The Memory Hierarchy

Memory hierarchy
Cache memory
Cache-block alignment
Prefetching
Loop interchange
Blocking
Loop fusion
Loop fission
The Memory Hierarchy

- REGISTERS
- CACHE
- MAIN MEMORY
- DISK MEMORY

Increasing cost → Decreasing speed

Size
Cache Memory

- When the processor requires the datum at memory address $x$, it first looks in the cache.
- If $x$ is in the cache, a *cache hit* occurs.
- If $x$ is not in the cache, a *cache miss* occurs. The processor fetches $x$ from main memory and places a copy of $x$ in the cache.
- Placing $x$ in the cache may mean displacing other
Cache Organization (1)

- The binary expansion of a memory address is divided into a memory block address and an offset within the block.
- The cache blocksize $B$ indicates how many contiguous bytes of memory are copied to cache.
Cache Organization (2)

- A memory address is further divided into a cache set and a tag.
- The number of cache sets \( S \) indicates how many sets are in the entire cache.
- What is associativity \( A \)?

**MEMORY ADDRESS:**

<table>
<thead>
<tr>
<th>tag</th>
<th>cache set</th>
<th>offset</th>
</tr>
</thead>
</table>

\[ \log_2 S \text{ bits} \]  
\[ \log_2 B \text{ bits} \]
Cache Organization (3)

- The capacity of the entire cache is $C = A \times B \times S$.
- Why is the organization of the cache so seemingly complex?
Mapping Memory to Cache

- A cache contains \( \frac{C}{B} \) cache frames, and each frame may be empty or may by occupied by a memory block.
- No two cache frames may contain the same memory block. Why?
- A cache set is a collection of \( A \) frames that a particular memory block may occupy in cache.
- The tag disambiguates memory blocks within the same cache set.
Cache Associativity

*direct-mapped*
$A=1, S=8$

*2-way associative*
$A=2, S=4$

*4-way associative*
$A=4, S=2$

*fully associative*
$A=8, S=1$
Locality

• A program is said to have good *locality of reference* if it can reuse data while it is still in the upper levels of memory.

• This requires that the use and reuse of data be close together in time.

• *Temporal locality*: accessing recently-referenced data.

• *Spatial locality*: accessing data with memory addresses near that of recently-referenced data.
Cache-Replacement Policy

- On a cache miss, the replacement policy selects which frame in a cache set to update with the new memory block. Optimal policy?
- *Random*: choose randomly which block to replace.
- *First-in first-out* (FIFO): choose to replace the block that has resided in the cache set the longest.
- *Least-recently used* (LRU): choose to replace the block that has been unused the longest.
Cache-Miss Categories (1)

1. A *compulsory* (or *cold*) *miss* occurs on the very first access to a memory block. Why? Possible to avoid?

2. A *capacity* *miss* occurs when accessing a block that previously resided in cache, but was replaced because the cache cannot hold all of the data needed to execute a program. Possible to avoid? (A capacity miss in an LRU set-associative cache with capacity $C$ is also a miss in an LRU fully associative cache with capacity $C$.)
3. A conflict miss occurs when accessing a block that previously resided in cache, but was replaced because too many blocks map to the same cache set.

Possible to avoid?

(A conflict miss in an LRU set-associative cache with capacity $C$ is a hit in an LRU fully associative cache with capacity $C$.)

Why wouldn't we make all caches fully-associative, to avoid conflict misses?
Write-Hit Policy

- What happens when the CPU writes data at address $x$?
- If $x$ is in the cache, it is a write-hit.
- On a write-hit, main memory may be updated at the time of the hit (write-through) or only when the cache block is flushed from the cache (write-back).
- What are the consequences of each policy?
Write-Miss Policy

- If $x$ is not in the cache, it is a write-miss.
- *Fetch-on-write*: Word at $x$ is written to cache and the other words in block are fetched from main memory. Why?
- *Write-validate*: Word at $x$ is written to cache and the other words in block are marked as invalid.
- *Write-around*: Word at $x$ is written directly to main memory. What happens next time $x$ is required?
Cache-Block Alignment

- Recall that a contiguous block of one or more words is fetched from memory every time.
- A program that accesses one field of object $x$ is likely to access other fields of $x$.
- Suppose $x$ straddles a block boundary.

To access all of $x$, which fits in a single block, two blocks must be fetched.
There are two simple ways to avoid splitting objects across blocks unnecessarily:

1. Allocate objects sequentially and if the next object does not fit in the remaining portion of the block, skip

2. Allocate two-word-size objects in one area of memory, four-word-size objects in another, and so on.
Alignment in Instruction Cache

- Instructions occupy cache blocks just as data do.
- Aligning the beginning of frequently executed basic blocks on cache block boundaries increases the number of basic blocks that fit simultaneously in cache.
- Infrequently-executed instructions should not be placed in the same cache blocks as frequently-executed ones.
- Trace scheduling can be used to order a frequently executed path through contiguous set of cache blocks.
Prefetching

- A miss on a load instruction causes a significant delay while the data is fetched from main memory.
- In some cases, the need for the data is predictable many cycles earlier.
- The compiler can insert a *prefetch* instruction to start the fetching earlier.
- What is the goal of inserting such an instruction?
A prefetch instruction is just a hint to hardware to start bringing the data at address $x$ to cache.

