G22.2243-001
High Performance Computer Architecture

Lecture 7
Compiling for VLIW/EPIC Processors
Memory System

March 1, 2006

Outline

• Announcements
  – Final Exam: Wednesday, May 3 5:00 - 6:50pm
  – Lab Assignment 2 due back today; deadline extended to next week
  – HW Assignment 3 out today. Due next week: March 8
• Last lecture:
  – Tomasulo’s algorithm
  – Multiple-issue processors (achieving IPC > 1)
• Superscalar processors
• Brief mention of VLIW processors
• VLIW processors
  – Software techniques
  – Hardware support
• Memory System

[ Hennessy/Patterson CA:AQA (3rd Edition): parts of Chapter 4, Chapter 5 ]

Architectural Features in VLIW Processors

• VLIW processors rely on the compiler to identify a packet of instructions that can be issued in the same cycle
  – Compiler takes responsibility for scheduling instructions so that their dependences are satisfied

\[ r1 = L r4 \]
\[ r2 = Add r1, M \]
\[ f1 = Mul f1, f2 \]
\[ r5 = Add r5, 4 \]

• Optimizations such as loop unrolling, and software pipelining expose more ILP, allowing the compiler to build issue packets
• Architectural support helps compiler expose/exploit more ILP

Basic Compiler Techniques (S1): Loop Unrolling

(Recap)

• Consider the example from last week:

```c
for (i=1000; i>0; i--)
  x[i] = x[i] + s
```

```c
L1: L.D F0, 0(R1)
ADD.D F4, F0, F2
S.D F4, 0(R1)
DADDUI R1, R1, #-8
BNE R1, R2, L1
```

• Loop unrolling optimization: Replicate loop body multiple times, adjusting the loop termination code

```c
L1: 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, L1
```

• Loop unrolling optimization: Replicate loop body multiple times, adjusting the loop termination code

```
L1: 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, L1
```

Basic Compiler Techniques: Loop Unrolling (cont’d)

• Unroll loop 5 times

```
L1: 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, L1
```

Basic Compiler Techniques: Loop Unrolling (cont’d)

```
L1: 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, L1
```

Provide instructions for VLIW
Hardware Support for VLIW

- To expose more parallelism at compile time
  - Conditional or predicated instructions
  - Predication registers in IA64
  - Allow the compiler to group instructions across branches
- To allow compiler to speculate, while ensuring program correctness
  - Result of speculated instruction will not be used in final computation if mispredicted
  - Speculative movement of instructions (before branches, reordering of loads/stores) must not cause exceptions
  - HW allows exceptions from speculative instructions to be ignored
    - Poison bits and Reorder Buffers
  - HW tracks memory dependences between loads and stores
    - LDS (speculative load) and LDV (load verify) instructions
    - Check for intervening stores
  - Variant: LDV instruction can point to fix-up code

HW Support for Speculative Operations (H1)

- Speculative operations in HPL-PD architecture from HP Labs written identically to their non-speculative counterparts, but with an "E" appended to the operation name.
  - E.g., DIVE, ADDE, FBBRE
- Poison bits: If an exceptional condition occurs during a speculative operation, the exception is not raised
  - A bit is set in the result register to indicate that such a condition occurred
  - Speculative bits are simply propagated by speculative instructions
  - When a non-speculative operation encounters a register with the speculative bit set, an exception is raised

(H1) Compiler Use of Speculative Operations

- Here is an optimization that uses speculative instructions:
  - Also the effect of the DIV latency is reduced
  - If a divide-by-zero occurs, an exception will be raised by ADD

HW Support for Predication (H2)

- Conditional or predicated instructions
  - Instruction is "conditionally" executed, else no-op
  - Originally: a separate set of (simple) instructions
  - Now: more general support
- In HPL-PD, most operations can be predicated
  - they can have an extra operand that is a one-bit predicate register.
  - If the predicate register contains 0, the operation is not performed
  - The values of predicate registers are typically set by "compare-to-predicate" operations
  - Also the effect of the ADD latency is reduced

Compiler Uses of Predication

- if-conversion
  - To aid code motion by instruction scheduler
    - e.g. hyperblocks

