Memory Management 101

Mark Mawson
Contents

• Introduction
• Basic Motherboard Structure
• Cache
  • What is cache?
  • Hits and misses
  • Cost
  • Multiple Cache Levels
• Taking advantage
Introduction

There are 5 'levels of abstraction' when writing software:

- Application level – Basic description of the problem.
- Specification Level – The mathematics of the problem are defined.
- Algorithm Level – A method of solving the problem (or an approximation of the problem).
Introduction

- There are 5 'levels of abstraction' when writing software:
  - Program Level – The algorithm is expressed as a program, constraints related to computer architecture appear.
  - Architecture Level – The program is implemented on a specific machine, changes may need to be made to maximise performance.
Introduction

• As CFD programmers, we tend to concentrate of levels 1-3. (Level 3 represents the various models for fluid flow).

• We can increase the performance of codes by optimizing in Levels 4 and 5.

• To do this we need to understand the architectures we are using.
Basic Motherboard Structure

- Registers & Cache
- Main Memory
- Hard Disks, Networks etc
Cache - what is it?

• Cache is fast but expensive memory on the CPU chip.

• When a datum is requested, the 'block' of data the requested datum is in is moved to cache.

• The 'block' of data is called a cache line. Size can be changed by the compiler (Generally ranges from 8-512 bytes).

• Assumption that we will use neighbouring data soon after. If we can do this then we reduce the amount of time spent waiting for data to be fetched from main memory.
Cache – Hits and misses

- With cache one of two scenarios can occur when the CPU requests a datum:
  - The required datum has already been fetched into cache as part of another cache line. This is a cache hit, access to the datum is fast (2-3 clock cycles).
  - The required datum is not in cache and must be fetched from the main memory. This is called a cache miss, access to the datum is slower (≈100 clock cycles).
Cache – How expensive?

- Manufacturers don't give price/MB for cache.
- Estimates for the fastest cache range from £100-£1000/MB.
- Prohibitively expensive, solution is to use multiple levels of cache with varying latencies.
- Slower caches are less expensive.
Memory Architecture

Typical Access Latencies (clock cycles):

0
CPU

Contains registers for storing data currently being used

2-3
Level 1 Cache

Very small (<64KB), very fast memory. Expensive!

6-8
Level 2 Cache

Slightly larger (512KB), and slower memory. Less expensive (£100/MB).

≈20
Level 3 Cache

Can be a few Mbs, again less expensive (still ≈ £20/MB)

≈100
Main Memory (RAM)

Can be very large (Gbs), relatively cheap (<£20/GB)
Cache – How expensive?

- Estimates of £50-£100/MB are common for Level 2.
- Level 3 cache is more easily comparable.
- Using the Xeon E5500 series as an example:
  - E5506 (4MB Level 3 cache) £217.
  - E5520 (8MB Level 3 cache) £309.
  - ≈£20/MB

Cache Levels

- Simple example to demonstrate the difference in latency between cache levels:
  - Elements of a vector are looped over and their values summed.
  - Vector size is increased to fill L1 and then L2 cache (No L3 cache in this example).
Cache Levels

Vector becomes larger than L1 Cache (64KB)

Vector becomes larger than L2 Cache (512KB)
What happens when we fill cache?

• Several methods to decide which cache line will be replaced:
  • Least Recently Used
  • Most Recently Used
  • Least Frequently Used
• Method is controlled by sysadmin.
• All of these methods still require us to search the entire cache for a datum. Is there a better way?
Set Associativity

- If any cache line can be placed in any cache index, the cache is fully associative.
- All of the cache must be searched for any datum.

<table>
<thead>
<tr>
<th>Main Memory</th>
<th>Cache Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Index</strong></td>
<td><strong>Index</strong></td>
</tr>
<tr>
<td><strong>Data</strong></td>
<td><strong>Tag</strong></td>
</tr>
<tr>
<td>0  xyz</td>
<td>0  2  abc</td>
</tr>
<tr>
<td>1  pdq</td>
<td>1  0  xyz</td>
</tr>
<tr>
<td>2  abc</td>
<td></td>
</tr>
<tr>
<td>3  rgf</td>
<td></td>
</tr>
</tbody>
</table>
Set Associativity

Direct Mapped Cache Fill

Main Memory

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

Cache Memory

<table>
<thead>
<tr>
<th>Index</th>
<th>Index 0</th>
<th>Index 1</th>
<th>Index 2</th>
<th>Index 3</th>
</tr>
</thead>
</table>

Each location in main memory can be cached by just one cache location.

2-Way Associative Cache Fill

Main Memory

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
</table>

Cache Memory

<table>
<thead>
<tr>
<th>Index</th>
<th>Index 0, Way 0</th>
<th>Index 0, Way 1</th>
<th>Index 1, Way 0</th>
<th>Index 1, Way 1</th>
</tr>
</thead>
</table>

Each location in main memory can be cached by one of two cache locations.
Set Associativity

• The less associative a cache is, the fewer cache lines we need to search.
• Tradeoff is that miss rates will rise.
Taking Advantage of Cache

• If we can change our code so contiguous data in a cache line is accessed in sequence (stride 1) then we maximise our cache hits.
  • Average data access times will fall.
  • Code will run faster!
Taking Advantage of Cache

• A simple Fortran example, $x$ is a $mxn$ array:

$$
\text{do } i=1, m, 1 \\
\hspace{1cm} \text{do } j=1, n, 1 \\
\hspace{2cm} x[i][j]=x[i][j]+1.0 \\
\hspace{1cm} \text{end do} \\
\text{end do}
$$

• In Fortran, data is stored in a column-wise manner.
Taking Advantage of Cache

• If \( x[0][0] \) is the first element stored in memory, then \( x[1][0] \) is the second.
• Cache lines will run down columns
• Using \( j \) as the inner loop means we access \( x[0][0] \) then \( x[0][1] \).
• This yields many cache misses.
• Solution?
Taking Advantage of Cache

- Interchange the loops:

```plaintext
do j=1, n, 1
  do i=1, m, 1
    x[i][j]=x[i][j]+1.0
  end do
end do
```

- Semantically equivalent, but many more cache hits.
Taking Advantage of Cache

- For large enough values of $n$, no cache hits will occur for the first example.
- The cache will become full and lines will be replaced before the next element in them is needed.
- The example was run on an older PC in the CS department with 512KB of cache with $n=20000$, $m=2000$.
  - Unoptimised loop took 0.5989 secs.
  - Optimised (stride 1) loop took 0.09 secs.
Taking Advantage of Cache

- A more useful example, matrix-matrix multiplication:

\[
\begin{align*}
  &\text{do } i=1, n, 1 \\
  &\quad \text{do } j=1, n, 1 \\
  &\quad\quad \text{do } k=1, n, 1 \\
  &\quad\quad\quad c[i][j] = c[i][j] + a[i][k]*c[k][j] \\
  &\quad\text{end do} \\
  &\text{end do} \\
  &\text{end do}
\end{align*}
\]
Taking Advantage of Cache

• Exchange loops to maximise cache hits:

\[
\begin{align*}
&\text{do } j=1, n, 1 \\
&\quad \text{do } k=1, n, 1 \\
&\quad \quad \text{do } i=1, n, 1 \\
&\quad \quad \quad c[i][j] = c[i][j] + a[i][k] \times c[k][j] \\
&\quad \quad \text{end do} \\
&\quad \text{end do} \\
&\text{end do}
\end{align*}
\]
Conclusions

• Knowledge of cache can help to optimise your code.
• Simple changes to loops can yield much better cache performance.
• Next time – compiler optimisations? Further loop optimisations?