Energy-efficient & High-performance Instruction Fetch using a Block-aware ISA

Ahmad Zmily and Christos Kozyrakis

Electrical Engineering Department
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

- However, energy efficiency is also important
  - Dense servers
  - Same processor core in server and notebook chips
  - Environmental concerns

- Focus of this paper
  - Can we build front-ends that achieve both goals?
The Problem

- **Front-end detractors**
  - Instruction cache misses
  - Multi-cycle instruction cache accesses
  - Control-flow mispredictions & pipeline flushing

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

![Graph showing percentage loss for different scenarios: Imperfect Predictor, Imperfect I-Cache, and Imperfect Predictor + Imperfect I-Cache. The graph indicates significant performance and energy loss across all scenarios.](image-url)
BLISS

* A block-aware instruction set architecture
  - Decouples control-flow prediction from instruction fetching
  - Allows software to help with hardware challenges

* Talk outline
  - BLISS overview
    * Instruction set and front-end microarchitecture
  - BLISS opportunities
    * Performance optimizations
    * Energy optimizations
  - Experimental results
    * 14% performance improvement
    * 16% total energy improvement
  - Conclusions
BLISS 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
  - Atomic execution of basic blocks
32-bit 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>9</td>
<td>4</td>
<td>13</td>
<td>2</td>
</tr>
</tbody>
</table>

- **Type**: type of terminating branch
  - Fall-through, jump, jump register, forward/backward branch, call, return, …
- **Offset**: displacement for PC-relative branches and jumps
  - Offset to target descriptor
- **Length**: number of instruction in the basic block
  - 0 to 15 instructions
  - Longer basic blocks use multiple descriptors
- **Instruction pointer**: address of the first instruction in the block
  - Remaining bits from TLB
- **Hints**: optional compiler-generated hints
  - This study: branch hints
  - Biased taken/non-taken branches
BLISS Code Example

```c
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

| BBD1: FT  | addu r4,r0,r0 |
| BBD2: B_F, BBD4, 2 | L1: lw r6,0(r1)  |
| BBD3: J, BBD5, 1 | bneqz r6,L2 |
| BBD4: JAL, FOO, 0 | addui r4,r4,1  |
| BBD5: B_B, 2 | j L3 |
|                  | L2: jal FOO |
|                  | L3: addui r1,r1,4 |
|                  | bneq r1,r2,L1 |

- 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
BLISS Decoupled Front-End

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

- BB-cache miss
  - Wait for refill from L2 cache
  - Calculate 32-bit instruction pointer & target on refill
  - Back-end only stalls when BBQ and IQ are drained

Mispredicted branch target
BLISS Decoupled Front-End

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

Ahmad Zmily, ISLPED’05
Performance Optimizations (1)

- 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

- I-cache misses can be tolerated
  - BBQ provides early view into instruction stream
  - Guided instruction prefetch
Performance Optimizations (2)

- Judicious use and training of predictor
  - All PCs refer to basic block boundaries
  - No predictor access for fall-through or jump blocks
  - Selective use of hybrid predictor for different types of blocks
    - If branch hints are used

- Better target prediction
  - No cold-misses for PC-relative branch targets
  - 36% less number of pipeline flushes with BLISS
Front-End Energy Optimizations (1)

- Access only the necessary words in I-cache
  - The length of each basic block is known
  - Use segmented word-lines

- Serial access of tags and data in I-cache
  - Reduces energy of associative I-cache
    * Single data block read
  - Increase in latency tolerated by decoupling

- Merged I-cache accesses
  - For blocks in BBQ that access same cache lines
Front-End Energy Optimizations (2)

- Judicious use and training of predictor
  - All PCs refer to basic block boundaries
  - No predictor access for fall-through or jump blocks
  - Selective use of hybrid predictor for different types of blocks
    - If branch hints are used

- Energy saved on mispredicted instructions
  - Due to better target and direction prediction
  - The saving is across the whole processor pipeline
    - 15% of energy wasted on mispredicted instructions
Evaluation Methodology

- 4-way superscalar processor
  - Out-of-order execution, two-level cache hierarchy
  - Simulated with Simplescalar & Wattch toolsets
  - SpecCPU2K benchmarks with reference datasets

- Comparison: fetch-target-block architecture (FTB) [Reinman et al.]
  - Similar to BLISS but pure hardware implementation
  - Hardware creates and caches block and hyperblock descriptors
  - Similar performance and energy optimizations applied

- BLISS code generation
  - Binary translation from MIPS executables
## Front-end Parameters

<table>
<thead>
<tr>
<th></th>
<th>Base</th>
<th>FTB</th>
<th>BLISS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fetch Width</strong></td>
<td>4 Instructions</td>
<td>1 Fetch block</td>
<td>1 Basic block</td>
</tr>
<tr>
<td><strong>Target Predictor</strong></td>
<td>BTB: 1K entries 4-way 1 cycle access</td>
<td>FTB: 1K entries 4-way 1 cycle access</td>
<td>BB-cache: 1K entries 4-way 1 cycle access 8 entries per line</td>
</tr>
<tr>
<td><strong>Decoupling Queue</strong></td>
<td>--</td>
<td></td>
<td>8 Entries</td>
</tr>
<tr>
<td><strong>I-cache Latency</strong></td>
<td>2-cycle pipelined</td>
<td></td>
<td>3-cycle pipelined</td>
</tr>
</tbody>
</table>

* BTB, FTB, and BB-cache have exactly the same capacity*
Consistent performance advantage for BLISS

- 14% average improvement over base
- 9% average improvement over FTB

Sources of performance improvement

- 36% reduction pipeline flushes compared to base
- 10% reduction in I-cache misses due to prefetching
FTB vs BLISS

- **FTB** ⇒ higher fetch IPC
  - Optimistic, large blocks needed to facilitate block creation
  - But they lead to overspeculation & predictor interference
    - Bad for performance and energy
- **BLISS** ⇒ higher commit IPC
  - Blocks defined by software
  - Always available in L2 on a miss, no need to recreate
  - But, no hyperblocks
    - Suboptimal only for 1 SPEC benchmark (vortex)
Front-End Energy

- 65% energy reduction in the front-end
  - 40% in the instruction cache
  - 12% in the predictors
  - 13% in the BTB/BB-cache

- Approximately 13% of total chip energy in front-end
  - I-cache, predictors, and BTB are bit SRAMs
Total Chip Energy

- Total energy = front-end + back-end + all caches
- BLISS leads to 16% total energy savings over base
  - Front-end savings + savings from fewer mispredictions
  - FTB leads to 9% savings
- ED2P comparison (appropriate for high-end chips)
  - BLISS offers 83% improvement over base
  - FTB limited to 35% improvement
Conclusions

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

- Enabled optimizations
  - Better prediction accuracy, tolerate I-cache misses
  - Judicious use of I-cache/predictors, less energy on mispredictions

- Result: better performance and energy consumption
  - 14% performance improvement
  - 16% total energy improvement
  - Compares favorably to hardware-only scheme