Uses of Predication: If-conversion

- If-conversion replaces conditional branches with predicated operations
- For example, the code generated for:
  ```c
  if (a < b)
    c = a;
  else
    c = b;
  if (d < e)
    f = d;
  else
    f = a;
  ```
  might be the two VLIW instructions:
  ```c
  p1 = CMPP.< a,b p2 = CMPP.< d,e p3 = CMPP.< a,b p4 = CMPP.< d,e
  c = a if p1 c = b if p2 f = d if p3 f = a if p4
  ```
Compare-to-predicate instructions

- In previous slide, there were two pairs of almost identical instructions just computing complement of each other
- HPL-PD provides two-output CMPP instructions

\[ p_{1,2} = \text{CMPP.W.<.UN.UC} \ r_{1,2} \]

(H2) If-conversion, revisited

- Using two-output CMPP instructions, the code generated for:
  - if \( a < b \)
    - \( c = a \)
    - else
    - \( d = b \)
    - if \( d < e \)
    - \( f = d \)
    - else
    - \( f = e \)
  - might instead be:

\[ p_{1,2} = \text{CMPP.W.<.UN.UC} \ a,b \]
\[ p_{3,4} = \text{CMPP.W.<.UN.UC} \ d,e \]
\[ c = a \ \text{if} \ p_{1} \]
\[ c = b \ \text{if} \ p_{2} \]
\[ f = d \ \text{if} \ p_{3} \]
\[ f = e \ \text{if} \ p_{4} \]

Uses of Predication: Hyperblock Formation

- In hyperblock formation, if-conversion is used to form larger blocks of operations than the usual basic blocks:
  - tail duplication used to remove some incoming edges in middle of block
  - if-conversion applied after tail duplication
  - larger blocks greater opportunity for code motion to increase ILP

HW Support for Memory Disambiguation (H3)

- Here’s a desirable optimization (due to long load latencies):
  - However, this optimization is not valid if the load and store reference the same location
    - i.e., if \( r_{2} \) and \( r_{3} \) contain the same address
    - this cannot be determined at compile time
  - HPL-PD solves this by providing run-time memory disambiguation

(H3) HW Support for Memory Disambiguation (cont’d)

HPL-PD provides two special instructions to replace a load instruction:
  - \( r_{1} = \text{LDS} \ r_{2} \): speculative load
    - Initiates a load like a normal load instruction
    - A log entry can made in a table to store the memory location
  - \( r_{1} = \text{LDV} \ r_{2} \): load verify
    - Checks to see if store to memory location has occurred since the LDS
    - If so, the new load is issued and the pipeline stalls. Otherwise, it’s a no-op

The previous optimization becomes

\[ \ldots \]
\[ r_{1} = \text{LDS} \ r_{2} \]
\[ r_{1} = \text{ADD} \ r_{1},7 \]
\[ \ldots \]

More Sophisticated Compiler Optimizations: Software Pipelining (S2)

- Software Pipelining is the technique of scheduling instructions across several iterations of a loop
  - reduces pipeline stalls on sequential pipelined machines
  - exploits instruction level parallelism on superscalar and VLIW machines
  - intuitively, iterations are overlaid so that an iteration starts before the previous iteration has completed
(S2) Software Pipelining Example

• Source code:
  `for(i=0;i<n;i++) sum += a[i]`

• Loop body in assembly:
  ```
  r1 = L r0
  --- ;stall
  r2 = Add r2, r1
  r0 = Add r0, 12
  r4 = L r3
  --- ;stall
  r2 = Add r2, r4
  r3 = Add r3, 12
  r7 = L r6
  --- ;stall
  r2 = Add r2, r7
  r6 = Add r6, 12
  r0 = Add r0, 12
  r10 = L r9
  --- ;stall
  r2 = Add r2, r10
  r9 = Add r9, 12
  ```

• Unroll loop and allocate registers

(S2) Software Pipelining Example (cont’d)

• Schedule unrolled Instructions, exploiting VLIW (or not)

(S2) Software Pipelining Example (cont’d)

• Schedule unrolled Instructions, exploiting VLIW (or not)

(S2) Software Pipelining Example (cont’d)

• Schedule unrolled Instructions, exploiting VLIW (or not)

(S2) Software Pipelining Example (cont’d)

