CACHES, MEMORY SYSTEMS AND ALGORITHMS
A VISICACHE CLASS AT THE UNIVERSITY OF CAMBRIDGE

You should read this handout and understand the theory *before* you arrive for the laboratory session. Otherwise, you will find it very difficult to complete the experiment in the allotted time.

1 Introduction

The main components of a modern computer system are the central processing unit (CPU), the memory system and the input/output devices. As CPUs get faster and faster, it is important that the performance of the memory system keeps pace: otherwise, the speed of the overall system will be compromised by the memory system bottleneck. A key component of the memory system is the *cache*, a small, fast memory which sits between the CPU and the main memory. This experiment will investigate the relative strengths and weaknesses of different cache configurations, and demonstrate how the speed of common software procedures, like sorting and matrix multiplication, depends not only on the procedure's algorithmic complexity, but also on the way the algorithm interacts with the cache.

Aims and objectives

- To appreciate the significant role played by the cache in a modern computer system.
- To explore the relative strengths and weaknesses of direct mapped, set associative and fully associative caches.
- To understand the role of the programmer in writing cache-friendly software.
- To appreciate the dominant role of algorithmic complexity in software performance.
- To compile and execute programs from the command line in a Unix environment; to understand the effects of compiler optimisation flags.

2 Theory

Memory systems and caches

Memory references occur frequently during the execution of most programs. For example, consider the C++ statement `a[i]++`. This might be translated by the compiler into three machine code instructions, one to load element `i` of array `a` from memory into a CPU register, another to increment the contents of the register, and another to store the contents of the register back into memory. So even a simple instruction like this generates two memory references. Since main memory access is slow compared with the speed of the CPU, the CPU might have to stall twice, waiting for the memory system to deliver and receive the data. Such behaviour would have a significant detrimental effect on the overall performance of the computer system.

For this reason, all modern computer systems make use of a cache, a small, fast memory which can be accessed at full CPU speed. The cache's speed owes much to its construction out of static RAM (SRAM), which is much faster than the main memory's dynamic RAM (DRAM), but also much more bulky and expensive. Another reason why the cache is so fast is its location: it is often implemented on the same chip as the CPU, so there is no need for slow bus transfers. Typical cache sizes are of the order of 1 MByte, compared with perhaps 256 MBytes for the main memory. So the cache can only store a subset of what's in the main memory at any given time. However, if the cache is well designed and the software well written, the CPU should be able to find the data it needs at any given time in the cache: only occasionally should it be necessary to access the

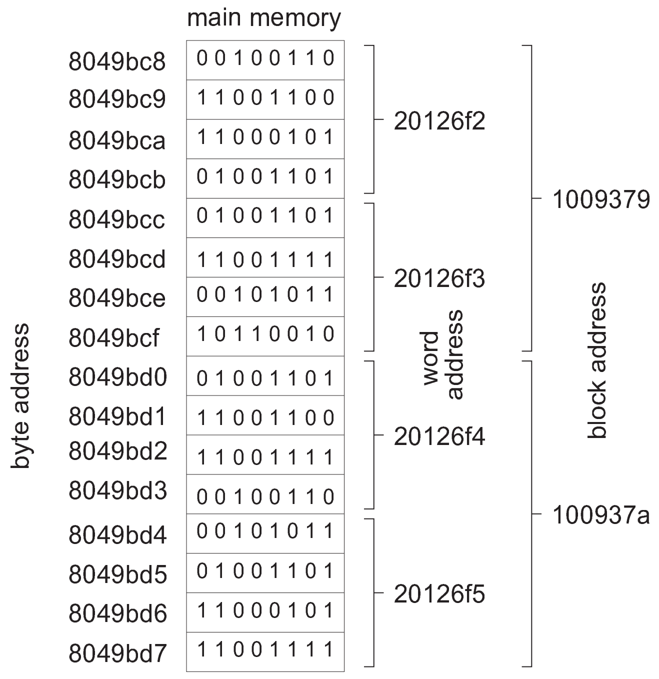

Figure 1: **Memory addresses**. Computer memories are byte addressable: each address corresponds to a group of 8 bits in memory. In this example, we see a segment of memory 16 bytes long, starting at hexadecimal address `8049bc8` and ending at `8049bd7`. Since most current CPUs manipulate 32 bits of data at a time, an alternative way to picture the memory is as a sequence of four-byte words, each with its own word address. Furthermore, caches load words one block at a time, where a block usually comprises several words (two in this example), so we can also consider the memory as a sequence of blocks, each with its own block address.

main memory. This experiment will explore what is meant by a “well designed” cache and “well written” software.

