-Cache – Selective Line Access

I-Cache Data Array

<table>
<thead>
<tr>
<th>W0</th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
<th>W4</th>
<th>W5</th>
<th>W6</th>
<th>W7</th>
</tr>
</thead>
</table>

Buffer

BBD | Length | Instruction Pointer

- BBD defines exactly the instructions needed
  - Access only the necessary words
Low Power Front-End Option

• Use small front-end arrays
  – Significant area and power savings
  – Negative impact on performance

• BLISS eliminates performance handicap
  – Instruction re-ordering
  – Selective caching
  – Unified BB-Cache and instruction cache
  – Tagless instruction cache
  – Guided instruction prefetching
Agenda

• Introduction
• BLISS Overview
• Performance Optimizations
• Energy Optimizations
• Code Size Optimizations
• Tools and Methodology
• Evaluation for Embedded Processors
• Conclusions
BLISS Code Size

- Naïve code generation
  - 10-20% code size increase compared to 32-bit RISC!
  - A block descriptor per 5 to 10 instructions

- Basic code size optimizations
  - All jump instruction are removed
    - BBD defines both control-flow type and the offset
  - Many conditional branches can be removed
    - Simple condition test encoded in the producing opcode
    - Branch target is provided by the block descriptor
  - Reduces code size increase to 5-7%
Block Subsetting Optimization

- Idea: duplicate descriptors but never instructions
  - Eliminate all instructions in a block if exact sequence found elsewhere in the binary
  - Adjust instruction pointer in block descriptor

<table>
<thead>
<tr>
<th>BBD</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>BBD1: B_F, BBD3, 1</td>
<td>beq L1 r8, r1</td>
</tr>
<tr>
<td>BBD2: J, BBD4, 2</td>
<td>add r3, r2, r8</td>
</tr>
<tr>
<td>BBD3: JAL, FOO, 3</td>
<td>addiu r17, r0, 1</td>
</tr>
<tr>
<td>BBD4:</td>
<td>j L2</td>
</tr>
</tbody>
</table>

L1: lw r6, 128(r30)  
    addu r4, 0, r2  
    add r3, r2, r8  
    jal FOO  

L2: addiu r17, r0, 1
16-bit/32-bit Code Interleaving

- 16-bit encoding leads to significant code savings
  - Limited opcodes, registers, and short immediate and offset fields
  - Performance degradation
    - Increased instruction count

- Goal: 32-bit code performance and 16-bit code size
  - Mix 16-bit code and 32-bit code
  - Use 32-bit code for performance critical sections
  - Use 16-bit code for all non-critical sections

- Interleaving trade-off options:
  - Overhead: Cost of switching between the two encodings
  - Flexibility: granularity of interleaving
Efficient 16-bit/32-bit Code Interleaving

- **MIPS16**: interleaving at function-level
  - JALX instruction is used to switch between functions  ⇒ *Low overhead*
  - Functions include both perf. critical & non-critical code  ⇒ *Not-flexible*

- **Thumb-2, rISA**: instruction-level interleaving
  - A couple of instructions per switch  ⇒ *High overhead*
  - Can switch encoding at arbitrary points  ⇒ *Very-flexible*

- **BLISS**: basic block interleaving
  - A block is either fully perf. critical or fully non-critical
  - Descriptor indicates the encoding size for each block
  - No other overhead/instructions for switch
    ⇒ *Very flexible & Low overhead*
Agenda

- Introduction
- BLISS Overview
- Performance Optimizations
- Energy Optimizations
- Code Size Optimizations
- Tools and Methodology
- Evaluation for Embedded Processors
- Conclusions
Tools

- Developed the BLISS simulator
  - Based on Simplescalar & Wattch toolsets
  - Extended power model for BLISS structures

- Developed a profile driven framework for code generation
  - Static binary translation from MIPS executables
  - Arbitrary programs from high-level languages like C or Fortran
  - Static profitability heuristic for interleaving 16-bit and 32-bit code
  - Enables block-subsetting optimizations
  - Dynamic profile driven optimizations
    - Selective caching
    - Instruction re-ordering
    - Branch hints
# Microarchitecture parameters

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>BLISS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch Width</td>
<td>1 inst/cycle</td>
<td>1 BB/cycle</td>
</tr>
<tr>
<td>BTB</td>
<td>32-entry, 4-way</td>
<td>–</td>
</tr>
<tr>
<td>BB-cache</td>
<td>–</td>
<td>8 KBytes, 4-way, 32B Blocks, 1-cycle access</td>
</tr>
<tr>
<td>I-cache</td>
<td>32 KBytes, 32-way, 32B Blocks, 2-cycle access</td>
<td>24 KBytes, 24-way, 32B Blocks, 2-cycle access</td>
</tr>
<tr>
<td>BBQ</td>
<td>–</td>
<td>4 entries</td>
</tr>
<tr>
<td>Execution</td>
<td>single-issue, in-order with 1 INT &amp; 1 FP unit</td>
<td></td>
</tr>
<tr>
<td>Predictor</td>
<td>256-entry bimod with 8 entry RAS</td>
<td></td>
</tr>
<tr>
<td>IQ/RUU/LSQ</td>
<td>16/32/32 entries</td>
<td></td>
</tr>
<tr>
<td>D-cache</td>
<td>32 KBytes, 4-way, 32B blocks, 1 port, 2-cycle access</td>
<td></td>
</tr>
<tr>
<td>L2-cache</td>
<td>256 KBytes, 4-way, 64B blocks, 1 port, 5-cycle access</td>
<td></td>
</tr>
<tr>
<td>Main memory</td>
<td>30-cycle access</td>
<td></td>
</tr>
</tbody>
</table>