Why should a prefetch be a hint, rather than mandatory?

If `prefetch(x)` is successful, next load from $x$ hits.

If `prefetch(x)` is unsuccessful, next load from $x$ may miss but the program still executes correctly.
Nonblocking Load

- Some machines have an actual prefetch instruction.
- Others may still be able to do prefetching if they have a nonblocking load instruction.
- When $r_3 \ M[x]$ is performed, the processor does not stall (even if the load results in a cache miss) until $r_3$ is used as the operand of some other instruction.
- A nonblocking load can start bringing the value to cache, without stalling the rest of the program.
Inserting Prefetch Instructions

- If the computation accesses every word of an array sequentially, we need not prefetch every word.
- Why? How often should we prefetch?
- Assuming 4-byte words and 16-byte cache blocks, how should we insert prefetch instructions?

```plaintext
L: x ← M[i]
y ← M[j]
z ← x * y
s ← s + z
i ← i + 4
j ← j + 4
if i < N goto L
```
Prefetching in a Loop

$L$: \textbf{if} $i \mod 16 = 0$ \textbf{then} prefetch $M[i+K]$

\textbf{if} $j \mod 16 = 0$ \textbf{then} prefetch $M[j+K]$

$x \leftarrow M[i]$

$y \leftarrow M[j]$

$z \leftarrow x \times y$

$s \leftarrow s + z$

$i \leftarrow i + 4$

$j \leftarrow j + 4$

\textbf{if} $i < N$ \textbf{goto} $L$

• Choose $K$ to match expected miss latency.

• Suppose we use prefetching to hide a latency of 11 instructions.

• Rounded up to next multiple of cache blocksize, $K=16$.

• How can we avoid testing $i \mod 16 = 0$ on every iteration?
unrolled loop

$L_1$: prefetch $M[i+16]$
prefetch $M[j+16]$

$x \leftarrow M[i]$
$y \leftarrow M[j]$

$z \leftarrow x \times y$
$s \leftarrow s + z$
i $\leftarrow i + 4$

$j \leftarrow j + 4$

if $i \geq N$ goto $L_2$

<table>
<thead>
<tr>
<th>BODY</th>
</tr>
</thead>
</table>
if $i \geq N$ goto $L_2$

<table>
<thead>
<tr>
<th>BODY</th>
</tr>
</thead>
</table>
if $i \geq N$ goto $L_2$

<table>
<thead>
<tr>
<th>BODY</th>
</tr>
</thead>
</table>
if $i < N$ goto $L_1$

$L_2:$

improved unrolled loop

if $i \geq N-16$ goto $L_2$

$L_1$: prefetch $M[i+16]$
prefetch $M[j+16]$

$x \leftarrow M[i];
\ y \leftarrow M[j]$

$z \leftarrow x \times y;
\ s \leftarrow s + z$

$x \leftarrow M[i+4];
\ y \leftarrow M[j+4]$

$z \leftarrow x \times y;
\ s \leftarrow s + z$

$x \leftarrow M[i+8];
\ y \leftarrow M[j+8]$

$z \leftarrow x \times y;
\ s \leftarrow s + z$

$x \leftarrow M[i+12];
\ y \leftarrow M[j+12]$

$z \leftarrow x \times y;
\ s \leftarrow s + z$
i $\leftarrow i + 16;
\ j \leftarrow j + 16$

if $i < N-16$ goto $L_1$ else $L_2$

$L_2$: $x \leftarrow M[i];
\ y \leftarrow M[j]$

$z \leftarrow x \times y;
\ s \leftarrow s + z$
i $\leftarrow i + 4;
\ j \leftarrow j + 4$

if $i < N$ goto $L_2$

$L_3:$
Prefetching Stores

for $i \leftarrow 0$ to $N-1$

$A[i] \leftarrow i$

- Each time $i$ crosses a new cache-block boundary, there is a write miss.
- If write-invalidate, no processor stall—no prefetch.
- If fetch-on-write, processor stall at each new cache block.

