

# System Architecture

Todd Davies

January 29, 2015

## Introduction

The basic architecture of computer systems has been covered in first year course units which detailed both the instruction set architecture and the micro- architecture (hardware structure) of simple processors. Although these principle underlie the vast majority of modern computers, there are a wide range of both hardware and software techniques which are employed to increase the performance, reliability and flexibility of systems.

## Aims

The aims of this course are to introduce the most important system architecture approaches. To give a wider understanding of how real systems operate and, from that understanding, the ability to optimise their use.

The syllabus includes:

- The motivation behind advanced architectural techniques.
- Caching
- The need to overcome latency. Caching as a principle, examples of caching in practice. Processor cache structure and operation.
- Pipelining
- Principles of pipelining. Implementation of a processor pipeline and its properties. Pipelining requirements and limitations. Additional support for pipelining.
- Multi-Threading
- Basic multi-threading principles. Processor support for multi-threading. Simultaneous multi-threading.
- Multi-Core
- Motivation for multi-core. Possible multi-core structures. Cache coherence.
- File System Support
- Implementation of file systems. RAID
- Virtual Machines
- Motivation for Virtual Machines. Language Virtual Machines. System Virtual Machines. Virtual Machine implementation. Binary Translation

# Contents

<b>Contents</b>		2.3.3 Set associative . . . . .	5
<b>1 Introduction</b>	<b>3</b>	<b>3 Pipelines</b>	<b>5</b>
<b>2 Caches</b>	<b>3</b>	<b>4 Multi-Threading</b>	<b>5</b>
2.1 Why are caches expensive? . . . . .	3	<b>5 Multi-Core</b>	<b>5</b>
2.2 L1 Cache . . . . .	3	<b>6 Virtualization</b>	<b>5</b>
2.3 Types of cache . . . . .	4	<b>7 Permanent Storage</b>	<b>5</b>
2.3.1 Fully associative . . . . .	4		
2.3.2 Directly mapped . . . . .	5		

# 1 Introduction

Performance is always an attribute in high demand in computer systems. Even though processors have become so much more powerful over the last half century, there's still loads of stuff that we cannot do with current technology, such as synthesising HD video in realtime, or computing realistic game physics.

Since 2004/5, companies haven't been able to increase the speed of microprocessors at such a rapid rate due to physical limits, such as power dissipation and device variability. Our devices are still getting faster, but now architecture and the design of systems play a larger role in making stuff run faster. An example of this include making computation more parallel.

## 2 Caches

Not all technology has improved at the same relative speed. CPU's have become over three orders of magnitude faster over the past thirty years, but memory has increased by only one order of magnitude. This is problematic, since it means that we need to reconcile this gap in order to achieve efficient computation.

Processor caching is used to let the processor do useful computation while it's also waiting on the memory. Modern processors couldn't perform anywhere near how fast they do now without equally modern caching techniques, since the imbalance between the CPU and main memory is so high.

Caches (in general) provide a limited, but very fast local space for the CPU to use. They are used in lots of places all over computer science, including web browsers, mobile phone UI's etc. Likewise, a processor cache is a temporary store for frequently used memory locations.

The principle of locality is what makes caches work for processors, which is that the CPU will only use a small subset of memory over a short period of time. If this subset of memory can be loaded into the cache, then the computation can be sped up significantly.

Every 'cache miss' takes *at least* sixty times longer to execute than a 'cache hit' will (that's assuming there are no page faults etc). Circuit capacitance is the thing that makes electronic devices slow, and larger components have a larger capacitance, henceforth large memories are slow. Dynamic memories (DRAM) store data using capacitance, and are therefore slower than static memories (SRAM) that work using bistable circuits.

Even the wires between the processor and the memory have a significant capacitance. Driving signals between chips needs specialised high power interface circuits. An ideal situation would be to have everything on a single chip, however current manufacturing limitations prevent this; maybe one day we will be able to do this.

### 2.1 Why are caches expensive?

L1, L2 and (usually) L3 caches are SRAM instead of DRAM (which is what main memory is made from).

SRAM needs six transistors per bit, DRAM needs one.

SRAM is henceforth physically larger, taking up more space on the chip, which is expensive, since real estate costs money.

### 2.2 L1 Cache

The L1 cache is the first level of caching between the processor and the main memory. The L1 cache is around 32kb, which is very small in comparison to the size of the main memory, but this is driven out of necessity, since the cache needs to be small to be fast. The cache must be

able to hold any arbitrary location in the main memory (since we don't know in advance what the CPU will want), and henceforth requires specialised structures to implement this.

## 2.3 Types of cache

The cpu will give the cache a full address of a word in memory that it wants to read. The cache will contain a small selection of values from memory that it has locally, but will ask the main memory for values that it does not have. This is called a cache miss and is expensive in comparison to a cache hit.

### 2.3.1 Fully associative

A **Fully Associative** cache is one where the cache is small (around 32,000 values), but stores both addresses and their corresponding data. The hardware compares the input address with all of the stored addresses (it does this in parallel). If the address is found, then the value is returned with no need to ask the RAM (cache hit), if the value isn't found, then a cache miss occurs, and the request must go to the main memory.

Caches rely on locality in order to function effectively. There are two types of locality; temporal locality, which is the principle that if you use an address once, you may use it again soon (e.g. loops), and spatial locality, where if you use an address once, you are also likely to use addresses nearby (e.g. arrays).

The cache hit rate is the ratio of cache hits to misses. We need a hit rate of 98% to hide the speed of memory from the CPU. Instruction hit rates are usually better than data hit rates (although they are in the same cache remember). The reason for this is that instructions are accessed in a more regular pattern, incrementing by one word every time, or looping around etc (have higher locality).

Spatial locality is exploited better by having a bigger data area in the cache (returning say 512 bits for every address instead of just one word)

When we do a cache miss (read), we should add the missed value to the cache. In order to do this, we need a cache replacement policy to find room in the cache to put the new value:

- LRU - slow, good for hit rates
- Round Robin - not as good, easier to implement
- Random - Easy to implement, works better than expected.

Memory writes are more complicated than reads. If we've already got the value in the cache, then we change the value in the cache. We can use three write strategies for cache writes (on hits):

- Write through (slow)
- Write through + buffer (faster, slow when heavily used)
- Copy back on cache replacement.

On misses:

- We can find a location in the cache, and write to that, then rely on copy back later, or write back straight away.
- we can skip the cache, and write directly to RAM. Subsequent read will add to the cache if necessary (good if you're initialising datastructures with zeroes).

Fastset one is write allocate or copy back. Main memory and the cache aren't coherent, which can be a problem for stuff like multiprocessors, autonomous IO devices etc. This may need cleaning up later.

Each cache line is at least half address and half data, but often, we store more data per address, so will have 64 bytes of data per 32 bit address.

A fully associative cache is ideal, but this is expensive (in terms of silicon and power).

### **2.3.2 Directly mapped**

We can use standard RAM to create a directly mapped cache, which mimics the functionality of an ideal cache. Usually, this uses static RAM, which is more expensive than dynamic RAM, but is faster. The address is divided into two parts,

### **2.3.3 Set associative**

Set associative caches are a compromise. They comprise of a number of directly mapped caches operating in parallel. If one matches, we have a hit and select the appropriate data. This is good because we can have more flexible cache replacement strategies. In a 4 way, we could choose any one of the four caches for example. The hit rate of set associative caches improves with the number of caches, but increasing the number increase the cost.

## **3 Pipelines**

## **4 Multi-Threading**

## **5 Multi-Core**

## **6 Virtualization**

## **7 Permanent Storage**