Memory Optimization
Outline

- Issues with the Memory System
- Loop Transformations
- Data Transformations
- Prefetching
- Alias Analysis
Memory Hierarchy

- **Registers**: 32 – 512 B
- **L1 Private Cache**: 16 – 128 KB
- **L2/L3 Shared Cache**: 1 – 16 MB
- **Main Memory (DRAM)**: 1 GB – 128 GB
- **Permanent Storage (Hard Disk)**: 250 GB – 4 TB
Processor-Memory Gap

- µProc: 60%/year (2×/1.5yr)
- DRAM: 9%/year (2×/10 yrs)

Performance vs. Year
# Cache Architecture

<table>
<thead>
<tr>
<th></th>
<th>Pentium D</th>
<th>Core Duo</th>
<th>Core 2 Duo</th>
<th>Athlon 64</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 code (per core)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>12 K uops</td>
<td>32 KB</td>
<td>32 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
<td>2 way</td>
</tr>
<tr>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L1 data (per core)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>16 KB</td>
<td>32 KB</td>
<td>32 KB</td>
<td>64 KB</td>
</tr>
<tr>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
<td>8 way</td>
</tr>
<tr>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L1 to L2</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>4 cycles</td>
<td>3 cycles</td>
<td>3 cycles</td>
<td>3 cycles</td>
</tr>
<tr>
<td><strong>L2 shared</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>size</td>
<td>4 MB</td>
<td>4 MB</td>
<td>4 MB</td>
<td>1 MB</td>
</tr>
<tr>
<td>associativity</td>
<td>8 way</td>
<td>8 way</td>
<td>16 way</td>
<td>16 way</td>
</tr>
<tr>
<td>Line size</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
<td>64 bytes</td>
</tr>
<tr>
<td><strong>L2 to L3 (off)</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Latency</td>
<td>31 cycles</td>
<td>14 cycles</td>
<td>14 cycles</td>
<td>20 cycles</td>
</tr>
</tbody>
</table>
Cache Misses

- Cold misses
  - First time a data is accessed

- Capacity misses
  - Data got evicted between accesses because a lot of other data (more than the cache size) was accessed

- Conflict misses
  - Data got evicted because a subsequent access fell on the same cache line (due to associativity)

- True sharing misses (multicores)
  - Another processor accessed the data between the accesses

- False sharing misses (multicores)
  - Another processor accessed different data in the same cache line between the accesses
Data Reuse

• Temporal Reuse
  – A given reference accesses the same location in multiple iterations

• Spatial Reuse
  – Accesses to different locations within the same cache line

• Group Reuse
  – Multiple references access the same location

\[
\begin{align*}
  &\text{for } i = 0 \text{ to } N \\
  &\text{for } j = 0 \text{ to } N \\
  &A[j] = \\
  \\
  &\text{for } i = 0 \text{ to } N \\
  &\text{for } j = 0 \text{ to } N \\
  &B[i, j] = \\
  \\
  &\text{for } i = 0 \text{ to } N \\
\end{align*}
\]
Outline

• Issues with the Memory System
• Loop Transformations
• Data Transformations
• Prefetching
• Alias Analysis
Matrix Multiply

for i = 1 to n
    for j = 1 to n
        for k = 1 to n
            c[i,j] += a[i,k]*b[k,j]
Example: Matrix Multiply

Data Accessed

1,050,624
Matrix Multiply

for i0 = 1 to n step b
  for j0 = 1 to n step b
    for k0 = 1 to n step b
      for 1 = i0 to min(i0+b-1, n)
        for j = j0 to min(j0+b-1, n)
          for k = k0 to min(k0+b-1, n)
            c[i,j] += a[i,k]*b[k,j]
Example: Matrix Multiply

Data Accessed

1,050,624

66,560
Loop Transformations

- Transform the iteration space to reduce the number of misses
- Reuse distance – For a given access, number of other data items accessed before that data is accessed again
- Reuse distance > cache size
  - Data is spilled between accesses
Divide and Conquer Matrix Multiply

\[
\begin{bmatrix}
A & B \\
C & D \\
\end{bmatrix}
\times
\begin{bmatrix}
E & F \\
G & H \\
\end{bmatrix}
=
\begin{bmatrix}
AE+BG & AF+BH \\
CE+DG & CF+DH \\
\end{bmatrix}
\]
for i = 0 to N
  for j = 0 to N
    for k = 0 to N
      A[k,j]