Caches exploit two important properties of memory references:

Spatial locality of reference: if a data item is referenced, neighbouring data items will tend to be referenced soon: eg. instruction fetching and data in arrays.

Temporal locality of reference: if a data item is referenced, the same item will tend to be referenced again soon: eg. instruction loops and local variables.

Temporal locality of reference implies that if the CPU has to fetch a data item from main memory and store it in the cache, the same item will be referenced again soon, and this time the CPU will find it in the cache, without having to wait for a lengthy main memory access. Spatial locality of reference suggests that the CPU should fetch not only the item it needs right now, but also the item’s immediate neighbours, since they are likely to be needed soon. Since main memory transfers involve a significant, fixed overhead, transferring n items into the cache does not take n times longer than transferring just one item. These are the two principles upon which all caches are founded: the software must also play its part by ensuring, wherever possible, that memory accesses *do* exhibit locality of reference, and are not totally random.

Bytes, words and blocks

Up until now, we’ve been thinking about moving “items” between the main memory and the cache. What precisely do we mean by an item? Most computers these days conform to a 32-bit

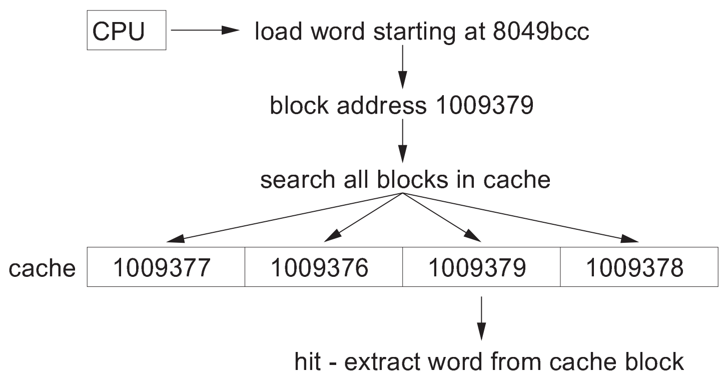
CPU
load word starting at 8049bcc
block address 1009379
search all blocks in cache
cache
1009377
1009376
1009379
1009378
hit - extract word from cache block

Figure 2: **Operation of a fully associative cache**. This example shows a small, four-block cache: there are two words per block. Each block is identified by its block address, which is stored in the cache along with the data itself. All blocks in the cache need to be searched for each memory reference: for large caches, this will take some time.

architecture, meaning that the data bus is 32 bits wide, as are the principal registers inside the CPU. In this context, a 32-bit quantity is referred to as a *word*, and it is words that are most often moved around the computer at any one time, not bytes. In C++, `float`s and `int`s are both 32 bits wide, and therefore occupy one word of memory: since the data bus is one word wide, a `float` or an `int` can be transferred from memory to the CPU in a single bus transaction. Nevertheless, memory systems are typically *byte addressable*: each memory address refers to an 8-bit quantity. The addresses of successive words differ by four so, for example, the CPU must increment the address by four if it wants to access the next element in an array of `float`s.

Even though memories are byte addressable, it is sometimes helpful to think of a *word address*, an address which refers to a memory word, not a memory byte. We can calculate the word address by dividing the byte address by four (the number of bytes per word) and rounding down to the nearest integer:

$$\text{word address} = \left\lfloor \frac{\text{byte address}}{\text{bytes per word}} \right\rfloor$$

Let's go one step further. We argued above that it makes sense to transfer more than one "item" at a time from the main memory into the cache. So let's say we transfer a *block* of data at a time, where a block comprises several words. This will require several bus transactions, but will not take that much longer than one transfer, since much of the transfer time is accounted for by a fixed overhead. We can now consider the memory as made up of sequential memory blocks, where a block is the smallest unit transferred between the main memory and the cache. We can give each block a number, a *block address*, calculated as follows:

$$\text{block address} = \left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor$$

When storing a block of data in the cache, we also need to store its block address for identification purposes: otherwise we have no way of knowing which blocks we have in the cache! These various ways of looking at a memory system are illustrated in Figure 1.

