18-547 Lecture 20

*Instruction Level Parallelism*

Instructor: Prof. Falsafi

Electrical & Computer Engineering
Carnegie Mellon University

Slides developed by Profs. Hill, Wood, Sohi, and Smith of University of Wisconsin — Madison

---

**Gshare (McFarling)**

- Reduces the conflicts among program phases
- Distributes patterns across tables
- Uses hashing of PC

![Diagram of Gshare (McFarling)](image-url)
Branch Target Buffer

Start with branch PC and produce target PC

Target PC available 1 cycle earlier
  • no bubble for correct predictions

E.g. implementation: a special cache
  • index and tags: branch PC, data: target PC
  • predicted outcome may be held in same table

Considerations
  • many more bits per entry than branch prediction table
  • size and associativity
  • can be associated with the I-cache
  • store not-taken branches?
Branch Target Cache

Start with branch PC and produce
- prediction + target PC + target instruction

E.g. implementation: a special cache
- index and tags: branch PC
- data: prediction info + target PC + one or more target instr

Could send target instruction rather than branch into ID
- zero cycle branch: “branch folding”

Considerations
- many more bits per entry than branch target buffer
- size and associativity
Call Return Stack

Subroutine return
- use call return stack (in hardware)

General indirect jumps
- more than one target => not just T/NT
- path-based schemes may have potential (later?)

More Advanced Techniques

20 years worth of research
We will learn the details in 18-747
Recent technique: Trace Cache
- Keep a path of instruction traces
- Fold (remove) branches
Also, in case studies:
- Hybrid predictors (local and global) in 21264
Dynamic Branch Prediction Summary

Basics
- results eventually always correct
- squash mispredicted instructions
- don’t slow clock cycle
- very fast on correct predictions
- high prediction accuracy
- not too slow for incorrect predictions

Bottomline
- useful for single-issue pipes
- critical for multiple-issue pipes

What We did & Where To Go

Done with advanced branch prediction
We will continue with
1. Static scheduling (VLIW)
2. Dispatch & issue
3. Renaming
4. Interrupts (we dealt with)
5. Case studies
Software Considerations: Scheduling

Basic Idea
Have compiler reorder code to mitigate effect of dependencies

Three examples:
- loop unrolling
- software pipelining
- trace scheduling

Think of dispatch/issue hardware to move into compiler

Loop Unrolling

With scheduling
loop:

```
ld [r1], f0
stall
add f2, f0, f4
sub r1, #8, r1
bnez r1, loop
st f4, [r1 + 8]
```
Software Pipelining

Loop Unrolling:
- Overlaps loops if no inter-iteration dependence
Will not work for “recurrences”
e.g.
for (l = 1; l <= N; l++)
    sum = sum + a[l] * b[l]
Software pipelining will do better
Originally done by microcode people
Rediscovered by Davidson and later Monica Lam

Software Pipelining: What is it?

Sum = 0.0
for (l = 1; l <= N; l++)
    sum = sum + a[l] * b[l]

every iteration =>
load a[l]
load b[l]
mult ab[l]
add sum[l]
Software Pipelining Cont.

Start-up block
load a[1]
load b[1]
load a[2]
load b[2]
mult ab[1]

Pipeline
for (l=3;l<=N;l++)
load a[l]
load b[l]
mult ab[l-1]
add sum[l-2]

Finish-up block
mult ab[N]
add sum[N-1]
add sum[N]

Software Pipelining: Timing

<table>
<thead>
<tr>
<th>a1</th>
<th>a2</th>
<th>a3</th>
<th>a4</th>
<th>...</th>
<th>aN</th>
</tr>
</thead>
<tbody>
<tr>
<td>b1</td>
<td>b2</td>
<td>b3</td>
<td>b4</td>
<td></td>
<td>bN</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*N-1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>+N-2</td>
</tr>
</tbody>
</table>

Copyright © 2001 Faisal, from Hill, Smith, Sohi, Vijaykumar, and Wood

© Chapter 4: Instruction Level Parallelism
Software Pipelining: When?

- Will work for recurrences in loops
- Will not work for non-loop code
- Will not work well for loops with branches
- Pipeline depth depends on recurrence

Trace Scheduling

- Statically schedule all code not just loops
- What is the key impediment?
  - Branches
- Can we predict which way a branch goes?
  - Common case (for error code)
- Trace scheduling:
  - Take the common case path (called a trace)
  - Remove the branch
  - Schedule code
  - Check later for branch
  - If the guess was wrong, execute “undo” code
### Trace Scheduling: Example

Original

\[
\begin{align*}
  b[i] & = \text{"old"} \\
  a[i] & = \\
  \text{if } (a[i] > 0) \text{ then} & \\
  b[i] & = \text{"new"} \\
  \text{else} & \\
  X & \\
  \text{endif} & \\
  c[i] & = \\
\end{align*}
\]