• Schedule unrolled Instructions, exploiting VLIW (or not)

(S2) Software Pipelining Example (cont’d)

• Schedule unrolled Instructions, exploiting VLIW (or not)

Constraints on Software Pipelining

The instruction-level parallelism in a software pipeline is limited by
• Resource Constraints
  – VLIW instruction width, functional units, bus conflicts, etc.
• Dependence Constraints
  – particularly loop carried dependences between iterations
  – arise when
    • the same register is used across several iterations
    • the same memory location is used across several iterations

Dynamic Memory Aliasing

• What if the code were:
  ```
  for(i=A;i<n;i++)
  a[i] = a[i-k] + c;
  ```
  where k is unknown at compile time?
  ```
  • k = 0 (no dependence), k > 0 (true dependence with distance k),
    k < 0  (anti-dependence with distance |k|)
  • The worst case is k = 1
• What can the compiler do?
  – Assume the worst, and generate the most pessimistic pipelined schedule
  – Generate different versions of the software pipeline for different distances
  – branch to the appropriate version at run-time
  – possible code explosion, cost of branch
Summary: VLIW Processors

- Architectural features enable aggressive compiler optimizations
  - To pack multiple instructions per VLIW packet
  - Loop unrolling and software pipelining
- Hardware support
  - Speculative instructions
  - Conditional/Predicated instructions
  - Run-time memory disambiguation
  - Hardware support for preserving exception behavior
    - Poison bits, reorder buffer
- Limiting factors
  - Increased code size: requires aggressive unrolling; not full instructions
  - VLIW lock step ⇒ 1 hazard and all instructions stall
  - Binary code compatibility is practical weakness

Memory Hierarchy Design
(Moving Outside the Processor)

Why Worry About the Memory Hierarchy?

- The course to this point has focused on processor performance issues
  - CPU cost/performance, ISA, Pipelined and dynamic execution

\[
\text{CPU time} = (\text{CPU execution clock cycles} + \text{Memory stall clock cycles}) \times \text{clock cycle time}
\]

\[
\text{Memory stall clock cycles} = (\text{Reads} \times \text{Read miss rate} \times \text{Read miss penalty}) + (\text{Writes} \times \text{Write miss rate} \times \text{Write miss penalty})
\]

\[
\text{CPU-DRAM performance gap} = 60\% \text{ per year}
\]

Processor-Memory Performance Gap “Tax”

- Fraction of processor area/transistors taken up by caches (~1997)

<table>
<thead>
<tr>
<th>Processor</th>
<th>% Area (cost)</th>
<th>% Transistors (power)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha 21164</td>
<td>37%</td>
<td>77%</td>
</tr>
<tr>
<td>StrongArm SA110</td>
<td>61%</td>
<td>94%</td>
</tr>
<tr>
<td>Pentium Pro</td>
<td>64%</td>
<td>88%</td>
</tr>
</tbody>
</table>

- Caches have no inherent value, only try to close performance gap

(Review) Cache Organization

- Cache is the name given to the first level of the memory hierarchy, encountered once the address leaves the CPU
  - It serves as a temporary place where frequently-used values can be stored
    - Retains the same name as in memory (different from registers)
    - To avoid having to go to memory every time this value is needed
  - Caches are faster (hence more expensive, limited in size) than DRAM
  - Caches store values at the granularity of cache blocks (lines)
    - Larger than a single word: efficiency and spatial locality concerns
    - Cache hit if value in cache, else cache miss
  - Effect of caches on CPU execution time

Four Questions for Memory Hierarchy Designers

Q1: Where can a block be placed in the upper level?
  (Block placement)
  - Fully Associative, Set Associative, Direct Mapped
Q2: How is a block found if it is in the upper level?
  (Block identification)
  - Tag per block
Q3: Which block should be replaced on a miss?
  (Block replacement)
  - Random, LRU
Q4: What happens on a write?
  (Write strategy)
  - Write Back or Write Through (with Write Buffer)
Question 1: Block Placement

- Fully associative: block can be placed anywhere
- Direct map: each block has one place
- Set associative: block can be placed anywhere in a set

Range of caches is really a continuum of levels of set associativity

Most caches today are direct-mapped (1-way), 2-way or 4-way associative

Question 2: Block Identification

- Caches have a tag on each block frame that gives the block address
- All possible tags, where the block may be present, are checked in parallel
- Quick check of whether a block contains data: Valid bit
- Organization determines which (subset of) blocks need to be checked

Selects the "set"
Selects "block" within set

Larger blocks
Fully-associative caches: Only tag

Question 3: Block Replacement

- When a new block needs to be brought in (on demand), an existing cache block may need to be freed up
