Achieving Out-of-Order Performance with Almost In-Order Complexity

Comprehensive Examination – Part II

By
Raj Parihar
Background Info: About the Paper

- Title
  - “Achieving Out-of-Order Performance with Almost In-Order Complexity”

- Authors
  - Francis Tseng and Yale Patt, U T Austin

- Published in
  - ISCA - 2008
Higher Level Philosophy

- Out-of-Order with wider issue is one of the well known techniques to achieve High-Performance

- Two aspects of Out-of-Order execution
  - Program behavior extraction and Dependence analysis
  - Execution of Instructions and Real computation

- Pure hardware based solutions tend to be more complex due to the extra responsibility of behavior extraction, so as an alternative
  - Let the compiler to assist in extracting the behavior
  - And Hardware to do the execution as suggested by compiler
Motivation

- **Wider Issue width Implementation**
  - Pure hardware solution with wider and complex structures
  - Problems and Bottlenecks
    - Poor scalability as the width grows
    - Increased Power Consumption, and
    - Increased Design Complexity
      - i.e. Register Alias Table (RAT)

- **Solution Proposed – Braid Processing**
  - Software based approach to enable hardware to achieve OOO performance while keeping the complexity In-order
Potential Performance Gain

### Table: Issue Width vs. Speedup

<table>
<thead>
<tr>
<th>Issue Width</th>
<th>Avg Speedup</th>
<th>Max Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>8-wide</td>
<td>44%</td>
<td>75%</td>
</tr>
<tr>
<td>16-wide</td>
<td>83%</td>
<td>230%</td>
</tr>
</tbody>
</table>

### Graph: Speedup (wrt 4-wide issue)

- **Avg Speedup**
  - 8-wide: ~50%
  - 16-wide: ~150%

- **Max Speedup**
  - 8-wide: ~0%
  - 16-wide: ~250%
Outline

- Motivation
- Braid Technique: Overview
- Implementation and Control Flow
- Results and Analysis
- Key Insights and Contributions
- Related work
- Future Scope and Conclusion
Braid Technique: Overview

- A compile-time identified entity that facilitate the execution engine to scale to wider width
  - By exploiting the Small fanout
  - Short lifetime of value in program

- Fanout
  - The number of times the value is used

- Lifetime
  - Number of INS between value’s producer and consumer
Dataflow Graph: Braids

- **Dataflow Graphs**
  - Graph showing the data flow in a program

- **Smaller fanout and short lifetime suggests**
  - Dataflow graph can be partitioned
  - Dataflow graph are more regular than thought

- **Ultimate Goal**
  - Use the compiler to partition the dataflow graph of the programs into sub-graphs
  - Sub-graphs can be more easily processed by underlying hardware
Braid Theory and Analysis

LOOP:
LD R1, 10[R6]
ADD R2, R1, R1
SUB R3, R4, R1
ST R3, 10[R6]
SUB R7, R7, 1
BNZ R7, LOOP

<table>
<thead>
<tr>
<th>SPEC CPU2000</th>
<th># of Braids/ BB</th>
<th>w/o 1-INS Braids</th>
</tr>
</thead>
<tbody>
<tr>
<td>INT</td>
<td>2.8</td>
<td>1.1</td>
</tr>
<tr>
<td>FLOAT</td>
<td>3.8</td>
<td>1.5</td>
</tr>
</tbody>
</table>
Implementation

- Braid Processing
  - Compiler
    - Constructions of Braids
  - ISA
    - Incorporation of Braid information
  - Microarchitecture
    - Awareness of Braids to exploit the information
Braid Construction

- Braid Construction is done at compile time
  - with the help of Binary profiling and translation tool
  - Program records the producer and consumer of each value

- INS from same braids are scheduled as a consecutive sequence
  - Width of Braid is equivalent to ILP within the braid

- Two level of register allocation
  - Internal register architecture stores the temp values
  - Greatly simplifies the overall register file architecture

