THIS IS A TEST

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

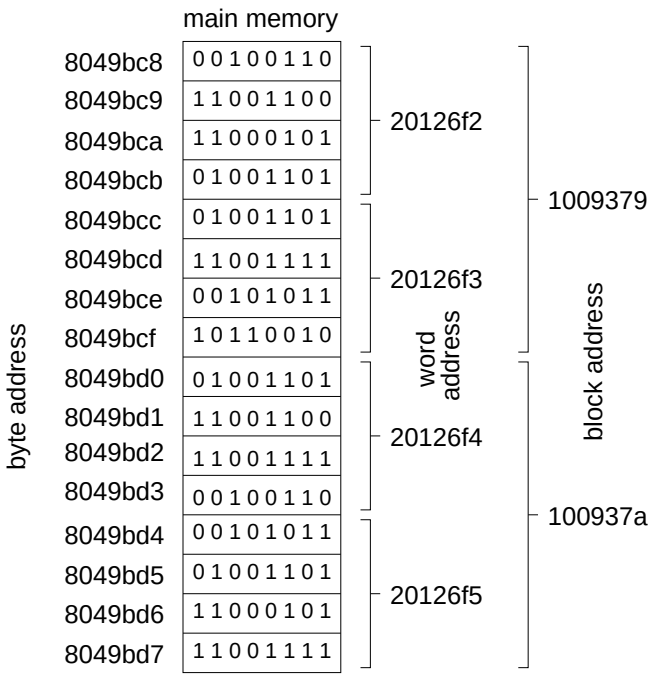


Figure 1: **Memory addresses**. Computer memories are byte addressable: each address corresponds to a group of 8 bits in memory. In this example, we see a segment of memory 16 bytes long, starting at hexadecimal address `8049bc8` and ending at `8049bd7`. Since most current CPUs manipulate 32 bits of 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

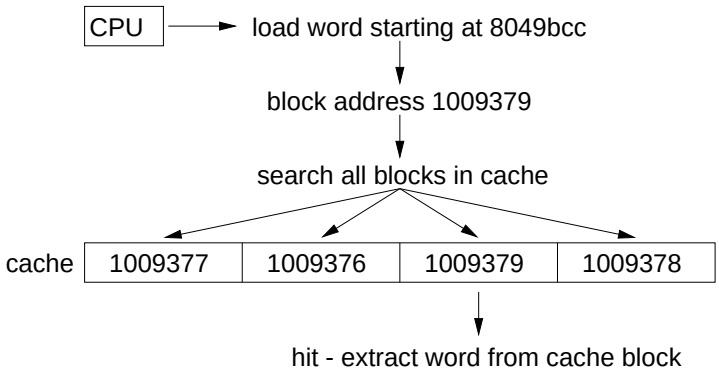


Figure 2: **Operation of a fully associative cache**. This example shows a small, four-block cache: there are two words per block. Each block is identified by its block address, which is stored in the cache along with the data itself. All blocks in the cache need to be searched for each memory reference: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

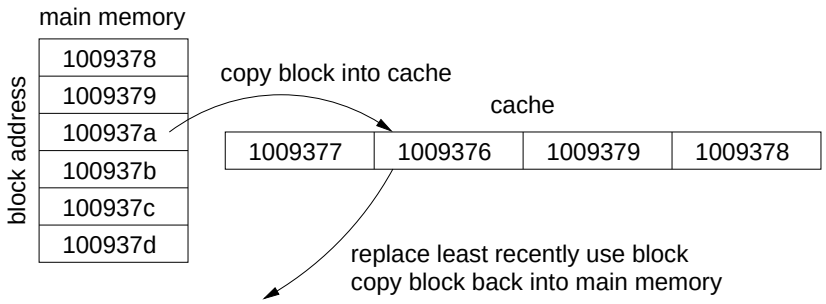


Figure 3: **Handling a cache miss with a fully associative cache**. The least recently used block is copied back to memory, then the block which caused the cache miss is loaded in its place.

