Compiler Techniques for ILP

So far we have explored dynamic hardware techniques for ILP exploitation:
- BTB and branch prediction
- Dynamic scheduling
- Scoreboard
- Tomasulo’s algorithm
- Speculation
- Multiple issue

How can compilers help?

Loop Unrolling

Let’s look at the code:

\[
\text{for } (i=1000; i>0; i=i-1) \\
x[i] = x[i] + s
\]

Scheduling On A 5 Stage MIPS

Loop:
- \( \text{L.D F0,0(R1)} \)
- stall, wait for F0 value to propagate
- ADD.D F4, F0, F2
- stall, wait for FP add to be completed
- S.D F4, 0(R1)
- DADDUI R1, R1, -8
- stall, wait for R1 value to propagate
- BNE R1, R2, Loop

We Could Rearrange Instructions

Loop:
- \( \text{L.D F0,0(R1)} \)
- stall, wait for F0 value to propagate
- ADD.D F4, F0, F2
- stall, wait for FP add to be completed
- S.D F4, 0(R1)
- DADDUI R1, R1, -8
- stall, wait for R1 value to propagate
- BNE R1, R2, Loop

10 cycles

6 cycles

Loop Unrolling

Getting more useful instructions into the loop and reducing overhead

Step 1: Put several iterations together

Loop:
- \( \text{L.D F0,0(R1)} \)
- ADD.D F4, F0, F2
- S.D F4, 0(R1)
- DADDUI R1, R1, -8
- BNE R1, R2, Loop

Step 2: Take out overhead, adjust offsets

Loop:
- \( \text{L.D F0,0(R1)} \)
- ADD.D F4, F0, F2
- S.D F4, 0(R1)
- DADDUI R1, R1, -8
- BNE R1, R2, Loop

Assume taken

BNE R1, R2, Loop

10 cycles

BNE R1, R2, Loop

BNE R1, R2, Loop
Loop Unrolling

Step 3: Rename registers

Loop:
L.D F0,0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
L.D F0, -8(R1)
ADD.D F4, F0, F2
S.D F4, -8(R1)
L.D F0, -16(R1)
ADD.D F4, F0, F2
S.D F4, -16(R1)
L.D F0, -24(R1)
ADD.D F4, F0, F2
S.D F4, -24(R1)
DADDUI R1, R1, #-32
BNE R1, R2, Loop

Loop Unrolling

Current loop still has stalls due to RAW dependencies

Loop:
L.D F0,0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
L.D F6, -8(R1)
ADD.D F8, F6, F2
S.D F8, -8(R1)
L.D F10, -16(R1)
ADD.D F12, F10, F2
S.D F12, -16(R1)
L.D F14, -24(R1)
ADD.D F16, F14, F2
S.D F16, -24(R1)
DADDUI R1, R1, #-32
BNE R1, R2, Loop

Loop Unrolling

Step 4: Interleave iterations

14 cycles = ~3.5 per it.

Loop:
L.D F0,0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
L.D F6, -8(R1)
ADD.D F8, F6, F2
S.D F8, -8(R1)
L.D F10, -16(R1)
ADD.D F12, F10, F2
S.D F12, -16(R1)
L.D F14, -24(R1)
ADD.D F16, F14, F2
S.D F16, -24(R1)
L.D F18, -32(R1)
ADD.D F20, F18, F2
S.D F20, -32(R1)
DADDUI R1, R1, #-32
BNE R1, R2, Loop

Loop Unrolling + Multiple Issue

Let’s unroll the loop 5 times, mark integer and FP operations

Loop:
L.D F0,0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
L.D F6, -8(R1)
ADD.D F8, F6, F2
S.D F8, -8(R1)
L.D F10, -16(R1)
ADD.D F12, F10, F2
S.D F12, -16(R1)
L.D F14, -24(R1)
ADD.D F16, F14, F2
S.D F16, -24(R1)
L.D F18, -32(R1)
ADD.D F20, F18, F2
S.D F18, -32(R1)
DADDUI R1, R1, #-32
BNE R1, R2, Loop