- Three commonly-used schemes
  - Random: Easiest to implement
  - Least-recently used (LRU): used as an approximation to LRU
- LRU outperforms Random and FIFO on smaller caches
  - FIFO outperforms Random
- Differences not as big for larger caches
  - Bigger benefit from avoiding misses in the first place

Question 4: Write Strategy

- When is memory updated with the contents of a store?
- Issue: Reads dominate cache traffic (writes typically 10% of accesses)
  - Optimization for read: Do tag checking and data transfer in parallel
  - Cannot do this for writes (also, only sub-portion of block needs update)
- Two write policies
  - Write through
    - Information written to both cache and memory
  - Least-recently used (LRU): used as an approximation to LRU
  - Also, simplifies data coherency (later in the course)
  - Write back
    - Information only written to the cache
    - Dirty bit keeps track of which blocks have data that needs to be sync-ed
    - Multiple writes lead to less number of writes to memory
    - Reduces memory bandwidth requirement (hence power)
    - Variants: With or without write-allocate (usually used with write back)
    - Write stalls in write-through caches reduced using write buffers

The Alpha 21264 Data Cache

- 64KB cache, 64B blocks
- 2-way set associative, write-back, write allocate
- 44-bit physical address
  - 9-bit index
    - Identifies 2 blocks from 512 sets
  - 29-bit tag
    - Identifies which of 2 blocks
- Tag checking and data extraction proceed in parallel

Figure shows steps involved in a "read hit"

Improving Cache Performance

$$\text{CPU time} = (\text{CPU access clock cycles} \times \text{Memory clock cycles}) \times \text{clock cycle time}$$

$$\text{Memory stall clock cycles} =$$

$$(\text{Reads} \times \text{Read miss rate} \times \text{Read miss penalty}) +$$

$$(\text{Writes} \times \text{Write miss rate} \times \text{Write miss penalty}) +$$

$$\text{Memory accesses} \times \text{Miss rate} \times \text{Miss penalty}$$

- Above assumes 1-cycle to hit in cache
  - Hard to achieve in current-day processors (faster clocks, larger caches)
  - More reasonable to also include hit time in the performance equation

Average memory access time = Hit Time + Miss rate x Miss Penalty

Small/simple caches
Avoiding address translation
Pipelined cache access
Large block size
Larger cache size
Higher associativity
Way prediction
Critical word first
Compiler optimizations
More aggressive coherency
Merging write buffers
Victim caches
Nonblocking caches
Hardware prefetching
Compiler prefetching
Multilevel caches
Cardinality
Critical word first
Read misses before write
Merging write buffers
Victim caches
Compiler optimizations

1. Small/simple caches
2. Avoiding address translation
3. Pipelined cache access
4. Large block size
5. Larger cache size
6. Higher associativity
7. Way prediction
8. Critical word first
9. Compiler optimizations
10. More aggressive coherency
11. Merging write buffers
12. Victim caches
13. Nonblocking caches
14. Hardware prefetching
15. Compiler prefetching
16. Multilevel caches
A.1. Reducing Miss Penalty via Multilevel Caches

- **Idea**: Have multiple levels of caches
  - Tradeoff between size (cache effectiveness) and cost (access time)
- **For a 2-level cache**
  - Average memory access time = Hit time (L1) + Miss rate (L1) x Miss penalty (L1)
  - Miss penalty (L1) = Hit time (L2) + Miss rate (L2) x Miss penalty (L2)
- **Distinguish between two kinds of miss rates**
  - Local miss rate = Miss rate (L1) or Miss rate (L2)
  - Global miss rate = Number of misses/total number of memory accesses
    = Miss rate (L1), but Miss rate (L1) x Miss rate (L2)
- **Example**: 1000 references, 40 misses in L1 cache and 20 in L2
  - Local miss rates: 4% (L1), 50% (L2) = 20/40
  - Global miss rates: 4% (L1), 2% (L2)
  - Avg. memory access time = 1 + 4% x (10 + 50% x 100) = 3.4 cycles

A.2. Reduce Miss Penalty via Critical Word First and Early Restart

- **Idea**: Don’t wait for full block to be loaded before restarting CPU
  - Early restart: request the words in a block in order. As soon as the requested word of the block arrives, send it to the CPU and let the CPU continue execution
  - Critical Word First: Request the missed word first from memory and send it to the CPU as soon as it arrives; let the CPU continue execution while filling the rest of the words in the block
    - Also called wrapped fetch and requested word first
- **Drawbacks**
  - Generally useful only in large blocks
  - Programs exhibiting spatial locality a problem; tend to want next sequential word, so limited benefit by early restart

A.3. Reducing Miss Penalty by giving Reads Priority over Writes on Misses

- **Solution**: Give read misses priority
- **Challenges**
  - Write-through with write buffers may result in RAW conflicts