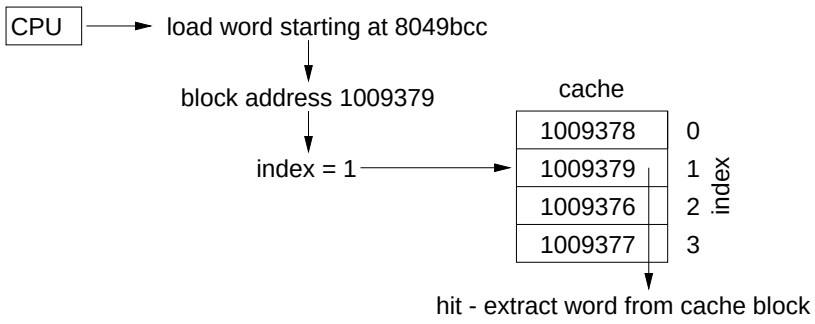


Figure 4: **Operation of a direct mapped cache**. Only one block of the cache needs to be checked. The index is calculated directly from the requested block address.

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

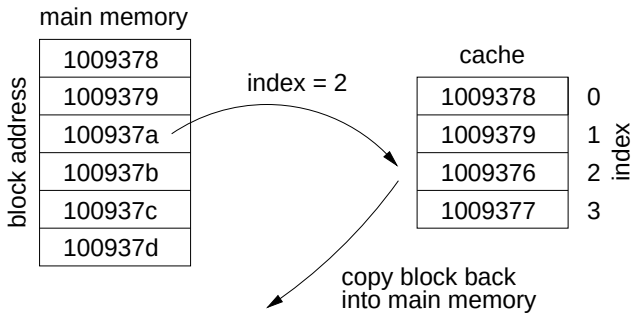


Figure 5: **Handling a cache miss with a direct mapped cache**. Since block `100937a` is only allowed to reside at index 2, there is no choice about which block to replace in the cache.

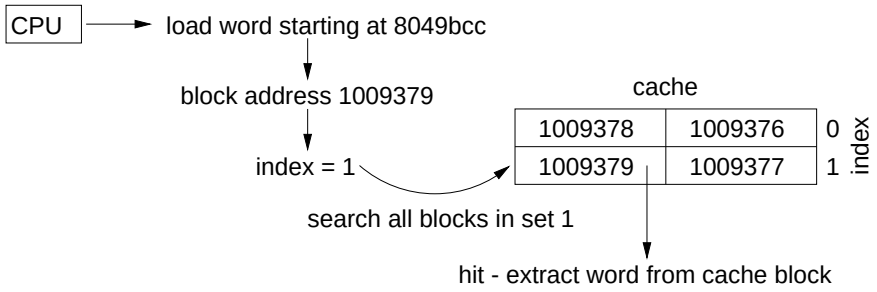


Figure 6: **Operation of a set associative cache**. The index points to a particular set, which contains a number of blocks. All blocks in the set need to be searched for the requested block.

slot, with an index given by

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

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

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

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

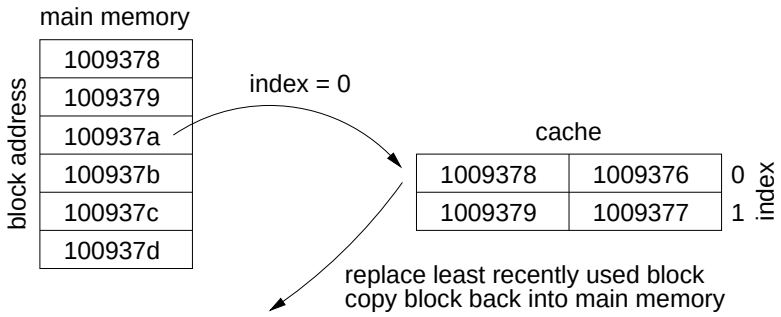


Figure 7: **Handling a cache miss with a set associative cache**. Any of the blocks in the indexed set can be replaced. In this example, an LRU replacement strategy is employed.

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).

3 Method

Experimental technique and apparatus

You are provided with a computer running a variant of the Unix operating system and a software development environment (`emacs` text editor and C++ compiler). Since it is impossible to reconfigure the hardware cache inside the computer, you will use a cache simulator to investigate the relative strengths and weaknesses of the various caching strategies. You will, however, use real sequences of memory references generated by pre-written programs and programs you will write yourself. You will also compare the performance of different programs which perform the same task but interact in different ways with the cache: for these experiments, you can use the computer's real, hardware cache, and not the simulator.