Loop Unrolling + Multiple Issue

Rearrange instructions to handle delay for DADDUI and BNE

Loop:
L.D F0,0(R1)
ADD.D F4, F0, F2
ADD.D F8, F4, F2
ADD.D F12, F8, F2
ADD.D F16, F12, F2
ADD.D F20, F16, F2
S.D F4, 0(R1)
S.D F8, -8(R1)
S.D F12, -16(R1)
S.D F16, -24(R1)
S.D F20, -32(R1)
DADDUI R1, R1, #40
BNE R1, R2, Loop

Loop Unrolling + Multiple Issue

Move all loads first, then ADD.D then S.D

Loop:
L.D F0,0(R1)
ADD.D F4, F0, F2
ADD.D F8, F4, F2
ADD.D F12, F8, F2
ADD.D F16, F12, F2
ADD.D F20, F16, F2
S.D F4, 0(R1)
S.D F8, -8(R1)
S.D F12, -16(R1)
S.D F16, -24(R1)
S.D F20, -32(R1)
DADDUI R1, R1, #40
BNE R1, R2, Loop

Loop Unrolling

28 cycles = ~7 per it.

Loop:
L.D F0,0(R1)
ADD.D F4, F0, F2
ADD.D F8, F4, F2
ADD.D F12, F8, F2
ADD.D F16, F12, F2
ADD.D F20, F16, F2
S.D F4, 0(R1)
S.D F8, -8(R1)
S.D F12, -16(R1)
S.D F16, -24(R1)
DADDUI R1, R1, #40
BNE R1, R2, Loop
Loop Unrolling + Multiple Issue

- Fix immediate displacement values

  Loop:
  L.D F0,0(R1)
  L.D F10,-16(R1)
  L.D F18,-32(R1)
  ADD.D F4, F0, F2
  ADD.D F12, F10, F2
  ADD.D F16, F14, F2
  ADD.D F20, F18, F2
  S.D F4, 0(R1)
  S.D F8, -8(R1)
  DADDUI R1, R1, #-40
  S.D F16, 16(R1)
  BNE R1, R2, Loop

- Now imagine we can issue 2 instructions per cycle, one integer and one FP

  Loop:
  L.D F0,0(R1)
  L.D F6,-8(R1)
  L.D F10,-16(R1)
  L.D F14,-24(R1)
  ADD.D F4, F0, F2
  ADD.D F6, F10, F2
  ADD.D F8, F14, F2
  ADD.D F10, F18, F2
  ADD.D F12, F20, F2
  ADD.D F14, F24, F2
  ADD.D F16, F28, F2
  S.D F4, 0(R1)
  S.D F8, -8(R1)
  S.D F10, -16(R1)
  S.D F14, -24(R1)
  DADDUI R1, R1, #-56
  S.D F20, 16(R1)
  BNE R1, R2, Loop

12 cycles = 2.4 per it.

Static Branch Prediction
- Analyze the code, figure out which outcome of a branch is likely
  - Always predict taken
  - Predict backward branches as taken, forward as not taken
  - Predict based on the profile of previous runs
- Static branch prediction can help us schedule delayed branch slots

Static Multiple Issue: VLIW
- Compiler can examine instructions and decide which ones can be scheduled in parallel – group instructions into instruction packets – VLIW
- Hardware can then be simplified
- Processor has multiple functional units and each field of the VLIW is assigned to one unit
  - For example, VLIW could contain 5 fields and one has to contain ALU instruction or branch, two have to contain FP instructions and two have to be memory references

Example
- Assume VLIW contains 5 fields: ALU instruction or branch, two FP instructions and two memory references (ignore branch delay slot)

  Loop:
  L.D F0,0(R1)
  Memory reference
  stall, wait for F0 value to propagate
  ADD.D F4, F0, F2
  FP instruction
  stall, wait for FP add to be completed
  S.D F4, 0(R1)
  Memory reference
  stall, wait for R1 value to propagate
  DADDUI R1, R1, #-8
  ALU instruction
  stall, wait for F1 value to propagate
  BNE R1, R2, Loop
  ALU instruction