Trace

\[
\begin{align*}
  b[i] & = \text{"old"} \\
  a[i] & = \\
  b[i] & = \text{"new"} \\
  c[i] & = \\
  \text{if } (a[i] <= 0) \text{ goto fixit} & \\
\end{align*}
\]

Repair code

fixit:

\[
\begin{align*}
  \text{restore old } b[i] & \\
  X & \\
  \text{recalculate } c[i]? & \\
\end{align*}
\]

### Static Scheduling: Summary

Loop unrolling:

- ✔ large block to schedule
- ✔ reduces branch frequency
- ✗ expands code size
- ✗ have to handle “extra” iterations

Software pipelining:

- ✔ no dependences in loop body
- ✗ does not reduce branch frequency
- ✗ need start-up and finish-up blocks
**Static Scheduling: Summary**

Trace scheduling:
- ✓ works for non-loops
- ✓ more complex than unrolling and software pipelining
- ✗ does not seem to handle more general cases

---

**Ambigious Dependences**

Software limitation
What compiler can’t analyze => can’t schedule
Ambigious memory dependences

- e.g.,
  *ptr1 =
  … = *ptr2 + 4
  e.g.,
  a[b[I]] =
  ….. = a[c[I]] -5

Registers don’t cause this. Why?
Software vs. Hardware

Equivalent techniques
• differ in applicability

Hardware:
• high branch prediction accuracy
• has dynamic information on latencies (e.g., cache misses)
• transparent!
• works for generic code (any code)
  • databases, desktop applications, numeric code, compilers
• limited reorder buffer size
• high cost/complexity

Software:
• can look at large amounts of code
• no hardware cost
• works for regular code
  • engineering applications, numeric code, weather prediction
• low branch prediction accuracy
• does not have dynamic information on latencies
VLIW: Software ILP

Very long instruction word
Provide a long instruction word with one operation per FU
Instruction latencies are fixed
Compiler packs independent instructions into VLIW
  □ compiler schedules all hardware resources
Entire long word issues as a “unit”
Result: ILP ith simple hardware, simple control, fast clock

VLIW: Scheduling

Code scheduling in software only
Loop unrolling, software pipelining, trace scheduling
Architectural support
  □ deferred interrupts
    □ enhances scheduling opportunity
  □ predicated execution (conditional moves)
    □ less need for hardware prediction
  □ more registers
    □ renaming not necessary?
VLIW: History

Floating point systems, array processors
- very successful in 70’s
- all latencies fixed
- fast, single-level memory

Multiflow
- Josh Fisher (now at HP)
- 1980’s minicomputer => failed

Cydrome
- Bob Rau (now at HP)
- 1980’s minicomputer => failed

Tera
- Burton Smith, 1980’s-1990’s supercomputer, keeps failing?

Intel IA-64
- co-developed by HP (surprise, surprise!)
- will it succeed?
- latest news => compiler works well for regular applications!
VLIW Problems

Same as those with static scheduling
Latencies are not fixed!
Caches are here to stay
Need mechanisms to deal with latency dynamically

Instruction Dispatch & Issue

Places data may potentially reside:
1. registers
2. reservation stations
3. within functional units
4. reorder buffer

Places instructions issue from:
1. single shared queue
2. per-instruction type queue
3. reservation stations
Logical registers correspond to physical storage
Reservation stations are randomly accessed
No reorder buffer => imprecise interrupts
RS6000 (POWER1)'s Method

At dispatch, rename registers

Instruction issue queues
- dynamic issue across (not within) queues

Conventional dataflow:
- only between registers & FUs

Precise state maintained in piecemeal fashion
- condition registers use history file
- integer instrs complete in order (FP not allowed to pass int)
- floating point exceptions are imprecise

Register Renaming

More physical registers than logical registers
Renaming avoids WAW and WAR hazards
Turns instruction stream into “single assignment” form
Assume 8 logical, 16 physical registers
Register Renaming

More physical registers than logical registers
Renaming avoids WAW and WAR hazards
Turns instruction stream into “single assignment” form
Assume 8 logical, 16 physical registers

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register Map (before)</th>
<th>Register Map (after)</th>
</tr>
</thead>
<tbody>
<tr>
<td>If f6, 34(r20)</td>
<td>Free list = p0, p2, p4, p6, p8</td>
<td>f6-&gt;p0</td>
</tr>
<tr>
<td>If f2, 45(r3)</td>
<td>f3 -&gt; p2, f2-&gt;p6</td>
<td></td>
</tr>
<tr>
<td>mul f4, f2, f6</td>
<td>f4 -&gt; p4, f4-&gt;p8</td>
<td></td>
</tr>
<tr>
<td>sub f2, f6, f4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>div f4, f2, f6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

POWER1

Reclaiming registers (returning to free pool)
Ignoring precise interrupts,
  ● a register can be returned after last read is done
Considering interrupts
  ● a register can be returned after it is logically over-written
This can be done in many ways
  ● e.g., via usage counter per register
  ● later we will see a straightforward way