To start the experiment, log on to the computer and type

start-3F5caches

This creates a directory, called `3F5caches`, containing the cache simulator and all the pre-written programs, and opens a terminal window into which you will type commands.

copy ~i366/cache to your directory

Compiling from the command line, timing execution [10 minutes]

The first program we're going to consider is `histogram.cc`, which is provided in the `3F5Caches` directory and also listed in Appendix A. This program generates an array of random numbers and then counts how many of the numbers lie in particular ranges: in other words, it puts each number into an appropriate *bin*. A histogram of the numbers could be generated by subsequently plotting the number of items in each bin. This program is good for testing caches, since accesses to the random number list `l` exhibit good spatial locality of reference, while accesses to the bin counters `b` exhibit both spatial and temporal locality of reference.

Read through the source code and check that you understand what the program is doing (you can examine the program in the `emacs` text editor by typing `emacs histogram.cc &`). The code contains some unusual features, like the `INST_R` and `INST_W` directives and the `spacer` array, which you should ignore for now[1]. Compile the program from the command line by typing

```
g++ -o histogram histogram.cc
```

Run the program from the command line by typing

```
histogram
```

Check that there are roughly the same number of items in each bin, as you'd expect from a uniform random number generator. Now time the execution of the program by typing

```
time histogram
```

The `time` program outputs several times: the one we're interested in is the one labelled `user`, since this is the time the CPU spent executing the `histogram` user process. The manual page of the C++ compiler (type `man g++`) indicates that the compiler can be asked to optimise the machine code it generates. Recompile the histogram program with optimisation by typing

```
g++ -O -o histogram histogram.cc
```

Now re-run `histogram` and record the new execution time. Finally, prepare the program for the next set of experiments by commenting out the lines which display the final bin contents and making the program work with a list of 100 (instead of 10,000,000) numbers.

Deducing memory references, code instrumentation [10 minutes]

Our next aim is to deduce the sequence of memory references generated by `histogram` as it runs. At this point, we should be careful to distinguish between two types of memory reference. First, there are instruction fetches: the CPU needs to fetch each machine code instruction from memory before it can execute it. Instruction fetches are normally sequential, unless there is a `branch` or a `jump` instruction which changes the flow of the program. Secondly, there are data reads and writes, caused by the instructions themselves accessing data in memory. The good news is that we can consider these two types of memory references independently, since most computers have two caches, one for instructions and one for data. In this experiment, we'll consider only the data references and the corresponding data cache.

So, how can we deduce the sequence of data memory references generated by the `histogram` program? Ideally, we'd go through the machine code produced by the compiler and identify those instructions which access memory. However, we'd much rather stick with the source code, so we'll have to make some sensible guesses about what the compiler's doing. Let's look just at the loop which runs through the list and bins each item. Each time round the loop, the program needs to access the integer `i`, the integer `listSize`, the integer `binWidth`, the integer `j`, one element of `l` and one element of `b`, the last reference happening twice, a read and then a write. Let's assume

[1] It might also seem strange that the arrays are declared outside of `main()`. If we didn't do this, the memory for the arrays would be allocated on the stack. This would cause problems if the arrays were very large, since we would then run out of stack space and get a run-time segmentation fault.

the compiler keeps all the regularly referenced variables (`i`, `listSize`, `binWidth` and `j`) in CPU registers, and accesses memory only when it needs to read `l[i]` and read/write `b[j]`. That makes three memory references each time round the loop, two reads and a write.

We can obtain the addresses of these three references by *instrumenting* the code appropriately. That's what the `INST_R` and `INST_W` directives do: `INST_R` stands for "instrumentation read" while `INST_W` stands for "instrumentation write". The directives are defined in the header file `inst_none.h`: if you look at this file, you'll see that they are currently defined to do nothing, which is why you haven't noticed their effect yet. However, if you look in the file `inst_legible.h` you'll see an alternative definition which displays on the screen the address of each memory reference. For example, `INST_R(l[i])` expands to

```
cout << "Read from " << hex << &(l[i]) << endl
```

which prints the hexadecimal address of `l[i]` on the standard output (normally the screen). Let's now put this instrumentation to the test. Change the line `#include "inst_none.h"` to `#include "inst_legible.h"`. Recompile the program and run it. You should see a long list of hexadecimal addresses (`0x` is the prefix for hexadecimal) go scrolling by on the screen. To examine the list more easily, you could redirect the standard output to a file instead of the screen, by typing

```
histogram > addresses.log
```

You can now examine the file `addresses.log` in `emacs`, or scroll through it page by page by typing

```
more addresses.log
```

Even neater, you could pipe the output of histogram straight into the `more` program by typing

```
histogram | more
```

A pipe (the `|` symbol) allow the output of one program (in this case `histogram`) to be sent directly to the input of another (in this case `more`). Now that you can examine the memory references with ease, check they follow the expected pattern: there should be sequential reads from `l` (remember that the addresses of consecutive elements of `l` differ by four, not one) followed by a read and write of an element of `b`. The `spacer` array in `histogram.cc` ensures that `l` and `b` have very different addresses, so you shouldn't have any difficulty distinguishing between the two.