- Intel XScale PXA270 processor
  - MediaBench benchmarks
Agenda

• Introduction
• BLISS Overview
• Performance Optimizations
• Energy Optimizations
• Code Size Optimizations
• Tools and Methodology

• Evaluation for Embedded Processors
• Conclusions
Code Size

- Compression ratio
  - compressed code size over the original code size

- Up to 60% average compression ratio
  - For mediabench applications
# Code Size Statistics

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>MIPS32</th>
<th>BLISS basic Optimization</th>
<th>Block Subset</th>
<th>Selective 16/32 Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Code Size (kb)</td>
<td>J/B inst. Removed</td>
<td># BB</td>
<td>Inst. eliminated</td>
</tr>
<tr>
<td>adpcm</td>
<td>36</td>
<td>1671</td>
<td>2607</td>
<td>2536</td>
</tr>
<tr>
<td>epic</td>
<td>64</td>
<td>2808</td>
<td>4345</td>
<td>4771</td>
</tr>
<tr>
<td>g721</td>
<td>42</td>
<td>1920</td>
<td>2942</td>
<td>3015</td>
</tr>
<tr>
<td>gsm</td>
<td>69</td>
<td>2866</td>
<td>4409</td>
<td>4483</td>
</tr>
<tr>
<td>jpeg</td>
<td>109</td>
<td>4535</td>
<td>6609</td>
<td>8033</td>
</tr>
<tr>
<td>mesa</td>
<td><strong>430</strong></td>
<td><strong>16692</strong></td>
<td><strong>24054</strong></td>
<td><strong>36628</strong></td>
</tr>
<tr>
<td>mpeg2.dec</td>
<td>77</td>
<td>3514</td>
<td>5128</td>
<td>5168</td>
</tr>
<tr>
<td>mpeg2.enc</td>
<td>104</td>
<td>4390</td>
<td>6502</td>
<td>6895</td>
</tr>
<tr>
<td>pegwit</td>
<td>74</td>
<td>2735</td>
<td>4094</td>
<td>5229</td>
</tr>
<tr>
<td>rasta</td>
<td>226</td>
<td>10628</td>
<td>13788</td>
<td>19191</td>
</tr>
</tbody>
</table>
Performance

- Consistent performance advantage for BLISS
  - 10% average improvement over base

- Sources of performance improvement
  - 36% reduction in pipeline flushes compared to base
  - 24% reduction in I-cache misses due to prefetching
Performance vs. Code Size

- Similar performance advantage with block-subsetting
  - Reduces I-cache capacity misses
  - Reduces spatial locality
  - Extra instruction cache misses can be tolerated

- Small degradation with 16-32 blocks
  - ~1% increase in dynamic instruction count
Total Chip Energy

- Total energy = front-end + back-end + all caches
- BLISS leads to 21% total energy savings over base
  - Front-end savings + savings from fewer mispredictions
- Similar performance advantage with code optimizations
  - Small degradation with 16-32 blocks
  - ~1% increase in dynamic instruction count
BLISS Vs. Selective 16-bit code

- BLISS achieves similar code size reduction with
  - 10% performance improvement
  - 21% total energy improvement
Research Papers

- [EuroPar’05] “Improving Instruction Delivery with a Block-Aware ISA”
  - Performance optimizations

- [ISLPED’05] “Energy-Efficient and High-Performance Instruction Fetch using a Block-Aware ISA”
  - Energy optimizations

- [DATE’06] “Simultaneously Improving Code Size, Performance, and Energy in Embedded Processors”
  - Code size optimizations

- [TACO V3, Sep’06] “Block-Aware Instruction Set Architecture”
  - In-depth analysis and sensitivity studies

  - Power optimizations
Conclusions

• Research focus: Addressing basic challenges with the processor front-end
  – Performance, energy, power, and code density

• We describe a block-aware ISA (BLISS)
  – Defines basic block descriptors separate from instructions
  – Expressive ISA to communicate software info and hints

• We propose a decoupled front-end for BLISS
  – Decouples prediction from instruction fetching
  – Replaces branch target buffer with a descriptors’ cache

• BLISS improves performance, energy consumption, power, and code size
  – Applies to both low-end and high-end processors
  – Provides a balance between hardware and software front-end features
  – Compares favorably to software-only and hardware-only schemes
Acknowledgements

Advisor: Prof. Kozyrakis
Associate advisor: Prof. Olukotun
Examination Chair: Prof. Saraswat
Examsining committee member: Prof. Horowitz
My family especially my mother for the endless support
Thank You