Performance, Caches and Virtual Memory

Computer Architecture

1DT016 distance
Fall 2017

http://xyx.se/1DT016/index.php

Per Foyer
Mail: per.foyer@it.uu.se
Performance Measures

- **CPI**: Cycles Per Instruction (average over all instructions in some set)
  - Measured in... (surprise) cycles / instruction
- **IPC** (not Interprocess Communication): Instructions per cycle
- **Instruction Count (IC)**: Instruction Count
- **Clock Cycle Time (CT)**: Time between rising edges of the clock (= the period)
  - Measured in seconds (or other units of time) per cycle
- **Clock Rate (CR)**: Number of cycles per second
  - Measured in cycles per second or Hz
- **MIPS** (not the CPU): old measure
  - Million Instructions Per Second (Misleading measurement)
MIPS as a Performance Metric

- **Millions of Instructions Per Second**
- **Doesn’t account for:** *(think CISC vs RISC)*
  - Differences in ISA between computers
  - Differences in complexity between instructions

\[
\text{MIPS} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6} = \frac{\text{Instruction count}}{\text{Instruction count} \times \text{CPI} \times 10^6} \times \frac{\text{Clock rate}}{\text{CPI} \times 10^6}
\]

- **CPI varies between programs on a given CPU**
Decreasing execution time

We want to make execution time (ET) as short as possible!

\[ ET = IC \times CPI \times CT \]
Improving CPI

- Many processor design techniques improve CPI
  - Often they only improve CPI for certain types of instructions

\[
\text{CPI} = \sum_{i=1}^{n} \text{CPI}_i \times F_i \quad \text{where} \quad F_i = \frac{l_i}{\text{Instruction Count}}
\]

- \( F_i \) = Fraction of instructions of type \( i \)

First Law of Performance:
Make the common case fast
Amdahl’s Law

- Amdahl’s Law states that optimizations are limited in their effectiveness

\[
\text{Execution time after improvement} = \frac{\text{Time affected by improvement}}{\text{Amount of improvement}} + \text{Time unaffected by improvement}
\]

- For example, doubling the speed of floating-point operations sounds like a great idea. But if only 10% of the program execution time T involves floating-point code, then the overall performance improves by just 5%.

\[
\frac{\text{Execution time after improvement}}{2} + 0.90T = 0.95T
\]

Second Law of Performance:
Make the fast case common
Reducing CPI

• Application writers choose instructions that take fewer cycles?
• Compilers choose instructions that take fewer cycles?
• ISA + microarchitecture defines only simpler instructions that take few clock cycles to complete?
Reducing IC

• Application writers write **concise code**?
• Compiler optimizes code and **eliminates instructions** while maintaining behavior?
• ISA defines many **instructions to do complex actions in a single instruction**?

RISC vs CISC?
Reducing IC: Tradeoffs

• Writing optimized code or writing optimizing compilers add complexity and make more work for program and compiler writers.

• Writing better code is not a reliable source of improvement for every program because significant reductions cannot always be found.

• More complex instructions add complexity to microarchitecture.

• Performance not always improved by having complicated instructions.
Reducing CT

• Increase clock rate? (perhaps overclocking)

But…

• Clock can only go as fast as slowest component.
• Power wall – Too much heat.
• Memory wall – processor may be running faster, but overall performance may not improve due to memory latency bottleneck.
• May increase CPI.
Reducing Power

• Suppose a new CPU has
  • 85% of capacitive load of old CPU
  • 15% voltage and 15% frequency reduction

\[
\frac{P_{\text{new}}}{P_{\text{old}}} = \frac{C_{\text{old}} \times 0.85 \times (V_{\text{old}} \times 0.85)^2 \times F_{\text{old}} \times 0.85}{C_{\text{old}} \times V_{\text{old}}^2 \times F_{\text{old}}} = 0.85^4 = 0.52
\]

• The power wall
  • We can’t reduce voltage any further \((V_{\text{new}} < V_{f_{\text{MOS}}})\)
  • We can’t remove more heat
The storage pyramid