Initial experiments with the cache simulator [40 minutes]

Without a cache, all those memory references would cause the CPU to stall while it waits for the memory system to respond. We are now in a position to see how a cache improves matters. The provided `cache` program is a cache simulator which reads a sequence of memory addresses (in binary, not legible ASCII) and reports the subsequent numbers of hits and misses. The first thing we need to do is change the instrumentation to output the addresses in a compact, binary form instead of the legible ASCII form. To do this, simply change `#include "inst_legible.h"` to `#include "inst_compact.h"` and recompile `histogram`. We can now send the memory references to the cache simulator by typing

```
histogram | cache
```

The simulator assumes a default 128 byte direct mapped cache with one-word blocks. You can reconfigure the cache by supplying command line arguments. For example

```
histogram | cache -b4 -a2 -s1k
```

simulates a 1 KByte, two-way set associative cache with four-word blocks. Typing `cache -h` displays the help page for the cache simulator, including the full set of command line arguments: the help page can also be found in Appendix B. Of particular interest is the `-g` flag, which launches a graphical user interface which should help you understand what's going on inside the cache. Read the help page and experiment with the `cache` program until you are familiar with the operation of the graphical interface.

Run the `histogram` sequence through a number of different 128 byte caches as follows:

```
histogram | cache -g                histogram | cache -b2 -a2 -g
histogram | cache -b2 -g             histogram | cache -b2 -a2 -r -g
```

In each case, record the cache access statistics (including the total time, the hit time and the miss penalty) and use the graphical interface to understand how the different configurations affect the miss rate. Pay particular attention to how the elements of `b` are cached in each case. Without changing the size of the cache, experiment with different block sizes, associativities and block replacement strategies until you find the minimum miss rate.

Caches and algorithmic complexity [30 minutes]

The next program we're going to look at is `sort.cc`, which is provided in the 3F5Caches directory and also listed in Appendix A. The program uses either exchange sort or QuickSort to sort a list of 10,000 random numbers. The simple version of QuickSort assumes that there are no repeated numbers in the list. The list is therefore initialised with the numbers 0 to 9999, and is then scrambled by repeatedly swapping items. Read through the source code and check that you understand what the program is doing.

Now instrument the sorting functions to identify all references to `list`. Starting with exchange sort, run the code[2] through the cache simulator (without the graphical interface) and experiment with different direct mapped caches: try cache sizes in the range 4 KBytes to 64 KBytes and block sizes in the range one to eight words. In each case, record the cache access statistics and attempt to understand the observed miss rates. Note that exchange sorting a list of 10,000 items takes some time: you'll probably want to interrupt each simulation (by pressing `ctrl-c`) after, say, 100,000 accesses, and record the statistics up to that point. Next, edit `sort.cc` to enable QuickSort in place of exchange sort, recompile and repeat the above experiments.

Writing cache-friendly software [30 minutes]

The final algorithms we'll consider perform matrix multiplication, a task that crops up frequently in numerical computation. The file `blocking.cc` (instrumented source code is in the `3F5Caches` directory and also in Appendix A) uses a clever, cache-friendly algorithm to calculate `b` times `c` and store the result in `a`. Compile and run this program: it displays the 10×10 product on the screen.

Now take a copy of `blocking.cc` and call it `obvious.cc` (to make the copy, type the command `cp blocking.cc obvious.cc`). Replace the central part of `obvious.cc` with the more obvious code for matrix multiplication. Use a single `for` loop to calculate each element of `a` as follows:

$$a_{ij} = \sum_{k=0}^{\text{matrixSize}-1} b_{ik} c_{kj}$$

Then embed this `for` loop inside two outer loops which generate all combinations of `i` and `j`. Compile and run `obvious.cc` and check it produces the same result as `blocking.cc`.