for $i \leftarrow 0$ to $N-1$

if $i \text{ mod blocksize}=0$ then prefetch $A[i+K]$

$A[i] \leftarrow i$
Accessing Memory in Nested Loops

- When nested loops access memory, successive iterations often reuse the same word or use adjacent words that occupy the same cache block.
- If it is the innermost loop whose iterations reuse the same values, there will be many cache hits.
- But if one of the outer loops reuses a cache block, it may be that the inner loops access enough
Example: Nested Loop

\[
\begin{align*}
&\text{for } i \leftarrow 0 \text{ to } N-1 \\
&\quad \text{for } j \leftarrow 1 \text{ to } M-2 \\
&\quad \quad \text{for } k \leftarrow 0 \text{ to } P-1 \\
&\quad \quad \quad A[i, j, k] \leftarrow (B[i, j-1, k]+B[i, j, k]+B[i, j+1, k]) / 3
\end{align*}
\]

- \(B[i, j+1, k]\) is reused in the next two iterations of the j-loop. How?
- But before the next iteration of \(j\)-loop the \(k\)-loop accesses \(4P\) different array elements.
- It is very possible that these accesses conflict with \(B[i, j+1, k]\) causing a miss the next time it's fetched.
Example: Loop Interchange

for $i \leftarrow 0$ to $N-1$
    for $k \leftarrow 0$ to $P-1$
        for $j \leftarrow 1$ to $M-2$
            $A[i, j, k] \leftarrow (B[i, j-1, k]+B[i, j, k]+B[i, j+1, k]) / 3$

• We can interchange the $j$-loop and the $k$-loop.
• Now $B[i, j, k]$ and $B[i, j-1, k]$ will always be cache hits.
• Is loop interchange always legal? (I.e., does it always yield the same result?)
Data Dependence

- Iteration \((j, k)\) depends on iteration \((j', k')\) if
  - \((j', k')\) computes values that are used by \((j, k)\)
    - called \textit{read-after-write} OR
  - \((j', k')\) stores values that are overwritten by \((j, k)\)
    - called \textit{write-after-write} OR
  - \((j', k')\) reads values that are overwritten by \((j, k)\)
    - called \textit{write-after-read}.
- If interchanged loops execute \((j, k)\) before \((j', k')\)
Loop-Interchange Legality

\[
\begin{align*}
\text{for } i & \leftarrow 0 \text{ to } N-1 \\
& \text{for } j \leftarrow 1 \text{ to } M-2 \\
& \text{for } k \leftarrow 0 \text{ to } P-2 \\
& \quad A[i, j, k] \leftarrow (B[i, j-1, k] + B[i, j, k] + B[i, j+1, k]) / 3 \\
& \quad C[j, k] \leftarrow C[j-1, k+1]
\end{align*}
\]

- Does interchanging the $j$-loop and the $k$-loop decrease the number of cache misses?
- Is interchanging the $j$-loop and the $k$-loop legal?
Loop Interchange for Spatial Locality

\[
\begin{align*}
&\text{for } i \leftarrow 0 \text{ to } N-1 \\
&\quad \text{for } j \leftarrow 0 \text{ to } M-1 \\
&\quad A[i, j] \leftarrow B[i, j]
\end{align*}
\]

- Does interchanging the \(i\)-loop and the \(j\)-loop decrease the number of cache misses?
- Is interchanging the \(i\)-loop and the \(j\)-loop legal?
Matrix-Multiply Loop Nest

\[
\text{for } i \leftarrow 0 \text{ to } N-1 \\
\text{for } j \leftarrow 0 \text{ to } M-1 \\
\text{for } k \leftarrow 0 \text{ to } P-1 \\
\quad C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]
\]

- Is there any interchanging loops that will decrease the number of cache misses?
- Is there any other useful transformation?
Matrix-Multiply Loop Nest (1)

```
for i ← 0 to N-1
    for j ← 0 to N-1
        for k ← 0 to N-1
```

- Suppose $N=50$, each element is a float (8 bytes), and the cache capacity is 16 kilobytes.
- Every reference to $B[k,j]$ in the innermost loop will incur a cache miss. All other elements of $B$ map to the cache in between its use and reuse in the $i$-loop.
- Will interchanging any of the loops help?
Matrix-Multiply Loop Nest (2)

- The solution is to reuse rows of $A$ and columns of $B$ while they are still in the cache.
- A $c \times c$ block of $C$ can be calculated from $c$ rows of $A$ and $c$ columns of $B$.

\[
\begin{align*}
\text{for } i & \leftarrow i_0 \text{ to } i_0+c-1 \\
\text{for } j & \leftarrow j_0 \text{ to } j_0+c-1 \\
\text{for } k & \leftarrow 0 \text{ to } N-1 \\
C[i, j] & \leftarrow C[i, j] + A[i, k] \times B[k, j]
\end{align*}
\]

- Only $c \times N$ elements of $A$ and $c \times N$ elements of $B$ are used in this loop (each used $c$ times).
Matrix-Multiply Loop Nest (3)

Each element of $C$ is computed from a row of $A$ and a column of $B$.