    - Solution 1: Wait for write buffer to empty (not great)
    - Solution 2: Check write buffer contents before read; if no conflicts, let the memory access continue
  - Write-back caches: Read miss may require replacing a dirty block
    - Normal: Write dirty block to memory, and then do the read
    - Better alternative: Copy the dirty block to a write buffer, then do the read, and then do the write
    - CPU stall less since restarts as soon as read is done

A.4. Reducing Miss Penalty using Merging Write Buffers

- **Normal mode of operation of a write buffer**
  - Absorb write from CPU, commit it to memory in the background
- **Problem** (particularly in write-through caches)
  - Small write-buffers may end up stalling processor if they fill up
  - Processor needs to wait till write committed to memory
- **Solution**: Merge cache-block entries in the write buffer
  - Multiword writes are usually faster than writes performed one at a time
  - Writes usually modify one word in a block; If a write buffer already contains some words from the given data block we will merge current modified word with the block parts already in the buffer

A.5. Reducing Miss Penalty via a “Victim Cache”

- **How to combine the fast hit time of direct-mapped caches, yet still avoid conflict misses?**
- **Remember what was recently discarded, just in case it is needed again**
  - Jouppi [1990]: 4-entry victim cache reduced conflict misses by 20% - 95% for a 4 KB direct mapped data cache
  - Used in Alpha, HP machines
B. Reducing Cache Misses

Classifying Misses: 3 Cs

• Compulsory (Also called cold start or first reference misses)
  – The first access to a block is not in the cache, so the block must be
  brought into the cache.
  (Misses in even an Infinite Cache)

• Capacity
  – The cache may not contain all blocks needed during program execution,
  so misses will occur due to blocks being discarded and later retrieved
  (Misses in Fully Associative Size X Cache)

• Conflict (Also called collision or interference misses)
  – Additional misses that occur because another block is occupying cache
  (the rest of the cache might be unused)
  (Misses in N-way Associative, Size X Cache)

How Can We Reduce Misses?

• 3 Cs: Compulsory, Capacity, Conflict

  1. Change block size
     Which of 3Cs is obviously affected?

  2. Change associativity
     Which of 3Cs is obviously affected?

  3. Change compiler
     Which of 3Cs is obviously affected?

B.1. Reducing Miss Rate via Larger Block Sizes

- Small blocks: Data accesses spread over multiple blocks
- Large blocks: Not all the data is useful, but displaces useful data
- Also note larger blocks mean higher miss penalty

B.2. Reducing Miss Rate via Higher Associativity

• 2:1 Cache Rule
  – Miss Rate of a direct-mapped cache size of size N ~
  Miss Rate of a 2-way cache of size N/2

  1. Change block size
     Which of 3Cs is obviously affected?

  2. Change associativity
     Which of 3Cs is obviously affected?

  3. Change compiler
     Which of 3Cs is obviously affected?

How Can We Reduce Misses?

• 3 Cs: Compulsory, Capacity, Conflict

  1. Change block size
     Which of 3Cs is obviously affected?

  2. Change associativity
     Which of 3Cs is obviously affected?

  3. Change compiler
     Which of 3Cs is obviously affected?
B.3. Reducing Miss Rate via Way Prediction and Pseudoassociativity

- How to combine fast hit time of direct-mapped caches with the lower conflict misses of set-associative caches?
  - Previously looked at Victim Caches

- Way prediction: Predict which block in a set is likely to be accessed by the next memory access hitting this set
  - Tag comparison only with this block (cheaper as opposed to with all)
  - Simplest prediction: remember the last word accessed
  - Used in Alpha 21264 (1-cycle if correct prediction (85%), 3-cycles o.w.)

- Pseudoassociative or Column associative
  - Access proceeds as in direct-mapped cache
  - On a miss, check another location ("pseudoset") before going to memory
    - Counts as a "slower hit"
    - If most hits become slow hits, degrading performance is possible
  - Used in MIPS R10000 L2 cache, similar in UltraSPARC

B.4. Reducing Miss Rate by Compiler Optimizations

- Compiler optimizations can help reduce both instruction and data cache misses (for a fixed cache organization)

  - Instruction misses
    - Reorder procedures in memory so as to reduce conflict misses
    - Conflicts determined by profiling
    - Reduced l-cache misses by 75% in an 8KB cache (McFarling 1989)
    - Cache-line alignment of basic blocks
      - Decreases likelihood of cache miss on sequential code

  - Data misses
    - Several optimizations that reorder data access patterns
    - Two examples
      - Loop interchange
      - Blocking

Loop Interchange Example