- Unroll seven times and rearrange

  Loop:
  L.D F0,0(R1)
  L.D F6,-8(R1)
  L.D F10,-16(R1)
  L.D F14,-24(R1)
  L.D F18,-32(R1)
  L.D F22,-40(R1)
  L.D F26,-48(R1)
  ADD.D F4, F0, F2
  BNE R1, R2, Loop
  ADD.D F8, F6, F2
  ADD.D F10, F10, F2
  ADD.D F12, F14, F2
  ADD.D F16, F18, F2
  ADD.D F20, F20, F2
  ADD.D F24, F22, F2
  ADD.D F28, F26, F2

  ALU/branch | FP | FP | mem | mem

  1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12
Example

Loop:

L.D F0, 0(R1)
L.D F6, 0(R1)
L.D F10, 16(R1)
L.D F14, 24(R1)
L.D F18, 32(R1)
L.D F22, 40(R1)
L.D F26, 48(R1)
ADD.D F4, F0, F2
ADD.D F8, F6, F2
ADD.D F12, F10, F2
ADD.D F16, F14, F2
ADD.D F20, F18, F2
ADD.D F24, F22, F2

S.D F4, 0(R1)
S.D F8, -8(R1)
S.D F12, -16(R1)
S.D F16, -24(R1)
S.D F20, 24(R1)
DADDUI R1, R1, #56
S.D F24, 16(R1)
BNE R1, R2, Loop
S.D F28, 8(R1)

Example

Loop:

L.D F0, 0(R1)
L.D F6, 0(R1)
L.D F10, 16(R1)
L.D F14, 24(R1)
L.D F18, 32(R1)
L.D F22, 40(R1)
L.D F26, 48(R1)
ADD.D F4, F0, F2
ADD.D F8, F6, F2
ADD.D F12, F10, F2
ADD.D F16, F14, F2
ADD.D F20, F18, F2
ADD.D F24, F22, F2

S.D F4, 0(R1)
S.D F8, -8(R1)
S.D F12, -16(R1)
S.D F16, -24(R1)
S.D F20, 24(R1)
DADDUI R1, R1, #56
S.D F24, 16(R1)
BNE R1, R2, Loop
S.D F28, 8(R1)

Detecting and Enhancing Loop Level Parallelism

- Determine whether data in later iterations depends on data in earlier iterations – loop-carried dependence

- Easier detected at source code level than at machine code

tfor(i=1; i<100; i=i+1)
{
A[i] = A[i-1] + B[i]; // S1
B[i] = C[i] + D[i]; // S2
}

S1 calculates a value A[i+1] which will be used in the next iteration of S1
S2 calculates a value B[i+1] which will be used in the current iteration of S2
⇒ This is a loop-carried dependence

S1 calculates a value A[i+1] which will be used in the current iteration of S2
⇒ This is dependence within the loop

Recursion creates loop-carried dependence

tfor(i=1; i<100; i=i+1)
{
}

But sometimes it may parallelizable if distance between dependent elements is >1

tfor(i=1; i<100; i=i+1)
{
}

Find all dependencies in the following loop (5) and eliminate as many as you can:

for(i=1; i<100; i=i+1)
{
Y[i] = X[i] / C[i]; // S1
X[i] = X[i] + C[i]; // S2
}

Y[i] = X[i] / C[i]; // S1
X[i] = X[i] + C[i]; // S2

Solution at page 325
Code Transformation

- Eliminating dependent computations
  - Copy propagation
    \[ \text{DADDUI R1, R2, #4} \rightarrow \text{DADDUI R1, R2, #8} \]
  - Tree height reduction
    \[
    \begin{align*}
    \text{ADD R1, R2, R3} & \rightarrow \text{ADD R4, R1, R6} \\
    \text{ADD R8, R4, R7} & \rightarrow \text{ADD R8, R1, R4}
    \end{align*}
    \]

Software Pipelining

- Combining instructions from different loop iterations to separate dependent instructions within an iteration

\[
\begin{align*}
\text{sum} &= \text{sum} + x \quad \text{/* suppose this is in a loop and we unroll it 5 times */} \\
\text{sum} &= (\text{sum} + x_1) + (x_2 + x_3) + (x_4 + x_5)
\end{align*}
\]

