MEMORY HIERARCHY
DESIGN
B649
Parallel Architectures and Programming
Basic Optimizations

Average memory access time = Hit time + Miss rate \times Miss penalty

- Larger block size to reduce miss rate
- Larger caches to reduce miss rate
- Higher associativity to reduce miss rate
- Multilevel caches to reduce miss penalty
- Prioritizing read misses over writes to reduce miss penalty
- Avoiding address translation during indexing of the cache to reduce hit time
ADVANCED OPTIMIZATIONS
Eleven Advanced Optimizations

Average memory access time = Hit time + Miss rate × Miss penalty

- Reducing the hit time
  - small and simple caches, way prediction, trace caches
- Increasing cache bandwidth
  - pipelined caches, multibanked caches, non-blocking caches
- Reducing the miss penalty
  - critical word first, merging write buffers
- Reducing the miss rate
  - compiler optimizations
- Reducing miss penalty / miss rate via parallelism
  - hardware prefetching, compiler prefetching
#1: Small and Simple Caches
*(To Reduce Hit Time)*

• Small caches can be faster
  ★ reading tags and comparing is time-consuming
  ★ L1 should be fast enough to be read in 1-2 cycles
  ★ desirable to keep L2 small enough to fit on chip
    ★ could keep data off-chip and tags on-chip

• Simpler caches can be faster
  ★ direct-mapped caches: can overlap tag check and transmission of data
    ★ Why is this not possible with set-associative caches?
Access Times on a CMOS Cache (CACTI)
#2: Way Prediction
*(To Reduce Hit Time)*

- Extra bits per block to predict the *way* (block within the set) of the next cache access
  - can set the multiplexer early
  - can match the tag and read data in parallel
  - miss results in matching other blocks in next clock cycle
- Prediction accuracy > 85% suggested by simulations
  - good match for speculative processors
  - used in Pentium 4
#3: Trace Caches
(To Reduce Hit Time)


If \( A[i] = 0 \)

\[ B[i] = \]

Else

\[ C[i] = X \]
#3: Trace Caches

*(To Reduce Hit Time)*

- **Goal:** to enhance instruction-level parallelism (find sufficient number of instructions without dependencies)
  - trace = dynamic sequence of executed instructions

- **Using traces**
  - branches folded into traces, hence need to be validated
  - more complicated address mapping (?)
  - better utilize long blocks
  - conditional branches cause duplication of instructions across traces
  - used in Pentium 4 (in general, benefits not obvious)
#4: Pipelined Cache Access

*(To Increase Cache Bandwidth)*

- Pipeline results in fast clock cycle time and high bandwidth, but slow hits
  - Pentium 1: 1 clock cycle for instruction cache
  - Pentium Pro / III: 2 clock cycles
  - Pentium 4: 4 clock cycles
#5: Nonblocking Caches
(To Increase Cache Bandwidth)

• *Nonblocking* or *lockup-free* cache increases the potential benefit of out-of-order processors by continuing to serve hits while a miss is outstanding
  ★ called *hit-under-miss* optimization

• Further optimization if multiple outstanding misses allowed
  ★ *hit-under-multiple-miss* or *miss-under-miss* optimization
  ★ useful only if memory system can serve multiple misses
  ★ recall that outstanding misses can limit achievable ILP

• In general, L1 misses possible to hide, but L2 misses extremely difficult to hide
#5: Nonblocking Caches
*(To Increase Cache Bandwidth)*

Ratio of average memory stall time for a blocking cache to hit-under-miss schemes for SPEC92 programs
#6: Multibanked Caches

(To Increase Cache Bandwidth)

- Originally used for memory, but also applicable to caches
  - L2: Opteron has two banks, Sun Niagara has four banks
- Sequential interleaving works well

<table>
<thead>
<tr>
<th>Block address</th>
<th>Bank 0</th>
<th>Block address</th>
<th>Bank 1</th>
<th>Block address</th>
<th>Bank 2</th>
<th>Block address</th>
<th>Bank 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
<td>10</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>13</td>
<td></td>
<td>14</td>
<td></td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

© 2007 Elsevier, Inc. All rights reserved.
#7: Critical Word First and Early Restart

*(To Reduce Miss Penalty)*

- Observation: cache usually needs one word of the block at a time
  - show impatience!
- Critical word first
  - fetch the missed word from the memory first and sent it to processor as soon as it arrives
- Early restart
  - fetch words in normal order, but send the requested word to the processor as soon as it arrives
- Useful for large block sizes
#8: Merging Write Buffers
*(To Reduce Miss Penalty)*