```c
/* Before */
for (k = 0; k < 100; k = k+1)
  for (j = 0; j < 100; j = j+1)
    for (i = 0; i < 5000; i = i+1)
      x[i][j] = 2 * x[i][j];

/* After */
for (k = 0; k < 100; k = k+1)
  for (i = 0; i < 5000; i = i+1)
    for (j = 0; j < 100; j = j+1)
      x[i][j] = 2 * x[i][j];
```

- "After" version accesses memory sequentially instead of in strides of 100 words
- Improved spatial locality: use all of the words in fetched blocks

Blocking Example

```c
/* Before */
for (i = 0; i < N; i = i+1)
  for (j = 0; j < N; j = j+1)
    {r = 0;
     for (k = 0; k < N; k = k+1){
       r = r + y[i][k]*z[k][j];}
    x[i][j] = r;
}
```

- Capacity misses depend on N, cache size
- if all three matrices fit and there are no conflict misses, best performance
- if cache can hold one NxN matrix and one row of N elements, then y and z can be in the cache
- else, misses for both y and z
- worst case: 2N^3 + N^2 misses

C. Using Parallelism to Reduce Miss Penalty/Rate

- Idea: Permit multiple “outstanding” memory operations
  - Can overlap memory access latencies
  - Can benefit from activity done on behalf of other operations

- Three commonly-employed schemes
  - Non-blocking caches
  - Hardware prefetching
  - Software prefetching
C.1. Non-blocking Caches to Reduce Stalls on Misses

- Decoupled instruction and data caches allow CPU to continue fetching instructions while waiting on a data cache miss.
  - L1 cache misses can be tolerated by superscalar out-of-order machines.
- Non-blocking or lockup-free caches allow data cache to continue to supply cache hits during a miss.
  - Requires out-of-order execution CPUs.
- "hit under miss" reduces the effective miss penalty by working during miss vs. ignoring CPU requests.
- "hit under multiple miss" or "miss under miss" may further lower the effective miss penalty by overlapping multiple misses.
  - Significantly increases the complexity of the cache controller as there can be multiple outstanding memory accesses.
  - Typically also requires multiple memory banks.
  - Pentium Pro allows 4 outstanding memory misses.

Value of Hit-Under-Miss for SPEC92

8KB direct-mapped cache, 32B blocks, 16-cycle penalty

Floating-point

- 76%
- 78%
- 79%

D. Reducing Cache Hit Time

- Obvious approach: Smaller and simpler (low associativity) caches.
  - Notable that L1 cache sizes have not increased.
  - Alpha 21264/21364; UltraSPARC II/III; AMD K6/Athlon.
- Other techniques:
  - Avoiding address translation during cache lookup.
  - Alternative 1: Index caches using virtual addresses.
    - Needs to cope with several problems:
      - Protection (performed during address translation).
      - Reuse of virtual addresses across processes (flushing cache after context switch).
      - Major/minor: Two processors refer to the same physical address (results in having multiple copies of the same data).
      - 10% typically uses physical addresses.
    - Alternative 2: Use page offset to index the cache.
      - Does not change between virtual and physical addresses.

C.2. Reducing Misses by Hardware Prefetching of Instructions & Data

- Instruction Prefetching:
  - Alpha 21064 fetches 2 blocks (requested and subsequent) on a miss.
  - Extra block in "stream buffer".
  - On miss, check stream buffer.
- Works with data blocks too:
  - Hardware identifies stream of accesses and then prefetches them.
  - Can compute stride by comparing current and previous access.
  - UltraSPARC III supports up to 8 simultaneous prefetches.
- Prefetching relies on having extra memory bandwidth that can be used without penalty.
- How well does this work?
  - Jouppi (1990):