High performance
High cost

Low performance
Low cost

Storage capacity
## Latency numbers to know

(Give or take)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>1 ns</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>3 ns</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>4 ns</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>17 ns</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100 ns</td>
</tr>
<tr>
<td>Send 2KB over commodity network</td>
<td>250 ns</td>
</tr>
<tr>
<td>Compress 1KB with zip</td>
<td>2 us</td>
</tr>
<tr>
<td>Read 1MB sequentially from main memory</td>
<td>9 us</td>
</tr>
<tr>
<td>SSD random read</td>
<td>16 us</td>
</tr>
<tr>
<td>Read 1MB sequentially from SSD</td>
<td>156 us</td>
</tr>
<tr>
<td>Round trip in datacenter</td>
<td>500 us</td>
</tr>
</tbody>
</table>
Memory

• The Memory Hierarchy: The pyramid picture.

• The top of the pyramid is registers while RAM is several tiers lower.

• Unacceptable to have the clock cycle be 70 ns while most of the stages are several orders of magnitude faster than that!

• To have a viable pipeline, we need a faster way to access memory.

• …Easier said than done (memory wall).
Simple computer

Let's say:
- RISC CPU with 1 GHz clock (cycle time 1 nS → CPI = 1)
- Memory response time for load / store: 70 nS
- A program where 50% of instructions are load/store

CPU utilization? How much performance increase if CPU clock is doubled?

~50%
But… no…
Advanced pipelining (1)

How do we improve the performance of the pipeline?

• Start by reevaluating:
  \[ ET = IC \times CPI \times CT \]

• How do we reduce ET?
  - The microarchitecture can only really influence CPI and CT?

• The pipelined datapath is preferred over the single cycle datapath because of the significant reduction of CT.

• In general, the CT can be reduced by splitting the datapath into more stages.

We can decrease the CT by increasing the number of stages AKA having a “deeper pipeline”.

Per.Foyer@it.uu.se 2017
Advanced pipelining (2)

- Typically we've been using cycles per instruction. How could an instruction take fewer than 1 cycles?
- Think of the inverse unit of measure, instructions per cycle (IPC, not Interprocess Communication)
- If we could complete more than one instruction in one cycle (for example, if we had two full pipelines), we could complete 2 IPC.
- This would correspond to a CPI of .5!

Theory:
More pipeline stages means lower clock time.
- The bottleneck here is the MEM stage.
Caches: Intro

Let’s say we have

• "The perfect” microarchitecture (ISA)
• Reduced CT to a minimum
• Introduced advanced, perhaps parallel, pipelines
• Have compilers that optimizes code and minimizes pipeline hazards
• …we still have the memory problem…

• Cache: A small(er) storage unit that holds a subset of memory that is much faster than accessing main memory.
Principle of Locality

• Usually difficult or impossible to figure out what data will be “most frequently accessed” before a program actually runs.
  • makes it hard to know what to store into the small, precious cache memory.

• In practice, most programs exhibit locality, which the cache can take advantage of.
  • The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again.
  • The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses.
To take advantage of temporal locality

- The first time the processor reads from an address in main memory, a copy of that data is also stored in the cache.
  - The next time that same address is read, we can use the copy of the data in the cache instead of accessing the slower dynamic memory.
  - So the first read is a little slower than before since it goes through both main memory and the cache, but subsequent reads are much faster.
- Takes advantage of temporal locality — commonly accessed data is stored in the faster cache memory.
Caches and spatial locality

• When the CPU reads location $i$ from main memory, a copy of that data is placed in the cache.

• But instead of just copying the contents of location $i$, we can copy several values into the cache at once, such as the four bytes from locations $i$ through $i + 3$.

  ▪ If the CPU later does need to read from locations $i + 1$, $i + 2$ or $i + 3$, it can access that data from the cache and not the slower main memory.

  ▪ For example, instead of reading just one array element at a time, the cache might actually be loading four array elements at once.