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>Mem[100]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>1</td>
<td>Mem[108]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116</td>
<td>1</td>
<td>Mem[116]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>124</td>
<td>1</td>
<td>Mem[124]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
#8: Merging Write Buffers

(To Reduce Miss Penalty)

• Write merging
  ★ used in Sun Niagara

• Helps reduce stalls due to write buffers being full

• Uses memory more efficiently
  ★ multi-word writes are faster than writes performed one word at a time

• The block replaced in a cache is called the victim.
  ★ AMD Opteron calls its write buffer victim buffer
  ★ do not confuse with victim cache!
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
- Branch straightening
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

• Reordering procedures to reduce conflict misses
#9: Compiler Optimizations: Code
(To Reduce Miss Rate)

- Reordering procedures to reduce conflict misses
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses

---

**Diagram:**

- Memory
- Direct-mapped cache

---

*B629: Practical Compiling for Modern Machines*
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
#9: Compiler Optimizations: Code (To Reduce Miss Rate)

- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
#9: Compiler Optimizations: Code

*(To Reduce Miss Rate)*

- Reordering procedures to reduce conflict misses
- Aligning basic blocks at cache block boundaries
- Branch straightening
Reordering procedures to reduce conflict misses
Aligning basic blocks at cache block boundaries
Branch straightening
#9: Compiler Optimizations: Data

(To Reduce Miss Rate)

• Loop interchange
  ★ to effectively leverage spatial locality

• Blocking
  ★ to improve temporal locality
#9: Compiler Optimizations: Loop Interchange

*(To Reduce Miss Rate)*

```c
for (j=0; j < 100; j++)
  for (i=0; i < 5000; i++)
    x[i][j] = 2*x[i][j];
```
#9: Compiler Optimizations: Loop Interchange

*(To Reduce Miss Rate)*

```c
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```
#9: Compiler Optimizations: Loop Interchange

*(To Reduce Miss Rate)*

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

Column-major ordering
#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

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

Only one cache miss

Column-major ordering
#9: Compiler Optimizations: Loop Interchange

(To Reduce Miss Rate)

```
for (j=0; j < 100; j++)
    for (i=0; i < 5000; i++)
        x[i][j] = 2*x[i][j];
```
#9: Compiler Optimizations: Loop Interchange

*(To Reduce Miss Rate)*

\[
\begin{align*}
\text{for} & \quad (j=0; \ j < 100; \ j++) \\
\text{for} & \quad (i=0; \ i < 5000; \ i++) \\
x[i][j] & = 2 \times x[i][j];
\end{align*}
\]

Row-major ordering
#9: Compiler Optimizations: Loop Interchange

*(To Reduce Miss Rate)*

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

Row-major ordering
#9: Compiler Optimizations: Loop Interchange
*(To Reduce Miss Rate)*

\[
\begin{align*}
\text{for } (i=0; i < 5000; i++) \\
\text{for } (j=0; j < 100; j++) \\
x[i][j] &= 2 \times x[i][j];
\end{align*}
\]
#9: Compiler Optimizations: Blocking
*(To Reduce Miss Rate)*

```c
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
        r = 0.0;
        for (k=0; k < N; k++)
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
    }
```
#9: Compiler Optimizations: Blocking

*(To Reduce Miss Rate)*

```c
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
        r = 0.0;
        for (k=0; k < N; k++)
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
    }
```

Accesses = anywhere between $3N^2$ and $(2N^3+N^2)$
#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

```c
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
        r = 0.0;
        for (k=0; k < N; k++)
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
    }
```
#9: Compiler Optimizations: Blocking

*(To Reduce Miss Rate)*

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
    {
        r = 0.0;
        for (k=0; k < N; k++)
            r = r + y[i][k]*z[k][j];
        x[i][j] = r;
    }
```
#9: Compiler Optimizations: Blocking