Can be done in parallel

Software Pipelining

- Apply software pipelining technique to the following loop:

\[
\begin{align*}
\text{L.D F0, 0(R1)} & \rightarrow \text{ADD.D F4, F0, F2} \\
\text{S.D F4, 0(R1)} & \rightarrow \text{DADDUI R1, R1, #8} \\
\text{BNE R1, R2, Loop} & \rightarrow \\
\text{L.D F0, 0(R1)} & \rightarrow \text{ADD.D F4, F0, F2} \\
\text{S.D F4, 0(R1)} & \rightarrow \text{DADDUI R1, R1, #8} \\
\text{BNE R1, R2, Loop} & \rightarrow \\
\text{L.D F0, 0(R1)} & \rightarrow \text{ADD.D F4, F0, F2} \\
\text{S.D F4, 0(R1)} & \rightarrow \text{DADDUI R1, R1, #8} \\
\text{BNE R1, R2, Loop} & \rightarrow \\
\text{L.D F0, 0(R1)} & \rightarrow \text{ADD.D F4, F0, F2} \\
\text{S.D F4, 0(R1)} & \rightarrow \text{DADDUI R1, R1, #8} \\
\text{BNE R1, R2, Loop} & \rightarrow \\
\text{L.D F0, 0(R1)} & \rightarrow \text{ADD.D F4, F0, F2} \\
\text{S.D F4, 0(R1)} & \rightarrow \text{DADDUI R1, R1, #8} \\
\text{BNE R1, R2, Loop} & \rightarrow
\end{align*}
\]

⇒

⇒

Startup code

Cleanup code

Software Pipelining vs. Loop Unrolling

- Loop unrolling eliminates loop maintenance overhead exposing parallelism between iterations
  - Creates larger code
  - Software pipelining enables some loop iterations to run at top speed by eliminating RAW hazards that create latencies within iteration
  - Requires more complex transformations

Memory Hierarchy

- We would like to have a lot of fast memory, but smaller memory units are faster
  - Build a hierarchy of memories, each of which has greater capacity than the preceding but is slower
  - Frequently application accesses to data exhibit spatial and temporal locality yielding a small subset that is frequently accessed
  - Smaller memories have subset of data from larger memories

\[
\begin{align*}
\text{CPU registers} & \quad \text{size} = 500B \quad \text{speed} = 0.25 \text{ ns} \\
\text{Cache memory} & \quad \text{size} = 64KB \quad \text{speed} = 1 \text{ ns} \\
\text{Main memory} & \quad \text{size} = 512MB \quad \text{speed} = 100 \text{ ns} \\
\text{Virtual memory} & \quad \text{size} = 100GB \quad \text{speed} = 5 \text{ ms}
\end{align*}
\]

Cache Memory

- When data is found in the cache we have a cache hit
  - We will stall CPU
  - We will fetch data from the main memory and place it in the cache
    - The main memory and the virtual memory have the same relationship as the cache and the main memory.
- When data is not found in the cache we have a cache miss
  - We will fetch a fixed-size block of data
  - We store this block and a part of its main-memory address in cache
  - If data is not in the main memory it may be in the virtual memory — stored on the disk
Cache Performance

CPU time + CC * CPI becomes CPU time + CC * (IC * CPI + Memory stall cycles) where Memory stall cycles = IC * Memory references * Miss rate * Miss penalty

Miss rate tells us how frequently we miss when we have a memory reference. Sometimes, we may use miss rate per instruction. This number tells us how frequently we miss for each instruction (memory reference or not).

Miss_rate_per_instruction = #Misses / #Instructions

Example

We have a computer with CPI = 1 when all memory accesses hit in cache. The only data accesses are loads and stores and these total 50% of the instructions. If the miss penalty is 25 clock cycles and the miss rate is 2% how much faster would the computer be if all instructions resulted in a cache hit.

What if the miss rate is 30 misses per 1000 instructions?

see page 395 for solution

Where Can a Block Be Placed?