Cache organisation

It's time now to consider the detailed operation of our first cache: see Figure 2. As with all the examples in this section, we'll assume that each block comprises two words. Let's imagine the CPU needs to load the word starting at byte address `8049bcc`. This word is contained in block

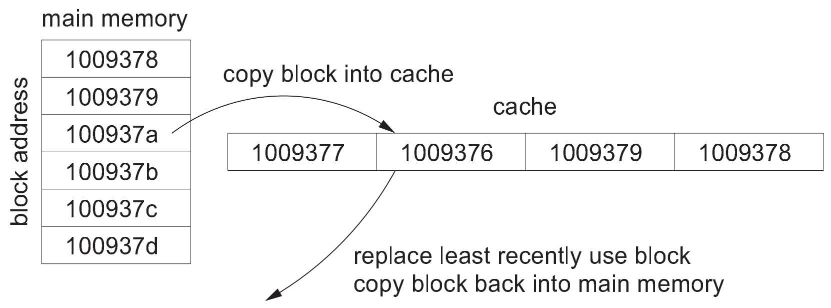

Figure 3: **Handling a cache miss with a fully associative cache**. The least recently used block is copied back to memory, then the block which caused the cache miss is loaded in its place.

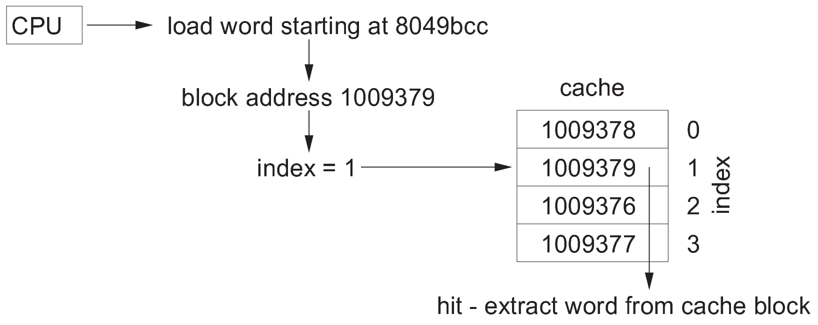

Figure 4: **Operation of a direct mapped cache**. Only one block of the cache needs to be checked. The index is calculated directly from the requested block address.

number `1009379`:

$$\text{block address} = \left\lfloor \frac{\text{byte address}}{\text{bytes per block}} \right\rfloor = \left\lfloor \frac{\texttt{8049bcc}}{8} \right\rfloor = \texttt{1009379}$$

Before resorting to a slow, main memory access, the CPU first checks to see if this word is in the cache. In this example, the cache is very small, only four blocks wide, and at this time contains blocks `1009376` to `1009379`. The CPU checks the block addresses of all the blocks in the cache and finds that the block it's looking for is indeed present: this is called a *cache hit*. It can then rapidly extract the word it wanted from the appropriate cache block, and move on to executing the next instruction.

If, however, the CPU had been trying to load a word from block `100937a`, we would have got a *cache miss*: see Figure 3. The CPU has no choice now but to stall while the required block is transferred from main memory to the cache. But which block should be thrown out of the cache to make way for the new data? Temporal locality of reference suggests that we should replace the *least recently used* (LRU) block, since the others are more likely to be needed again soon. An alternative to LRU is random block replacement, which is less well founded theoretically but faster to implement in hardware. Once the block has been transferred from the cache, the CPU can resume execution of the program.

The cache we've been considering is *fully associative*. A block can reside anywhere in such a cache, and the entire cache must be searched on every memory access. For large caches this is prohibitively slow (a main memory access would be faster!), so we need to come up with a better strategy. One candidate is the *direct mapped* cache, illustrated in Figure 4. In this scheme, each cache slot is labelled with an index in the range `0` to `n-1`, where there are `n` blocks in the cache (`n` is four in this little example). Each memory block is only allowed to occupy a particular cache

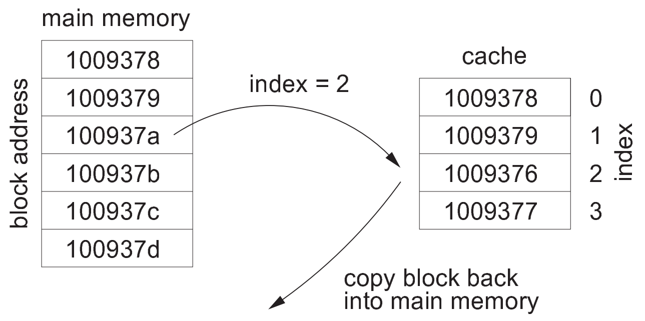

Figure 5: **Handling a cache miss with a direct mapped cache**. Since block `100937a` is only allowed to reside at index 2, there is no choice about which block to replace in the cache.