• Again, the initial load incurs a performance penalty, but we’re gambling on spatial locality and the chance that the CPU will need the extra data.
Temporal locality in programs

• The principle of temporal locality says that if a program accesses one memory address, there is a good chance that it will access the same address again. *Already said*

• Loops are excellent examples of temporal locality in programs.
  ▪ The loop body will be executed many times.
  ▪ The computer will need to access those same few locations of the instruction memory repeatedly.

For example:

```
Loop: lw $t0, 0($s1)
      add $t0, $t0, $s2
      sw $t0, 0($s1)
      addi $s1, $s1, -4
      bne $s1, $0, Loop
```

• Each instruction will be fetched over and over again, once on every loop iteration.
Temporal locality in data

• Programs often access the same variables over and over, especially within loops. Below, `sum` and `i` are repeatedly read and written.

```c
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + f(i);
```

• Commonly-accessed variables can sometimes be kept in registers, but this is not always possible.
  • There are a limited number of registers.
  • There are situations where the data must be kept in memory, as is the case with shared or dynamically allocated memory.
Spatial locality in programs

• The principle of spatial locality says that if a program accesses one memory address, there is a good chance that it will also access other nearby addresses. *(already said)*

```
sub  $sp, $sp, 16
sw   $ra, 0($sp)
sw   $s0, 4($sp)
sw   $a0, 8($sp)
sw   $a1, 12($sp)
```

• Nearly every program exhibits spatial locality, because instructions are usually executed in sequence — if we execute an instruction at memory location $i$, then we will probably also execute the next instruction, at memory location $i+1$.

• Code fragments such as loops exhibit *both* temporal and spatial locality.
Spatial locality in data

- Programs often access data that is stored contiguously.
  - Arrays, like `a` in the code on the top, are stored in memory contiguously.
  - The individual fields of a record or object like `employee` are also kept contiguously in memory.

```
sum = 0;
for (i = 0; i < MAX; i++)
    sum = sum + a[i];
```

```
employee.name = "Homer Simpson";
employee.boss = "Mr. Burns";
employee.age = 45;
```

Can data have both spatial and temporal locality?
Simple computer with cache

Let’s say:
• CPU with 1 GHz clock (cycle time 1 nS → CPI = 1)
• Cache response time for load / store: 1 nS (3 cycles)
• Memory response time for load / store: 100 nS (100 cycles)
• A program where 50% of instructions are load/store

CPU utilization? How much performance increase if CPU clock is doubled?
Cache Memories

Cache Accesses (10 or fewer cycles) 
Main Memory Access (100s of cycles) 
Disk Access (100,000s of cycles)

Cache Req 
Cache Resp 
Cache Hit 
Cache Miss 
Cache Miss 
Cache Miss 
Page Fault (SW Involved)
An array with \( n \) elements indexed 0 to \( n-1 \):

Store a number \( x \) in the hash table:
\[ \text{Index} = x \mod n \]

Example for 16 slots:
Store 113, 355 and 6174:
\[ \begin{align*}
\text{Index1} &= 113 \mod 16 = 1 \\
\text{Index2} &= 355 \mod 16 = 3 \\
\text{Index3} &= 6174 \mod 16 = 14
\end{align*} \]

Complexity:
\[ t_{\text{mod}} + t_{\text{store}} = 2 \Rightarrow O(1) \]

…but what shall we do after this to store 225 in the structure? (Collision)
Intro: A hashed datastructure (2)

An array with $n$ elements indexed 0 to $n-1$:

Store a number $x$ in the hash table:
Index = $x \mod n$

Example for 16 slots in two sets:
Store 225, 1022 and 98765:

Index = 225 mod 16 = 1 (collision)
Index = 1022 mod 16 = 14 (collision)
Index = 98765 mod 16 = 13

$\text{idx} = x \mod n$
if set1[$\text{idx}$] is empty then
store $x$ at set1[$\text{idx}$]
else
if set2[$\text{idx}$] is empty then
store $x$ at set2[$\text{idx}$]
else ...
...