- At a fixed place in the cache – direct mapping
  Index = BlockAddress mod CacheSize
- Anywhere in the cache – fully associative
- Anywhere within a restricted set of places – set associative
  SetIndex = BlockAddress mod NumberOfSets
  e.g. if there are 4 blocks in a set, we call the cache 4-way set associative

Replacement Strategy

- When a miss occurs we must replace some block from the cache (if the cache is full) with a new block from the memory
  * Random
  * Least-recently used
  * FIFO (round-robin)

Write Strategy

- Block reads are optimized so that they start in parallel with reading and checking tag
  * If this is not the block we want, read data is simply discarded
- Block writes cannot start in parallel with tag checking
- When we modify a block in cache we must decide when to update the memory
  * Write-through – always update the memory
  * Write-back – update the memory only when we delete the block from the cache. We add a dirty bit to the block, this bit is modified when block write occurs and checked on write-back
  * We can write into write buffer and release processor
  * On write miss we can update the cache (write allocate) but we don’t have to (no write allocate)

Example

Assume a fully associative write-back cache with many cache entries that starts empty and performs operations:

Write Mem[100];
Write Mem[100];
Read Mem[200];
Write Mem[200];
Write Mem[100];

What are numbers of hits and misses when using no-write allocate vs. write allocate?

see page 402 for solution
Instructions and Data Caches

- We can keep the data and instructions in the same cache – **unified cache**
- Or we can keep them in separate caches – **split cache**
  - This improves hit rate as instructions and data are accessed in a different manner

Cache Performance

To realistically evaluate cache performance we need to look beyond miss rate:

\[
\text{Average Memory Access Time} = \text{Hit Time} + \text{Miss Rate} \times \text{Miss Penalty}
\]

Example

Which has the lower miss rate: a 16KB instruction cache with a 16KB data cache or a unified 32KB cache? Use the following data for number of misses per 1000 instructions:

<table>
<thead>
<tr>
<th>size</th>
<th>instruction cache</th>
<th>data cache</th>
<th>unified cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 KB</td>
<td>3.82</td>
<td>40.9</td>
<td>51</td>
</tr>
<tr>
<td>32 KB</td>
<td>1.36</td>
<td>38.4</td>
<td>43.3</td>
</tr>
</tbody>
</table>

Assume 36% of instructions are data transfer instructions, a hit takes 1 clock cycle and penalty is 100 clock cycles. A load or store takes 1 extra clock cycle on a unified cache if there is only one port. What is the average memory access time for both cases? Assume write-through caches with write buffer.

Example

Assume cache miss penalty is 100 cycles and all instructions Normally take 1 clock cycle. Average miss rate is 2%, there is an average of 1.5 memory references per instruction, and the average number of cache misses per 1000 instructions is 30. Calculate CPU time both with average miss rate per memory reference and average miss rate per instruction.

What if we had no cache?

Example

What is the impact of two different cache organizations on the performance of a CPU? Assume that CPI for a perfect cache is 2, clock cycle time is 1ns, there are 1.5 memory references per instruction, size of both caches is 64KB and block size is 64B. One cache is direct mapped and the other is 2-way set associative. For a set associative cache assume that clock cycle is 1.25 ns. Cache miss penalty is 75ns for both caches. Calculate memory access time and then CPU time. Assume that hit time is 1 clock cycle, the miss rate of direct mapped cache is 1.4% and the miss rate of set associative cache is 1%.

Example

When processor supports out-of-order execution, some part of the miss penalty can be overlapped

- It is a bit tricky to define miss rate and miss penalty
- We say that processor is stalled whenever it cannot commit maximum number of instructions
- First instruction that is causing the stall is regarded as a reason for the stall
- If this is a memory reference waiting because of the cache miss we will count this stall towards miss rate
Example

Assume that CPI for a perfect cache is 2, clock cycle time is 1.25ns, there are 1.5 memory references per instruction. Cache is direct mapped and supports out-of-order execution. Cache miss penalty is 75ns but 30% of that can be overlapped. The miss rate is 1.4%. How much is CPU time? see page 411 for solution