Reuse distance = $N^2$

If Cache size < 16 doubles?
A lot of capacity misses
Loop Transformations

for i = 0 to N
  for j = 0 to N
    for k = 0 to N
      A[k,j]

Loop Interchange

for j = 0 to N
  for i = 0 to N
    for k = 0 to N
      A[k,j]
for j = 0 to N
    for i = 0 to N
        for k = 0 to N
            A[k,j]

Cache line size > data size
Cache line size = L
Reuse distance = LN

If cache size < 8 doubles?
    Again a lot of capacity misses
Loop Transformations

for $j = 0$ to $N$
    for $i = 0$ to $N$
        for $k = 0$ to $N$
            $A[k,j]$

Loop Interchange

for $k = 0$ to $N$
    for $i = 0$ to $N$
        for $j = 0$ to $N$
            $A[k,j]$
Loop Transformations

for i = 0 to N
    for j = 0 to N
        for k = 0 to N

• No matter what loop transformation you do one array access has to traverse the full array multiple times
for i = 0 to N
  for j = 0 to N

for ii = 0 to ceil(N/b)
  for jj = 0 to ceil(N/b)
    for i = b*ii to min(b*ii+b-1, N)
      for j = b*jj to min(b*jj+b-1, N)
Outline

• Issues with the Memory System
• Loop Transformations
• Data Transformations
• Prefetching
• Alias Analysis
False Sharing Misses

for J =
    forall I =
    X(I, J) = ...

Array X
Conflict Misses

for J =
forall I =
X(I, J) = ...
Data Transformations

- Similar to loop transformations
- All the accesses have to be updated
  - Whole program analysis is required
# Strip-Mining

Create two dims from one

With blocksize=4

<table>
<thead>
<tr>
<th>Storage Declaration</th>
<th>Array Access</th>
<th>Memory Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>([N])</td>
<td>(i)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(i \mod 4)</td>
<td>[0,0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1,0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2,0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3,0]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0,1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[1,1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[2,1]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[3,1]</td>
</tr>
</tbody>
</table>
### Strip-Minding
Create two dims from one

With blocksize=4

### Permutation
Change memory layout

With permutation matrix \[
\begin{pmatrix}
0 & 1 \\
1 & 0 \\
\end{pmatrix}
\]

<table>
<thead>
<tr>
<th>Storage Declaration</th>
</tr>
</thead>
</table>
| \[
\begin{bmatrix}
N \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
4 \\
\frac{N}{4}
\end{bmatrix}
\]
| \[
\begin{bmatrix}
N_1 \\
N_2
\end{bmatrix}
\rightarrow
\begin{bmatrix}
N_2 \\
N_1
\end{bmatrix}
\]|

<table>
<thead>
<tr>
<th>Array Access</th>
</tr>
</thead>
</table>
| \[
\begin{bmatrix}
i \\
\end{bmatrix}
\rightarrow
\begin{bmatrix}
i \mod 4 \\
i / 4
\end{bmatrix}
\]
| \[
\begin{bmatrix}
i_1 \\
i_2
\end{bmatrix}
\rightarrow
\begin{bmatrix}
i_2 \\
i_1
\end{bmatrix}
\]|

<table>
<thead>
<tr>
<th>Memory Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
| \[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & 1,0 & 2,0 & 3,0 & 4,0 & 5,1 & 6,2 & 7,3 \\
\end{array}
\]
| \[
\begin{array}{cccc}
0 & 1,0 & 2,0 & 3,0 \\
0 & 1,1 & 2,1 & 3,1 \\
0,0 & 1,1 & 2,1 & 3,1 \\
\end{array}
\]
| \[
\begin{array}{cccc}
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3 \\
0,0 & 1,1 & 2,2 & 3,3 \\
\end{array}
\]|
Data Transformation Algorithm

- Rearrange data: Each processor’s data is contiguous
- Use data decomposition
  - *, block, cyclic, block-cyclic
- Transform each dimension according to the decomposition
- Use a combination of strip-mining and permutation primitives
Example I: (Block, Block)
Example I: (Block, Block)

\[ i_1 \mod \left\lfloor \frac{d_1}{P} \right\rfloor \]

\[ i_1 / \left\lfloor \frac{d_1}{P} \right\rfloor \]

Strip-Mine

\[ i_2 \]
Example I: (Block, Block)