*(To Reduce Miss Rate)*
#9: Compiler Optimizations: Blocking

*(To Reduce Miss Rate)*

```c
for (ii=0; ii < N/B; ii++)
    for (jj=0; jj < N/B; jj++)
    {
        R = ZEROS(B,B);
        for (kk=0; kk < N/B; kk++)
            R = R ⊞ Y[ii][kk] ⊠ Z[kk][jj];
        X[ii][jj] = R;
    }
```
#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

\[
\text{for } (\text{ii}=0; \text{ii} < \text{N}/\text{B}; \text{ii}++) \\
\text{for } (\text{jj}=0; \text{jj} < \text{N}/\text{B}; \text{jj}++) \\
\{ \\
\quad \text{R} = \text{ZEROS}(\text{B},\text{B}) \\
\quad \text{for } (\text{kk}=0; \text{kk} < \text{N}/\text{B}; \text{kk}++) \\
\quad \quad \{ \\
\quad \quad \quad \text{for } (\text{i}==\text{ii}; \text{i} < \text{ii}+\text{B}; \text{i}++) \\
\quad \quad \quad \quad \text{for } (\text{j}==\text{jj}; \text{j} < \text{jj}+\text{B}; \text{j}++) \\
\quad \quad \quad \quad \quad \text{for } (\text{k}==\text{kk}; \text{k} < \text{kk}+\text{B}; \text{k}++) \\
\quad \quad \quad \quad \quad \quad \text{R}[\text{i}][\text{j}] = \text{R}[\text{i}][\text{j}] + \text{y}[\text{i}][\text{k}]*\text{z}[\text{k}][\text{j}] \\
\quad \quad \quad \} \\
\quad \text{X}[\text{ii}][\text{jj}] = \text{R}; \\
\} \\
\]
#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

```
for (ii=0; ii < N/B; ii++)
    for (jj=0; jj < N/B; jj++)
        for (kk=0; kk < N/B; kk++)
            for (i=ii; i < ii+B; i++)
                for (j=jj; j < jj+B; j++)
                {
                    r = 0.0;
                    for (k=kk; k < kk+B; k++)
                        r = r + y[i][k]*z[k][j];
                    x[i][j] = r;
                }
for (ii=0; ii < N/B; ii++)
    for (jj=0; jj < N/B; jj++)
    {
        R = ZEROS(B,B);
        for (kk=0; kk < N/B; kk++)
        {
            for (i=ii; i < ii+B; i++)
                for (j=jj; j < jj+B; j++)
                for (k=kk; k < kk+B; k++)
                    R[i][j] = R[i][j] + y[i][k]*z[k][j];
        }
        X[ii][jj] = R;
    }
```
#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

```c
for (jj=0; jj < N; jj = jj+B)
  for (kk=0; kk < N; kk = kk+B)
    for (i=0; i < N; i++)
      for (j=jj; j < min(jj+B,N); j++)
        {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
            r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
        }
```
#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

for (jj=0; jj < N; jj = jj+B)
for (kk=0; kk < N; kk = kk+B)
for (i=0; i < N; i++)
for (j=jj; j < min(jj+B,N); j++)
{
    r = 0.0;
    for (k=kk; k < min(kk+B,N); k++)
        r = r + y[i][k]*z[k][j];
    x[i][j] = x[i][j]+r;
}
for (jj=0; jj < N; jj = jj+B)
for (kk=0; kk < N; kk = kk+B)
for (i=0; i < N; i++)
for (j=jj; j < min(jj+B,N); j++)
{
    r = 0.0;
    for (k=kk; k < min(kk+B,N); k++)
        r = r + y[i][k]*z[k][j];
    x[i][j] = x[i][j]+r;
}
#9: Compiler Optimizations: Blocking  
*(To Reduce Miss Rate)*

```c
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
                {
                    r = 0.0;
                    for (k=kk; k < min(kk+B,N); k++)
                        r = r + y[i][k]*z[k][j];
                    x[i][j] = x[i][j]+r;
                }
```
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
            {
                r = 0.0;
                for (k=kk; k < min(kk+B,N); k++)
                {
                    r = r + y[i][k]*z[k][j];
                    x[i][j] = x[i][j]+r;
                }
            }
#9: Compiler Optimizations: Blocking
*(To Reduce Miss Rate)*

```c
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
            {
                r = 0.0;
                for (k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k]*z[k][j];
                x[i][j] = x[i][j]+r;
            }
```

```
<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>z</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

© 2007 Elsevier Inc. All rights reserved.
#9: Compiler Optimizations: Blocking

*(To Reduce Miss Rate)*

```c
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
                {
                    r = 0.0;
                    for (k=kk; k < min(kk+B,N); k++)
                        r = r + y[i][k]*z[k][j];
                    x[i][j] = x[i][j]+r;
                }
```
#9: Compiler Optimizations: Blocking

(To Reduce Miss Rate)