Complexity: $t_{\text{mod}} + t_{\text{miss}} + t_{\text{store}} = 3 \to O(1)$
Cache policies

• Single line (1-way) cache
• Multiline (multi-way) cache
• Replacement policies
• Write policies
• Categorizing misses
Cache Hierarchies

• Trade-off between access time & hit rate
  • L1 cache can focus on fast access time (okay hit rate)
  • L2 cache can focus on good hit rate (okay access time)

• Is there a way to boost main memory performance?
  • Memory interleaving
  • Burst read
Definitions: Hits and misses

• A cache hit occurs if the cache contains the data that we’re looking for. Hits are good, because the cache can return the data much faster than main memory.

• A cache miss occurs if the cache does not contain the requested data. This is bad, since the CPU must then wait for the slower main memory.

• There are two basic measurements of cache performance.
  • The hit rate is the percentage of memory accesses that are handled by the cache.
  • The miss rate (1 - hit rate) is the percentage of accesses that must be handled by the slower main RAM.

• Typical caches have a hit rate of 95% or higher, so in fact most memory accesses will be handled by the cache and will be dramatically faster.
High-Level Idea for a Cache

Check Tag → read hit → Access Data
read miss → Select Victim → Refill → write hit
write → Write Mem → read miss

FSM:
- Check Tag → read hit
- Access Data
- Check Tag → write hit
- Access Data
- Check Tag → read hit
- Access Data
- Check Tag → read hit

Refill
- Access Data
- Check Tag
- Access Data
Direct mapped cache

- When the CPU tries to read from memory, the address will be sent to a cache controller.
  - The lowest k bits of the address will index a block in the cache.
  - If the block is valid and the tag matches the upper \((m-k)\) bits of the m-bit address, then that data will be sent to the CPU.
- Diagram of a 32-bit memory address and a \(2^{10}\)-byte cache:
Loading a block into the cache

After data is read from main memory:

• The lowest k bits of the cache specifies a cache block
• The upper (m - k) address bits are stored in the block’s tag field
• The data from main memory is stored in the block’s data field.
• The valid bit is set to 1
Single-line cache

Consider only 4 byte word accesses and only the read path for three single-line cache designs:
### Four-line direct mapped

Four-line direct-mapped cache with 4 byte cache lines:

<table>
<thead>
<tr>
<th>0x000</th>
<th>0x004</th>
<th>0x008</th>
<th>0x00c</th>
<th>0x010</th>
<th>0x014</th>
<th>0x018</th>
<th>0x01c</th>
<th>0x020</th>
<th>0x024</th>
</tr>
</thead>
</table>

![Diagram of the cache](image)

- **Tag**: 28 bits
- **Index**: 2 bits
- **Data**: 32 bits
- **Sets**: 4 sets

Hit determination: If the tag matches, it is a hit.
Four-line two-way set-associative

Four-line two-way set-associative cache with 4 byte cache lines
Four-line fully-associative cache with 4 byte cache lines:
The price of full associativity

A fully associative cache is expensive to implement.  

- There is no index field in the address, the entire address must be used as the tag, increasing the total cache size.  
- Data could be anywhere in the cache, so check the tag of every cache block must be checked.  
  → A lot of comparators!
Associativity Tradeoffs

• What are the benefits of using a Direct Mapped Cache?
  • Easy to know exactly where a block should belong and what block should be evicted

• What's the downside?
  • Thrashing and the consequent poor performance when a cache is not used well

• What's the benefit of having increased associativity?
  • The problem of thrashing is reduced. Potentially better use of the cache

• Downside?
  • Hardware complexity increases.
  • In n-way associative, must search multiple entries
  • Consider the cost of searching a fully associative cache.
Write-Hit (write-)policies (1)

• What do we do when we write to memory and the address in question is in the cache?

• **Write through**
  - For each write-hit, write to the cache and also to the backing memory.