Each $c \times c$ block of $C$ is computed from a $c \times N$ block of $A$ and a $N \times c$ block of $B$. 
Matrix-Multiply Loop Nest (4)

Now to compute each block of $C$, we need to set our loops for computing a single block of $C$ inside some outer loops.

```
for $i_0 \leftarrow 0$ to $N-1$ by $c$
  for $j_0 \leftarrow 0$ to $N-1$ by $c$
    for $i \leftarrow i_0$ to $\min(i_0+c-1, N-1)$
      for $j \leftarrow j_0$ to $\min(j_0+c-1, N-1)$
        for $k \leftarrow 0$ to $N-1$
          $C[i, j] \leftarrow C[i, j] + A[i, k] \times B[k, j]$
```
Blocking

- The *blocking* transformation reorders computations so that all computations that use one portion (i.e., block) of data are computed before moving on to the next portion.
- For many loop nests, an optimizing compiler can automatically perform blocking.
- However, there must be no data dependencies.
- How is $c$ set?
Scalar Replacement

- Accesses to $C[i,j]$ in the innermost $k$-loop will almost always result in a cache hit.
- To do better, we can “cache” $C[i,j]$ in a register.

```plaintext
for i ← $i_0$ to $i_0+c-1$
  for j ← $j_0$ to $j_0+c-1$
    s ← $C[i, j]$
    for k ← 0 to N-1
      s ← s + $A[i, k]*B[k, j]$
    C[i, j] ← s
```

- *Scalar replacement* avoids many fetches and stores.
Blocking for Registers

- To do blocking for L1 or L2 cache, we can set the value of $c$ appropriately.
- We can also block for registers.
- Suppose there are 32 floating-point registers, and we want to use $d$ of them as a kind of a cache.

```plaintext
for i ← i₀ to i₀+c-1
  for k₀ ← 0 to N-1 by d
    for k ← k₀ to k₀+d-1
      T[k-k₀] ← A[i, k]
  for j ← j₀ to j₀+c-1
    s ← C[i, j]
    for k ← k₀ to k₀+d-1
      s ← s + T[k-k₀] * B[k, j]
    C[i, j] ← s
```
Unroll and Jam

- Registers cannot be indexed using subscripts.
- Unroll the $k$-loops $d$ times and keep each $T[k]$ in a separate scalar temporary variable.
- Let $d=3$, even though $d=25$ would be more realistic.

```plaintext
for i ← i₀ to i₀+c-1
  for k₀ ← 0 to N-1 by 3
    t₀ ← A[i, k₀];  t₁ ← A[i, k₀+1];  t₂ ← A[i, k₀+2]
    for j ← j₀ to j₀+c-1
      C[i, j] ← C[i, j] + t₀*B[k₀, j] + t₁*B[k₀+1, j] + t₂*B[k₀+2, j]
```
Loop Fusion (1)

• We would like to reuse access $A[i-1]$ (of the second loop) as $A[i]$ in the next iteration (of the first loop).

• *Loop fusion* combines adjacent loops to exploit

\[
\text{for } i \leftarrow 1 \text{ to } N \\
\quad x \leftarrow A[i] + x
\]

\[
\text{for } i \leftarrow 1 \text{ to } N \\
\quad C[i] \leftarrow A[i-1] + D[i-1] \\
\quad D[i] \leftarrow C[i]
\]

original loop

\[
\text{for } i \leftarrow 1 \text{ to } N \\
\quad x \leftarrow A[i] + x \\
\quad C[i] \leftarrow A[i-1] + D[i-1] \\
\quad D[i] \leftarrow C[i] \\
\text{fused loop}
\]
Loop Fusion (2)

Is loop fusion always legal?

for $i \leftarrow 1$ to $N-1$
  $A[i] \leftarrow x$

for $i \leftarrow 1$ to $N-1$
  $C[i] \leftarrow A[i+1] + D[i-1]$
  $D[i] \leftarrow C[i]$

original loop

for $i \leftarrow 1$ to $N-1$
  $A[i] \leftarrow x$
  $C[i] \leftarrow A[i+1] + D[i-1]$
  $D[i] \leftarrow C[i]$

fused loop
Loop Fission or Distribution

- Reuse of $A[j,k]$ in the $i$-loop may result in a miss if the accesses to $C$ and $D$ cause displacement.

- *Loop fission* splits a single loop nest into multiple adjacent loops to distribute memory accesses.

```plaintext
for i ← 0 to N-1
  for j ← 0 to N-1
    for k ← 0 to N-1
      B[i] ← A[j,k]
      C[i] ← D[k,j]
original loop
```

```plaintext
for i ← 0 to N-1
  for j ← 0 to N-1
    for k ← 0 to N-1
      B[i] ← A[j,k]

for i ← 0 to N-1
  for j ← 0 to N-1
    C[i] ← D[k,j]
distributed loop
```