```plaintext
for (jj=0; jj < N; jj = jj+B)
  for (kk=0; kk < N; kk = kk+B)
    for (i=0; i < N; i++)
      for (j=jj; j < min(jj+B,N); j++)
        {
          r = 0.0;
          for (k=kk; k < min(kk+B,N); k++)
            r = r + y[i][k]*z[k][j];
          x[i][j] = x[i][j]+r;
        }
```
#9: Compiler Optimizations: Blocking
*(To Reduce Miss Rate)*

```
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
            {
                r = 0.0;
                for (k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k]*z[k][j];
                x[i][j] = x[i][j]+r;
            }
```

Accesses:

\[ z : N^2 \]
#9: Compiler Optimizations: Blocking
*(To Reduce Miss Rate)*

```c
for (jj=0; jj < N; jj = jj+B)  
  for (kk=0; kk < N; kk = kk+B)  
    for (i=0; i < N; i++)  
      for (j=jj; j < min(jj+B,N); j++)  
        {  
          r = 0.0;  
          for (k=kk; k < min(kk+B,N); k++)  
            r = r + y[i][k]*z[i][j];  
          x[i][j] = x[i][j]+r;  
        }
```

Accesses:
\[z : N^2\]

Compulsory misses
#9: Compiler Optimizations: Blocking (To Reduce Miss Rate)

Accesses:
- \( z : N^2 \)
- \( x : B^2 \times N/B \times \# \text{blocks} = N \times B \times (N/B)^2 = N^3/B \)

```c
for (jj=0; jj < N; jj = jj+B)
  for (kk=0; kk < N; kk = kk+B)
    for (i=0; i < N; i++)
      for (j=jj; j < min(jj+B,N); j++)
        r = 0.0;
        for (k=kk; k < min(kk+B,N); k++)
          r = r + y[i][k]*z[k][j];
        x[i][j] = x[i][j]+r;
```

Compulsory misses
#9: Compiler Optimizations: Blocking

*(To Reduce Miss Rate)*

for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
                {
                    r = 0.0;
                    for (k=kk; k < min(kk+B,N); k++)
                        r = r + y[i][k]*z[k][j];
                    x[i][j] = x[i][j]+r;
                }

Elements in a block

Accesses:

- \(z: N^2\)
- \(x: B^2 \times N/B \times \#blocks = N \times B \times (N/B)^2 = N^3/B\)

Compulsory misses
#9: Compiler Optimizations: Blocking
(To Reduce Miss Rate)

Accesses:
- \( z : N^2 \)
- \( x : B^2 \times N/B \times \text{#blocks} = N \times B \times (N/B)^2 = N^3/B \)

Compulsory misses

```
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
            {
                r = 0.0;
                for (k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k]*z[k][j];
                x[i][j] = x[i][j]+r;
            }
```
#9: Compiler Optimizations: Blocking

*(To Reduce Miss Rate)*

**Accesses:**

- \( z : N^2 \)
- \( x : B^2 \times N/B \times \#blocks = N \times B \times (N/B)^2 = N^3/B \)
- \( y : N^3/B \)

**Number of sweeps**

**Total:** \( 2N^3/B+N^2 \) (against \( 2N^3+N^2 \))

