Conventional & Block-based Trace Caches

• In high performance superscalar processors the instruction fetch bandwidth requirements may exceed what can be provided by conventional instruction cache fetch mechanisms.
• The fetch mechanism is expected to supply a large number of instructions, but this is hindered because:
  – Long instruction sequences are not always in contiguous cache locations.
  – Also it is difficult to fetch a taken branch and its target in a single cycle.
• All these factors lead to limiting the fetch bandwidth of an instruction cache to a basic block.
• All methods perform multiple-branch prediction the cycle before instruction fetch.
• The three challenges in high-bandwidth instruction fetching: multiple-branch prediction, multiple fetch groups, and instruction alignment and collapsing, in previous studies.
• All methods to increase instruction fetching bandwidth perform multiple-branch prediction the cycle before instruction fetch, and fall into two general categories:
  • Enhanced Instruction Caches
  • Trace Cache
Approaches To High-Bandwidth Instruction Fetching:

Enhanced Instruction Caches

• Support fetch of non-contiguous blocks with a multi-ported, multi-banked, or multiple copies of the instruction cache.

• This leads to multiple fetch groups that must be aligned and collapsed at fetch time, which can increase the fetch latency.

• Examples:
  – Branch Address Cache.
  – Collapsing Buffer (CB)
Collapsing Buffer (CB)

- This method works on the concept that there are the following elements in the fetch mechanism:
  - An interleaved I-cache and branch target buffer (BTB),
  - A multiple branch predictor,
  - A collapsing buffer.
- The goal of this method is to fetch multiple cache lines from the I-cache and collapse them together in one fetch iteration.
- This method will require that the BTB be accessed more than once to predict the successive branches after the first one and the new cache line.
- The successive lines from different cache lines must also reside in different cache banks from each other.
- Therefore, this method not only increases the hardware complexity, and latency, but also is not very scalable.
Collapsing Buffer (CB)

- BRANCH TARGET BUFFER
  - 16-way Interleaved
- BTB LOGIC
- MULTIPLE BRANCH PREDICTOR
  - valid instructions bit vector
  - 1st: 0011100111111000
  - 2nd: 000001111111000
- target address
- Line Size = 16 Instructions
- 2-Way Interleaved Instruction Cache
  - interblock branch
- INTERCHANGE / MASK
  - C
  - intrablock branch
- COLLAPSING BUFFER
  - to decoder
Branch Address Cache

• This method has four major components:
  – The branch address cache (BAC),
  – A multiple branch predictor.
  – An interleaved instruction cache.
  – An interchange and alignment network.

• The basic operation of the BAC is that of a branch history tree mechanism with the depth of the tree determined by the number of branches to be predicted per cycle.

• The tree determines the path of the code and therefore, the blocks that will be fetched from the I-cache.

• Again, there is a need for a structure to collapse the code into one stream and to either access multiple cache banks at once or pipeline the cache reads.

• No matter which methods are implemented, the BAC method will add many cycles to the instruction pipeline and also add code complexity.
Approaches To High-Bandwidth Instruction Fetching:

Conventional Trace Caches

• The concept of the trace cache is that it is an instruction cache which captures dynamic instruction sequences and makes them appear contiguous.
• Each line of this cache stores a trace of a dynamic instruction stream.
• The trace cache line size is n and the maximum branch predictions that can be generated is m. Therefore a trace can contain at most n instructions and up to m basic blocks.
• A trace is defined by the starting address and a sequence of m-1 branch predictions. These m-1 branch predictions define the path followed, by that trace, across m-1 branches.
• The first time a control flow path is executed, instructions are fetched as normal through the instruction cache. This dynamic sequence of instructions is allocated in the trace cache after assembly in the fill unit.
• Later, if there is a match for the trace (same starting address and same branch predictions), then the trace is taken from the trace cache and put into the fetch buffer. If not, then the instructions are fetched from the instruction cache.
High Level View of Trace Cache Operation