    - 1-block stream buffer catches 15-25% of misses, 4-block stream buffer: 50%, 16-block stream buffer: 72%.
  - Palacharla & Kessler (1994):
    - For scientific programs, 8 stream buffers got 50% to 70% of misses from a system with 2 64KB, 4-way set associative caches (one for instructions one for data).

C.3. Reducing Misses by Software Prefetching of Data

- Compiler can insert special instructions to request prefetching.
- Two variants:
  - Load data into register (HP PA-RISC loads).
  - Load data into cache (MIPS IV, PowerPC, SPARC v. 9).
- Issues:
  - Special prefetching instructions typically cannot cause faults (a form of speculative execution: non-faulting vs. faulting).
  - Processor must be able to proceed while prefetched data is being fetched to make this approach valuable.
  - i.e., non-blocking data caches.
  - Issuing the prefetch instructions takes time.
  - Is cost of prefetch issues < savings in reduced misses?
  - Higher superscalar reduces difficulty of issue bandwidth.

D.1. Virtually Indexed, Physically Tagged Caches

- Overlap indexing of cache with translation of virtual addresses:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>110</td>
<td>1 &lt;/br&gt; 132</td>
</tr>
</tbody>
</table>

- Direct-mapped caches can be no bigger than page size.
- Set-associative caches:
  - Page offset can be viewed as (Index + block offset) above.
  - Cache size = 4 way x Set associativity.
  - So, increased associativity allows larger cache sizes:
    - Pentium III (4KB page): 2-way set-associative 16 KB cache.
    - IBM 3033 (8KB page): 16-way set-associative 64 KB cache.
D.2. Trace Caches

- A challenge in multiple-issue processors is to supply enough instructions every cycle without dependencies
  - Challenge: fetching across branches
  - Cache impact is significant with large cache blocks
- Option 1: Combine branch prediction with instruction prefetching
  - Instructions stored according to memory addresses
- Option 2: A separate cache that stores and provides a dynamic sequence of instructions including taken branches (Trace Cache)
  - Pros
    - Effective use of cache blocks: no wasted words, no conflicts, …
  - Cons
    - Complicated address mapping mechanisms
    - Same instruction may be stored multiple times
  - Used in the Intel NetBurst microarchitecture (Pentium 4)

---

### Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>MP</th>
<th>MR</th>
<th>HT</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multilevel caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Early Restart &amp; Critical Word 1st</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Priority to Read Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Merging write buffer</td>
<td>+</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Victim Caches</td>
<td>+</td>
<td>+</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Larger Block Size</td>
<td>-</td>
<td>+</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Higher Associativity</td>
<td>+</td>
<td>-</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Pseudo-Associative Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Compiler Reduce Misses</td>
<td>+</td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>Non-Blocking Caches</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>HW Prefetching of Inst/Data</td>
<td>+</td>
<td>+</td>
<td></td>
<td>2/3</td>
</tr>
<tr>
<td>Compiler Controlled Prefetching</td>
<td>+</td>
<td></td>
<td>+</td>
<td>3</td>
</tr>
<tr>
<td>Avoiding Address Translation</td>
<td>+</td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>Trace Cache</td>
<td>+</td>
<td></td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>