```
for (jj=0; jj < N; jj = jj+B)
    for (kk=0; kk < N; kk = kk+B)
        for (i=0; i < N; i++)
            for (j=jj; j < min(jj+B,N); j++)
            {
                r = 0.0;
                for (k=kk; k < min(kk+B,N); k++)
                    r = r + y[i][k]*z[k][j];
                x[i][j] = x[i][j]+r;
            }
```
#10: Hardware Prefetching

(To Reduce Miss Penalty or Miss Rate)

• Instruction prefetch
  ★ prefetch two blocks, instead of one, on miss

• Data prefetch
  ★ extend the same idea to data
  ★ an older study found 50% to 70% misses could be captured with 8 stream buffers (one for instruction, 7 for data)
  ★ Pentium 4 can prefetch into L2 cache from up to 8 streams
    ★ invokes prefetch upon two successive misses to a page
    ★ won’t prefetch across 4KB page boundary
#10: Hardware Prefetching  
*(To Reduce Miss Penalty or Miss Rate)*

Speedup due to hardware prefetching on Pentium 4
#11: Compiler-Controlled Prefetching
(To Reduce Miss Penalty or Miss Rate)

- Register prefetch
  - preload register
- Cache prefetch
  - load into the cache, but not register
- Either could be faulting or non-faulting
  - normal load is faulting register prefetch
  - non-faulting prefetches turn into no-ops
- Usually need non-blocking caches to be effective
## Cache Optimization Summary

<table>
<thead>
<tr>
<th>Technique</th>
<th>Hit time</th>
<th>Bandwidth</th>
<th>Miss penalty</th>
<th>Miss rate</th>
<th>Hardware cost/complexity</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small and simple caches</td>
<td>+</td>
<td></td>
<td>–</td>
<td>0</td>
<td>0</td>
<td>Trivial; widely used</td>
</tr>
<tr>
<td>Way-predicting caches</td>
<td>+</td>
<td></td>
<td>–</td>
<td>1</td>
<td>1</td>
<td>Used in Pentium 4</td>
</tr>
<tr>
<td>Trace caches</td>
<td>+</td>
<td></td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Used in Pentium 4</td>
</tr>
<tr>
<td>Pipelined cache access</td>
<td>–</td>
<td>+</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Widely used</td>
</tr>
<tr>
<td>Nonblocking caches</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Widely used</td>
</tr>
<tr>
<td>Banked caches</td>
<td>+</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Used in L2 of Opteron and Niagara</td>
</tr>
<tr>
<td>Critical word first and early restart</td>
<td>+</td>
<td></td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>Widely used</td>
</tr>
<tr>
<td>Merging write buffer</td>
<td>+</td>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>Widely used with write through</td>
</tr>
<tr>
<td>Compiler techniques to reduce cache misses</td>
<td>+</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Software is a challenge; some computers have compiler option</td>
</tr>
<tr>
<td>Hardware prefetching of instructions and data</td>
<td>+</td>
<td>+</td>
<td>2 instr., 3 data</td>
<td>0</td>
<td>0</td>
<td>Many prefetch instructions; Opteron and Pentium 4 prefetch data</td>
</tr>
<tr>
<td>Compiler-controlled prefetching</td>
<td>+</td>
<td>+</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>Needs nonblocking cache; possible instruction overhead; in many CPUs</td>
</tr>
</tbody>
</table>
VIRTUAL MACHINES
A virtual machine is taken to be an efficient, isolated duplicate of the real machine. We explain these notions through the idea of a virtual machine monitor (VMM) ... a VMM has three essential characteristics. First, the VMM provides an environment for programs which is essentially identical with the original machine; second, programs run in this environment show at worst only minor decreases in speed; and last, the VMM is in complete control of the system resources.

Gerlad Popek and Robert Goldberg
“Formal requirements for virtualizable third generation architectures,”
Communications of the ACM (July 1974)
Protection via Virtual Machines

• (Operating) System Virtual Machines
  ★ does not include JVM or Microsoft CLR
  ★ VMware ESX, Xen (hypervisors or virtual machine monitors, can run on bare machines)
  ★ Parallels, VMware Fusion (run on a host OS)

• Regained popularity
  ★ increased importance of isolation and security
  ★ failures in security and reliability of standard OSes
  ★ sharing of computers among unrelated users
  ★ increased hardware speeds, making VM overheads acceptable
Popularity of Virtual Machines: II

• Protection
  ✤ see previous slide

• Software management
  ✤ could run legacy operating systems

• Hardware management
  ✤ let separate software stacks share hardware
    ✤ also useful at the end-user level
  ✤ some VMMs support migration of a running VM to a
different computer, for load-balancing and fault tolerance

Hardware that allows VM to execute directly
on hardware is called virtualizable
Complications

• “Difficult” instructions
  ★ *paravirtualization*: make some minimal changes to the guest OS to avoid difficult instructions

• Virtual memory
  ★ separate *virtual, physical, and machine* memory
  ★ maintain shadow page table to avoid double translation; alternatively, need hardware support for multiple indirections

• TLB virtualization
  ★ VMM maintains per-OS TBB copies
  ★ TLBs with process-ID tag can avoid TLB flush on VM context switch through the use of virtual PIDs

• I/O sharing
Xen vs Native Linux

![Performance bar chart comparison between Xen and Native Linux](chart.png)

- SPECint2000: 100%
- Linux build time: 97%
- PostgreSQL inf. retrieval: 92%
- PostgreSQL OLTP: 95%
- dbench: 96%
- SPECWeb99: 99%

© 2007 Elsevier, Inc. All rights reserved.