Decoupled Fetch/Execute Superscalar Processor Engines

- Superscalar processor micro-architecture is divided into a front-end instruction fetch/decode engine and an execution engine.

  The instruction fetch/fill mechanism serves as the producer of fetched and decoded instructions and the execution engine as the consumer.

  Control dependence provide feedback to the fetch mechanism.

  To maintain high-performance the fetch mechanism must provide a high-instruction bandwidth to maintain a sufficient number of instructions in the instruction buffer window to detect ILP.
Instruction Bandwidth Issues

• In current high performance superscalar processors the instruction fetch bandwidth requirements may exceed what can be provided by conventional instruction cache fetch mechanisms.

• Wider-issue superscalars including those for future multi-threaded cores even have higher instruction-bandwidth needs.

• 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.
    • Due to frequency of branches and the resulting small sizes of basic blocks.
  – Also it is difficult to fetch a taken branch and its target in a single cycle:
    • Current fetch units are limited to one branch prediction per cycle.
    • Thus can only fetch a single basic block per cycle.

• All methods proposed 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
## Typical Branch & Basic Block Statistics

Sample programs: A number of SPEC92 integer benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>taken %</th>
<th>avg basic block size</th>
<th># instr between taken branches</th>
</tr>
</thead>
<tbody>
<tr>
<td>eqntott</td>
<td>86.2%</td>
<td>4.20</td>
<td>4.87</td>
</tr>
<tr>
<td>espresso</td>
<td>63.8%</td>
<td>4.24</td>
<td>6.65</td>
</tr>
<tr>
<td>xlisp</td>
<td>64.7%</td>
<td>4.34</td>
<td>6.70</td>
</tr>
<tr>
<td>gcc</td>
<td>67.6%</td>
<td>4.65</td>
<td>6.88</td>
</tr>
<tr>
<td>sc</td>
<td>70.2%</td>
<td>4.71</td>
<td>6.71</td>
</tr>
<tr>
<td>compress</td>
<td>60.9%</td>
<td>5.39</td>
<td>8.85</td>
</tr>
</tbody>
</table>

Outcome:

Fetching one basic block every cycle may severely limit available instruction bandwidth available to fill instruction buffers/window and execution engine.
Requirements For High-Bandwidth Instruction Fetch Units

• To achieve a high effective instruction-bandwidth a fetch unit must meet the following three requirements:
  – Multiple branch prediction in a single cycle to generate addresses of likely basic instruction blocks in the dynamic execution sequence.
  – The instruction cache must be able to supply a number of noncontiguous basic blocks in a single cycle.
  – The multiple instruction blocks must be aligned and collapsed (assembled) into the dynamic instruction execution sequence or stream.
Interleaved Sequential Core Fetch Unit

- This core fetch unit is implemented using established hardware schemes.
- Fetching up to the first predicted taken branch each cycle can be done using the combination of an accurate multiple branch predictor, an interleaved branch target buffer (BTB), a return address stack (RAS), and a 2-way interleaved instruction cache.
- The core fetch unit is designed to fetch as many contiguous instructions possible, up to a maximum instruction limit and a maximum branch limit.
  - The instruction constraint is imposed by the width of the datapath, and the branch constraint is imposed by the branch predictor throughput.
- For demonstration, a fetch limit of 16 instructions and 3 branches is used.
- The cache is interleaved so that 2 consecutive cache lines can be accessed; this allows fetching sequential code that spans a cache line boundary, always guaranteeing a full cache line or up to the first taken branch.
- This scheme requires minimal complexity for aligning instructions:
  - Logic to swap the order of the two cache lines (interchange switch).
  - A left-shifter to align the instructions into a 16-wide instruction latch, and
  - (3) logic to mask off unused instructions.
- All banks of the BTB are accessed in parallel with the instruction cache. They serve the role of detecting branches in all the instructions currently being fetched and providing their target addresses, in time for the next fetch cycle.
A Current Representative Fetch Unit:

Interleaved Sequential Core Fetch Unit
Multiple Branch Prediction using a Global Pattern History Table (MGAg)