DYNAMIC INSTRUCTION STREAM

trace \{A,taken,taken\}

later...

trace \{A,taken,taken\}

Fill new trace from instruction cache

Access existing trace using A and predictions(t,t)

TRACE CACHE

1st basic block

2nd basic block

3rd basic block (still filling)

to DECODER

EECC722 - Shaaban
Conventional Trace Cache

Figure 1 - The conventional trace cache.
The Complete Trace Cache Fetch Mechanism
Block-Based Trace Cache

- Block-Based Trace Cache improves on conventional trace cache by instead of explicitly storing instructions of a trace, pointers to blocks constituting a trace are stored in a much smaller trace table.

- The block-based trace cache renames fetch addresses at the basic block level and stores aligned blocks in a block cache.

- Traces are constructed by accessing the replicated block cache using block pointers from the trace table.

- Four major components:
  - the trace table,
  - the block cache,
  - the rename table
  - the fill unit.
Block-Based Trace Cache

- Trace Table
- Block Cache
- History Hash
- Final Collapse
- Fetch Buffer
- Rename Table
- Fill Unit
- Execution Core
- Completion

Flow:
- block_ids
- pre-collapse
- trace_id
- br. hist.
Block-Based Trace Cache: Trace Table

- The Trace Table is the mechanism that stores the renamed pointers (block ids) to the basic blocks for trace construction.
- Each entry in the Trace Table holds a shorthand version on the trace. Each entry consists of a valid bit, a tag, and the block ids of the trace.
- These traces are then used in the fetch cycle to tell which blocks are to be fetched and how the blocks are to be collapsed using the final collapse MUX.
- The next trace is also predicted using the Trace Table. This is done using a hashing function, which is based either on the last block id and global branch history (gshare prediction) or a combination of the branch history and previous block ids.
- The filling of the Trace Table is done in the completion stage. The block ids and block steering bits are created in the completion stage based on the blocks that were executed and just completed.
Trace Table

Hash Function

Next trace_id

b_id0 b_id1 b_id2 b_id3 branch history

tag index

block_ids

v tag 1 2 ... w

Trace Table

w pred. block_ids to the block cache

Hit
Block-Based Trace Cache: Block Cache

• The Block Cache is the storage mechanism for the instruction blocks to execute.
• The Block Cache consists of replicated storage to allow for simultaneous accesses to the cache in the fetch stage.
• The number of copies of the Block Cache will therefore govern the number of blocks allowed per trace.
• At fetch time, the Trace Cache provides the block ids to fetch and the steering bits. The blocks needed are then collapsed into the predicted trace using the final collapse MUX. From here, the instructions in the trace can be executed as normal on the Superscalar core.
Block Cache With Final Collapse MUX

$N = 2^n$ word lines

Instructions from the block fill unit

block_id (n-bit)

direct mapped cache

$F_A i_1 i_2 \ldots i_b$

$\downarrow b_{inst}$

Final Collapse

Fetch Buffer

16
Example Implementation of The Rename Table

Block fetch address

<table>
<thead>
<tr>
<th>Tag</th>
<th>Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>v</td>
<td>tag</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N=8 entries

Block fetch address renamed to a block_id
Block-Based Trace Cache: The Fill Unit

- The Fill Unit is an integral part of the Block-based Trace Cache. It is used to update the Trace Table, Block Cache, and Rename Table at completion time.

- The Fill Unit constructs a trace of the executed blocks after their completion. From this trace, it updates the Trace Table with the trace prediction, the Block Cache with Physical Blocks from the executed instructions, and the Rename Table with the fetch addresses of the first instruction of the execution blocks.

- It also controls the overwriting of Block Cache and Rename Table elements that already exist. In the case where the entry already exists, the Fill Unit will not write the data, so that bandwidth is not wasted.
Performance Comparison:
Block vs. Conventional Trace Cache