- Braids are broken if there are not sufficient resources i.e. regs or in case of special store-load pairs (compiler can’t disambiguate)
ISA Extension

- ISA a.k.a. Contract between Software and Hardware
  - Braids are conveyed to microarchitecture by compiler through minor augmentations in ISA
Braid Microarchitecture

Diagram showing the Braid Microarchitecture with the following stages:
- Fetch
- Decode
- Allocate
- Rename
- Distribute
- Bypass
- External Register File
- Generic Microarchitecture
- Braid Microarchitecture
Control Flow and Recovery

- Braids do not span basic block boundaries
  - No guarantee for internal and external registers value if different path is traversed in run time
  - Mainly avoided in order to reduce the complexity

- Checkpoint recovery mechanism to recover from a branch Misprediction
  - Less states because internal values are not needed

- Exception Handling
  - State is rolled back to most recent checkpoint prior to exception
  - Only one BEU execute the instruction – In order execution
  - INS which caused the interrupt calls the handler
Simulation Configuration

- Simulation Platform and Simulator
  - A cycle-accurate, execution-driven simulator
  - Issue width: 8-wide configuration

- Common Parameters
  - L1 I and D – cache: 64 KB, 3-cy latency
  - L2 – cache: 1 MB, 6-cy latency
  - Register file: 256 entry with 16R, 8W ports

- Braid Architecture
  - Significant savings come from shorter rename stage and shorter register access
Simulation Result

Conclusion: For Braid Processing architecture external Regfile with 8-entry is optimum which results into only ~5% degradation in performance.
Simulation Result: Summary

- **Baseline Architecture**
  - 8-wide aggressive OOO processor; 256-entry Register file

- **Braid Microarchitecture**
  - With 8-entry reg file results into < 5% degradation
  - With 6R,3R reg file results into < 0.5% degradation
  - With 2 bypass value (as oppose to 3) degrades with 1%
  - FIFO scheduling with 2/cy and 32-entry degrades with 1%
  - 8 BEUs achieve equivalent 16-wide OOO execution rate

- 8-wide braid microarchitecture achieves **at least** 91% of aggressive OOO architecture’s performance
Key Insights

- Appropriate “Partitioning” is one of the key things to improve the performance i.e.
  - Hardware/Software Partitioning: Braid Processing
  - Resource Partitioning: Register file into EXT and INT

- Bits to think
  - If any particular structure in microarchitecture becomes too COMPLEX, there is a way to mitigate the complexity
    - One possible way is to use hybrid approach
    - Distributed resources as oppose to monolithic implementation
Related Work

- **Block-based Processing**
  - Most of the previous work targeted to exploit the parallelism across the basic building blocks
  - i.e. Multiscalar Architecture
    - Unit is a task – set of basic blocks

- **Dependency chains and sub-graph processing**

- **Register File Implementation**
  - Virtual – Physical register file
    - Delayed allocation of registers
  - 2-level hierarchical register file
  - Register file banking
Future Scope

- **Braid aware compiler and Microarchitecture**
  - As an extension to this work a compiler can more aggressively gather the information about the Braids

- **Clustering of BEUs by grouping a subset together**
  - Values synchronized with a cluster can be really fast

- **Braid aware INS scheduling among basic blocks**
  - Scheduler can make use of braids spanning among two consecutive basic blocks
  - This would eliminate the possibility of future stalls
Braid Across Basic Blocks

- Try to schedule the INS from the braids which span across the basic blocks
  - This would ensure that by the time we reach to next block the source data is ready
Conclusion

- Combined Compiler + Microarchitecture Approach
  - Enable wider issue widths
  - Reduced design complexity
- Most values in a program have short lifetime and small fanout
- Braid microarchitecture uses FIFO scheduler for INS scheduling and partitioned register space
- Performance improvement of ~9% of a very aggressive conventional OOO microarchitecture with in-order complexity
Questions and Comments!