- Algorithm to make 2 branch predictions from a single branch history register:
  - To predict the secondary branch, the right-most k-1 branch history bits are used to index into the pattern history table.
  - k -1 bits address 2 adjacent entries, in the pattern history table.
  - The primary branch prediction is used to select one of the entries to make the secondary branch prediction.
3-branch Predictions/cycle MGAg

GLOBAL HISTORY REGISTER

$2^{14}$ 2-bit counters arranged in $(2^{12} \times 4)$ array

PATTERN HISTORY TABLE

4:1 MUX

3 branch predictions
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:
  – Collapsing Buffer (CB)
  – Branch Address Cache (BAC).
Collapsing Buffer (CB)

• This method works on the concept that there are the following elements in the fetch mechanism:
  – A 2-way interleaved I-cache and
  – 16-way interleaved branch target buffer (BTB),
  – A multiple branch predictor,
  – A collapsing buffer.

• The hardware is similar to the core fetch unit but has two important distinctions.
  – First, the BTB logic is capable of detecting intrablock branches – short hops within a cache line.
  – Second, a single fetch goes through two BTB accesses.

• 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 requires 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 to prevent cache bank conflicts.

• Therefore, this method not only increases the hardware complexity, and latency, but also is not very scalable.
The fetch address A accesses the interleaved BTB. The BTB indicates that there are two branches in the cache line, target address B, with target address C. Based on this, the BTB logic indicates which instructions in the fetched line are valid and produces the next basic block address, C. The initial BTB lookup produces (1) a bit vector indicating the predicted valid instructions in the cache line (instructions from basic blocks A and B), and (2) the predicted target address C of basic block B. The fetch address A and target address C are then used to fetch two nonconsecutive cache lines from the interleaved instruction cache. In parallel with this instruction cache access, the BTB is accessed again, using the target address C. This second, serialized lookup determines which instructions are valid in the second cache line and produces the next fetch address (the predicted successor of basic block C). When the two cache lines have been read from the cache, they pass through masking and interchange logic and the collapsing buffer (which merges the instructions), all controlled by bit vectors produced by the two passes through the BTB. After this step, the properly ordered and merged instructions are captured in the instruction latches to be fed to the decoders.
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.

• The BAC method may result in two extra stages to the instruction pipeline.
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 upon instruction completion, not at fetch time as in previous techniques.
• 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,\text{taken},\text{taken}\} later...

A \quad \quad \quad \quad \quad t \quad \quad \quad \quad \quad t

Fill new trace from instruction cache

TRACE CACHE

1st basic block
2nd basic block
3rd basic block (still filling)

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

TRACE CACHE

to DECODER
Conventional Trace Cache

Figure 1 - The conventional trace cache.
Trace Cache: Core Fetch Unit

- Trace Cache: Structure of the core fetch unit.
- BRANCH TARGET BUFFER: 16-way Interleaved.
- BTB LOGIC: Valid instructions bit vectors.
- MULTIPLE BRANCH PREDICTOR: Target address.
- INTERCHANGE, SHIFT, MASK: To decoder.
The Complete Trace Cache Fetch Mechanism
Trace Cache Line Control Fields

• The control information is similar to the control fields of standard caches but contains additional state information:
  – *valid bit*: indicates this is a valid trace.
  – *tag*: identifies the starting address of the trace.
  – *branch flags*: there is a single bit for each branch within the trace to indicate the path followed after the branch (taken/not taken). The \( m \)th branch of the trace does not need a flag since no instructions follow it, hence there are only \( m-1 \) bits instead of \( m \).
  – *branch mask*: state is needed to indicate (1) the number of branches in the trace and (2) whether or not the trace ends in a branch.
  – *trace fall-through address*: next fetch address if the last branch in the trace is predicted not taken.
  – *trace target address*: next fetch address if the last branch in the trace is predicted taken.
Trace Cache Operation