• **Write back**
  - For each write-hit, write only to the cache but mark the cache block as dirty. Then if that dirty block is evicted, write it back to memory.
Write-Hit (write-)policies (1½)

Write through

[Diagram of memory request, request type, cache hit, locate cache block, read from lower memory, return data, write into cache block, write into lower memory, done]
Write-Hit (write-)policies (2)

• Benefits of Write-Through?
  • Memory and cache are always synchronized and consistent.

• Downsides?
  • Have to incur a severe penalty every single time data is written.
  • Or maybe not? A write buffer can be used in conjunction with a write-through cache. Write to the write buffer after which the processor is allowed to continue while the buffer writes to main memory.
Write-Hit (write-)policies ($2^{1/2}$)

Write back

1. **Memory request**
2. **Request type**
3. **Cache hit?**
   - Yes
     - Is it 'dirty'?
       - Yes: Write its previous data back to the lower memory
       - No: Read data from lower memory into the cache block
       - Mark the cache block as 'not dirty'
       - Return data
     - No: Locate a cache block to use
4. **Cache hit?**
   - Yes
     - Is it 'dirty'?
       - Yes: Write its previous data back to the lower memory
       - No: Read data from lower memory into the cache block
       - Mark the cache block as 'dirty'
   - No: Locate a cache block to use

Done
Benefits of Write-Back?

Write-through requires 1 write to memory for every store instruction. Write-back effectively collects all of the writes to a particular block and needs to perform one write to memory. This is much better compared to write through if data is frequently written to the same block.

• Downsides?
  • Write-through pays the cost of writing to memory for stores, but write-back may mean paying the cost of writing to memory for loads.
  • Load performance may be more critical.
Eviction policies

• So... what to do when the cache is full?
• Under the assumption that all memory accesses must first go through the cache, this means that something needs to be thrown out of the cache.
• How do we decide what to throw out?
Eviction/Replacement policies (1)

- Random
  - Simple
  - As likely to evict a useful block as it is to throw out a useless one.
- LRU (Least Recently Used)
  - “The block that has been used least recently is the one that is least likely to be used again” (temporal locality)
  - Complicated and costly to do perfectly accurately (requires updating every block in set in access)
Eviction/Replacement policies (2)

- **FIFO** (First in First out):
  - Approximates LRU with less complexity.

- **MRU** (Most recently used):
  - Extremely simple and fast.
  - Not useful if trying to utilize locality.

- **NMRU** (Not most recently used):
  - Randomly evict any block that's not the most recently used.
  - “Approximates” LRU.
Split caches

We already know:

• Harvard architecture:
  Separate program and data memories

• von Neuman architecture:
  Same memory for both program and data

So, what about caches here?

• Use two caches for the Harvard architecture (split cache).
  • Program: Temporal locality
  • Data: Both temporal and spatial locality
Example: AMD Opteron

von Neumann multi-core

• L1 Caches: Instruction & Data
  • 64 kB
  • 64 byte blocks
  • 2-way set associative
  • 2 cycle access time

• L2 Cache:
  • 1 MB
  • 64 byte blocks
  • 4-way set associative
  • 16 cycle access time (total, not just miss penalty)

• Memory:
  • 200+ cycle access time
Virtual memory (1)

Running multiple programs at the same time brings up problems:

1. Even if each program fits in (main) memory, running 10 programs might not

2. Multiple programs may want to store something at the same address in (main) memory

3. How to protect a program’s data from being read or written by another program?

We need to be able to:

- “Swap” data in and out from memory to disk
- Have Hardware support for memory sharing and protection
Virtual memory (2)

Some facts:

• Virtual memory is handled by an MMU (Memory management Unit)
• The MMU may be CPU internal or external
• Virtual memory is completely transparent to programs
• If (main) memory is full, too fragmented or information not in (main) memory, information will be written or read from disk
• Straight forward for the vN architecture but needs some trickery for the Harvard architecture

Virtual memory is not only about hardware!
It needs supporting software: An Operating System
Virtual memory (3)

- Translate “virtual addresses” used by the program to “physical addresses” that represent places in the machine’s “physical” memory.
  - The word “translate” denotes a level of indirection

A virtual address can be mapped to either physical memory or disk
Virtual memory (4)

• Because different processes will have different mappings from virtual to physical addresses, two programs can freely use the same virtual address.