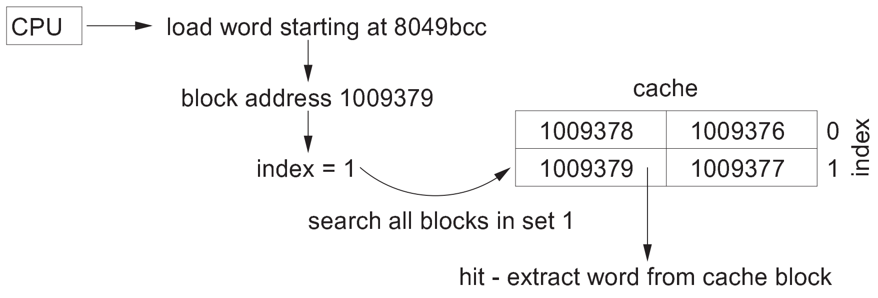
CPU
load word starting at 8049bcc
block address 1009379
index = 1
search all blocks in set 1
cache
1009378
1009376
1009379
1009377
0
1
index
hit - extract word from cache block

Figure 6: **Operation of a set associative cache**. The index points to a particular set, which contains a number of blocks. All blocks in the set need to be searched for the requested block.

slot, with an index given by

$$\text{index} = (\text{block address})\,\text{modulo}\ \mathtt{n}$$

Remember that "modulo" refers to the remainder after division: for example, `8` modulo 4 is 0 and `5b` modulo 4 is 3. So, with a direct mapped cache there is no need for a lengthy search of the cache: we simply calculate the index from the block address and check only one slot in the cache: if the block we want isn't there, it isn't in the cache at all.

In the event of a cache miss, we have no choice which block to replace in the cache. The block indicated by the index is copied back to main memory, and then the requested block is loaded from main memory into the cache. Figure 5 illustrates a cache miss when the CPU requires a word within block `100937a`.

Let's consider the relative merits of direct mapped and fully associative caches. A direct mapped cache has a smaller *hit time*, since no searching of the cache is required. On the other hand, we might expect the fully associative cache to have a lower *miss rate*, since a sensible block replacement strategy like LRU can be used to decide which block to throw out of the cache. This suggests a compromise known as a *set associative* cache, illustrated in Figure 6. In this scheme, the cache contains `n` *sets* (two in this example), each of which contains a number of blocks (again, two in this example). Each set is labelled with an index in the range `0` to `n-1`. A memory block is allowed to reside only in a particular set, with an index given by

$$\text{index} = (\text{block address})\,\text{modulo}\ \mathtt{n}$$

All the blocks in the indexed set need to be checked for a hit. The hit time is therefore somewhat slower than with a direct mapped cache, but faster than the fully associative cache, where all the blocks in the cache need to be checked. In the event of a cache miss, we have some choice as to

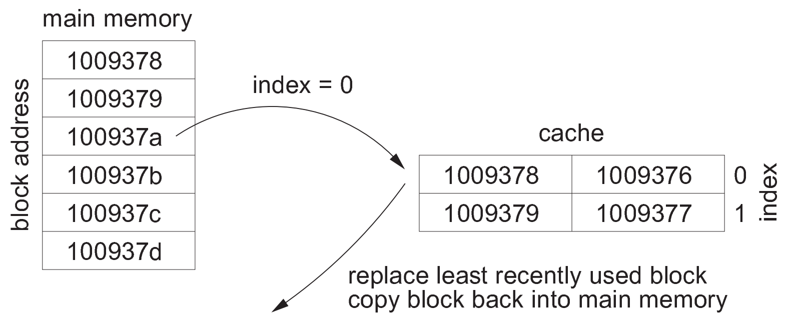

Figure 7: **Handling a cache miss with a set associative cache**. Any of the blocks in the indexed set can be replaced. In this example, an LRU replacement strategy is employed.

which block to replace: we can replace any of the blocks in the indexed set. Figure 7 illustrates how a cache miss would be handled with an LRU replacement strategy.

In the jargon, we would say that the cache in Figures 6 and 7 is *two-way set associative*, since there are two blocks per set. Set associative caches span the spectrum between direct mapped and fully associative caches: a one-way set associative cache is direct mapped, while an `n`-way set associative cache, where there are `n` blocks in the cache, is fully associative.

Cache design criteria

The total time taken by a program's memory references is given by

$$\text{time} = (\text{hits} + \text{misses}) \times \text{hit time} + \text{misses} \times \text{miss penalty}\ , \quad \text{miss rate} = \frac{\text{misses}}{\text{hits} + \text{misses}}$$

The aim of the cache design is therefore to minimise the miss rate, miss penalty and the hit time. As with most engineering design problems, there are trade-offs to consider. For example, increasing the block size normally reduces the miss rate (spatial locality of reference) but increases the miss penalty (more data needs to be copied from main memory to the cache).