Strip-Mine

\[ i_1 \mod \left\lfloor \frac{d_1}{P} \right\rfloor \]
\[ i_1 \left/ \left\lfloor \frac{d_1}{P} \right\rfloor \right. \]

Permute

\[ i_1 \mod \left\lfloor \frac{d_1}{P} \right\rfloor \]
\[ i_2 \]
\[ i_1 \left/ \left\lfloor \frac{d_1}{P} \right\rfloor \right. \]
Example I: (Cyclic, *)
Example I: (Cyclic, *)
Example I: (Cyclic, *)

Strip-Mine

\[
\begin{align*}
  i_1 & \quad i_1 \mod P \\
  i_2 & \quad i_1 / P
\end{align*}
\]

Permute

\[
\begin{align*}
  i_2 & \quad i_2 \\
  i_1 / P & \quad i_1 / P
\end{align*}
\]
Performance

LU Decomposition (256x256)

5 point stencil (512x512)

LU Decomposition (1Kx1K)

- Parallelizing outer loop
- Best computation placement
- + data transformations
Optimizations

• Modulo and division operations in the index calculation
  – Very high overhead

• Use standard techniques
  – Loop invariant removal, CSE
  – Strength reduction exploiting properties of modulo and division
  – Use knowledge about the program
Outline

- Issues with the Memory System
- Loop Transformations
- Data Transformations
- Prefetching
- Alias Analysis
Prefetching

- Cache miss stalls the processor for hundreds of cycles
  - Start fetching the data early so it’ll be available when needed

- Pros
  - Reduction of cache misses → increased performance

- Cons
  - Prefetch contents for fetch bandwidth
    - Solution: Hardware only issue prefetches on unused bandwidth
  - Evicts a data item that may be used
    - Solution: Don’t prefetch too early
  - Pretech is still pending when the memory is accessed
    - Solution: Don’t prefetch too late
  - Prefetch data is never used
    - Solution: Prefetch only data guaranteed to be used
  - Too many prefetch instructions
    - Prefetch only if access is going to miss in the cache
Prefetching

- Compiler inserted
  - Use reuse analysis to identify misses
  - Partition the program and insert prefetches

- Run ahead thread (helper threads)
  - Create a separate thread that runs ahead of the main thread
  - Runahead only does computation needed for control-flow and address calculations
  - Runahead performs data (pre)fetched
Outline

- Issues with the Memory System
- Loop Transformations
- Data Transformations
- Prefetching
- Alias Analysis
Alias Analysis

- Aliases destroy local reasoning
  - Simple, local transformations require global reasoning in the presence of aliases
  - A critical issue in pointer-heavy code
  - This problem is even worse for multithreaded programs

- Two solutions
  - Alias analysis
    - Tools to tell us the potential aliases
  - Change the programming language
    - Languages have no facilities for talking about aliases
    - Want to make local reasoning possible
Aliases

• Definition

Two pointers that point to the same location are aliases

• Example

\[
\begin{align*}
Y &= \&Z \\
X &= Y \\
\*X &= 3 & /* changes the value of \*Y */
\end{align*}
\]
Example

foo(int * A, int * B, int * C, int N)
    for i = 0 to N-1

• Is this loop parallel?

• Depends

int X[1000];
int Y[1000];
int Z[1000]
foo(X, Y, Z, 1000);
Points-To Analysis

• Consider:
  \( P = \&Q \)
  \( Y = \&Z \)
  \( X = Y \)
  \( *X = P \)

• Informally:
  - P can point to Q
  - Y can point to Z
  - X can point to Z
  - Z can point to Q
Points-To Relations

• A graph
  – Nodes are program names
  – Edge \((x, y)\) says \(x\) may point to \(y\)

• Finite set of names
  – Implies each name represents many heap cells
  – Correctness: If \(*x = y\) in any state of any execution, then \((x, y)\) is an edge in the points-to graph
Sensitivity

• *Context sensitivity*
  – Separate different uses of functions
  – Different is the key – if the analysis think the input is the same, reuse the old results

• *Flow sensitivity*
  • For insensitivity makes any permutation of program statements gives same result
  • Flow sensitive is similar to data-flow analysis
Conclusion

• Memory systems are designed to give a huge performance boost for “normal” operations

• The performance gap between good and bad memory usage is huge

• Programs analyses and transformations are needed

• Can off-load this task to the compiler