• By allocating distinct regions of physical memory to A and B, they are prevented from reading/writing each other's data.
Memory protection

• In order to prevent one process from reading/writing another process’s memory, we must ensure that a process cannot change its virtual-to-physical translations.

• Typically, this is done by:
  • Having two processor modes: user & kernel.
    • Only the O/S runs in kernel mode
  • Only allowing kernel mode to write to the virtual memory state:
    • The page table
    • The page table base pointer
    • The TLB
Sharing memory

• Paged virtual memory enables sharing at the granularity of a page, by allowing two page tables to point to the same physical addresses.

• For example, if two copies of a program is run, the O/S will share the code pages between the programs.
Finding the right page

• If it is fully associative, how do we find the right page **without scanning** all of memory?
  • Use an **index**

• The index happens to be called the **page table**:
  • Each process has a separate page table
    • A “page table register” points to the current process’s page table
  • The page table is indexed with the **virtual page number (VPN)**
    • The VPN is all of the bits that aren’t part of the page offset.
  • Each entry contains a valid bit, and a **physical page number (PPN)**
    • The PPN is concatenated with the page offset to get the physical address
  • No tag is needed because the index is the full VPN.
Page table picture

```
Page table register

Virtual address
31 30 29 28 27 ························ 15 14 13 12 11 10 9 8 ····· 3 2 1 0

Virtual page number  Page offset

Valid
20

Physical page number

Page table

If 0 then page is not present in memory

29 28 27 ························ 15 14 13 12 11 10 9 8 ····· 3 2 1 0

Physical page number  Page offset

Physical address
```
Virtual memory caching

• Once the translation infrastructure is in place, the problem boils down to caching.
  • We want the size of disk, but the performance of memory.
• The design of virtual memory systems is really motivated by the high cost of accessing disk.
  • While memory latency is \(\sim 100\) times that of cache, disk latency is \(\sim 100,000\) times that of memory.
• Hence, try to minimize the miss rate:
  • VM “pages” are much larger than cache blocks. Why?
  • A fully associative policy is used.
    • With approximate LRU

Should a write-through or write-back policy be used?
Disk Caching Translations

Virtual to Physical translations are cached in a Translation Lookaside Buffer (TLB).

Diagram showing the process of virtual to physical translation with a TLB hit and subsequent steps involving cache and page offset calculations.
Handling a TLB miss

• If a miss in the TLB, there is a need to “walk the page table”
  • In MIPS, an exception is raised and software fills the TLB
  • In x86, a “hardware page table walker” fills the TLB

• What if the page is not in memory?
  • This situation is called a page fault.
  • The operating system will have to request the page from disk.
  • It will need to select a page to replace.
    • The O/S tries to approximate LRU
    • The replaced page will need to be written back if dirty.

• If TLB in main memory, it may be transferred or evicted from the memory cache.
  • Use a separate TLB (memory) cache for speed
Dealing with larger page tables

- Multi-level page tables
  - “Any problem in CS can be solved by adding a level of indirection” :-D
  - or two…

Since most processes don’t use the whole address space, there is no need to allocate the tables that aren’t needed.

- Also, the 2nd and 3rd level page tables can be “paged” to disk.
Virtual memory: summary

• Virtual memory is great:
  • It means that we don’t have to manage our own memory.
  • It allows different programs to use the same memory.
  • It provides protect between different processes.
  • It allows controlled sharing between processes (albeit somewhat inflexibly).

• The key technique is indirection:
  • Many problems can be solved with indirection. 😊

• Caching made a few appearances, too:
  • Virtual memory enables using physical memory as a cache for disk.
  • Caching was used (in the form of the Translation Lookaside Buffer) to make Virtual Memory’s indirection fast.
Häpp! Finito la musica!