- The trace cache is accessed in parallel with the instruction cache and BTB using the current fetch address.
- The predictor generates multiple branch predictions while the caches are accessed.
- The fetch address is used together with the multiple branch predictions to determine if the trace read from the trace cache matches the predicted sequence of basic blocks. Specifically a trace cache hit requires that:
  - Fetch address match the tag and the branch predictions match the branch flags.
  - The branch mask ensures that the correct number of prediction bits are used in the comparison.
- On a trace cache hit, an entire trace of instructions is fed into the instruction latch, bypassing the conventional instruction cache.
- On a trace cache miss, fetching proceeds normally from the instruction cache, i.e. contiguous instruction fetching.
- The line-fill buffer logic services trace cache misses:
  - Basic blocks are latched one at a time into the line-fill buffer; the line-fill control logic serves to merge each incoming block of instructions with preceding instructions in the line-fill buffer.
  - Filling is complete when either n instructions have been traced or m branches have been detected in the new trace.
  - The line-fill buffer are written into the trace cache. The branch flags and branch mask are generated, and the trace target and fall-through addresses are computed at the end of the line-fill. If the trace does not end in a branch, the target address is set equal to the fall-through address.
Comparing Trace Cache to Ideal

IPC

verilog  groff  gs  mpeg  jpeg  nroff

IBS benchmark

TC (4KB)  TC (32KB)  ideal

EECC722 - Shaaban
Current Implementations of Trace Cache

- Intel’s P4/Xeon NetBurst microarchitecture is the first and only current implementation of trace cache in a commercial microprocessor.
- In this implementation, trace cache replaces the conventional I-L1 cache.
- The execution trace cache which stores traces of already decoded IA-32 instructions or upos has a capacity 12k upos.

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| TC Nxt IP | TC Fetch | Drive | Alloc | Rename | Que | Sch | Sch | Disp | Disp | RF | RF | Ex | Flgs | Br Ck | Drive |

Basic Pipeline

Basic Block Diagram
Possible Trace Cache Improvements

The trace cache presented is the simplest design among many alternatives:

- **Associativity**: The trace cache can be made more associative to reduce conflict misses.
- **Multiple paths**: It might be advantageous to store multiple paths emanating from a given address. This can be thought of as another form of associativity – path associativity.
- **Partial matches**: An alternative to providing path associativity is to allow partial hits. If the fetch address matches the starting address of a trace and the first few branch predictions match the first few branch flags, provide only a prefix of the trace. The additional cost of this scheme is that intermediate basic block addresses must be stored.
- **Other indexing methods**: The simple trace cache indexes with the fetch address and includes branch predictions in the tag match. Alternatively, the index into the trace cache could be derived by concatenating the fetch address with the branch prediction bits. This effectively achieves path associativity while keeping a direct mapped structure, because different paths starting at the same address now map to consecutive locations in the trace cache.
- **Fill issues**: While the line-fill buffer is collecting a new trace, the trace cache continues to be accessed by the fetch unit. This means a miss could occur in the midst of handling a previous miss.
- **Victim trace cache**: It may keep valuable traces from being permanently displaced by useless traces.
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
- Final Collapse
- Fetch Buffer
- Execution Core
- Rename Table
- Fill Unit
- History Hash
- 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

<table>
<thead>
<tr>
<th>b_id0</th>
<th>b_id1</th>
<th>b_id2</th>
<th>b_id3</th>
<th>branch history</th>
</tr>
</thead>
</table>

Hash Function

Next trace_id

<table>
<thead>
<tr>
<th>tag</th>
<th>index</th>
</tr>
</thead>
</table>

block_ids

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>1</th>
<th>2</th>
<th>...</th>
<th>w</th>
</tr>
</thead>
</table>

Trace Table

Hit

w pred. block_ids to the block cache
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

Instructions from the block fill unit

$N = 2^n$ word lines

$\text{block\_id (n-bit)}$

FA | i1 | i2 | ib
---|----|----|---
    |    |    | ...

direct mapped cache

$\text{copy}_2$

$\text{copy}_3$

$\text{copy}_4$

$\text{Final Collapse}$

16

Fetch Buffer

EECC722 - Shaaban
Example Implementation of The Rename Table

Block fetch address

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

(8 entries, 2-way set associative).

\[
\begin{array}{c|c|c}
 v & \text{tag} & \text{block\_id} \\
\hline
 & 0 & \\
 & 2 & \\
 & 4 & \\
 & 6 & \\
\end{array}
\]

\[
\begin{array}{c|c|c}
 & 1 & \\
 & 3 & \\
 & 5 & \\
 & 7 & \\
\end{array}
\]

\[N=8\text{ entries}\]

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

![Graph comparing IPC vs. bytes for block-based and conventional trace caches.](image-url)