Now let's think about how cache-friendly these two alternative programs are. First let's work on a larger problem by changing `matrixSize` to 100, `blockingFactor` to 20, and commenting out the lines which display the product at the end of both programs. Add instrumentation to `obvious.cc`, along the same lines as the instrumentation in `blocking.cc`. Include the header file for compact instrumentation and recompile both programs. Run them both through the cache simulator: record the miss rates for direct mapped caches of size 4, 8, 16 and 32 KBytes, with block sizes of 1, 2, 4 and 8 words (32 measurements in total).

Finally, let's see how the computer's real, hardware cache copes with a very large matrix multiplication problem. Change `matrixSize` to 1000 and `blockingFactor` to 50. Remove the instrumentation by including the header file `inst_none.h` instead of `inst_compact.h`. Compile `obvious.cc` and `blocking.cc` with and without optimisation, and record the execution time of each run (four measurements in total).

[2] Execute the program by typing `./sort` and not `sort`, otherwise you'll probably run the standard Unix sort utility by accident!

4 Writing up

You should refer to the advice given in Sections 2.2 and 2.3 of the *General Instructions* document.

The Laboratory Report [2 hours]

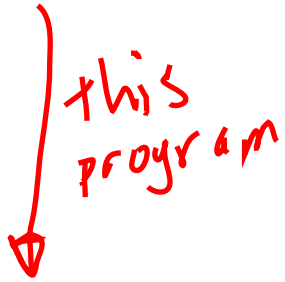
This should be no more than five sides in longhand or three if word processed, excluding any diagrams or program listings. The report should contain all the numerical results requested in Section 3 and also the following:

- A discussion of the results of the `histogram` experiments, explaining the observed miss rates.
- An explanation of the different hit times and miss penalties of the caches studied in the `histogram` experiments.
- A listing of the instrumented `sort.cc` source code.
- A discussion of the `sort` experiments, including an explanation of the observed miss rates and an assessment of the relative importance of cache-friendliness and algorithmic complexity.
- A listing of the instrumented `obvious.cc` source code.
- A comparison of `blocking.cc` and `obvious.cc` in terms of their algorithmic complexity and cache-friendliness. Why did increasing the block size sometimes *increase* the miss rate?
- A discussion of the significance of the results, given that all of the simulated caches were considerably smaller than current hardware caches.

The Full Technical Report [10 additional hours]

Guidance on the preparation of Full Technical Reports is provided both in Appendix I of the *General Instructions* document and in the CUED booklet *A Guide to Report Writing*, with which you were issued in the first year. If you are offering a Full Technical Report on this experiment, you should read Chapter 7 of [1] and discuss the following points in addition to those listed above. You should include your Laboratory Report as an appendix and refer to it where appropriate.

- What would be the optimal `blockingFactor` to use for matrix multiplication?
- Does increased associativity always reduce the miss rate?
- How are caches implemented in hardware? Why does the simulator insist that the block size, and the number of sets, must be a whole power of two? Is it always necessary to store the entire block address in the cache for identification purposes?
- Why does a main memory access involve a significant, fixed overhead? How can main memory systems be designed to support caches?
- What are the attractions of a *multilevel* caching strategy?
- Describe how a program like the one overleaf could be used to deduce information about the configuration of a hardware cache.




```
const int listSize = 1000000; // try different list sizes
int l[listSize];

int main()
{
  int i, j, k, stride, repetitions;

  stride = 1; // try different strides
  repetitions = 10; // access the same data many times

  for (i=0; i < repetitions; i++)
    for (j=0; j < listSize; j+= stride) k = l[j];

  return 0;
}
```

References

[1] D. A. Patterson and J. L. Hennessey. *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, 2nd edition, 1998.

A Program listings

histogram.cc

```
const int maxNumber = 10000;
const int listSize = 10000000;
const int numBins = 10;

int l[listSize];
int spacer[10000000]; // separates the addresses of b and l
int b[numBins];

int main()
{
  int i, j, binWidth;

  // Generate a list of random integers between 0 and maxNumber-1
  for (i=0; i < listSize; i++) l[i] = rand() % maxNumber;

  // Reset the bin counts to zero
  for (i=0; i < numBins; i++) b[i] = 0;

  // Work out the width of each bin - must be an integer
  if (maxNumber % numBins) return 1; // return error code
  else binWidth = maxNumber / numBins;

  // Run through the list and bin each item
  for (i=0; i<listSize; i++) {
    INST_R(l[i]);
    j = l[i]/binWidth;
    INST_R(b[j]);
    b[j]++;
    INST_W(b[j]);
  }

  // Display the final contents of each bin
  for (i=0; i < numBins; i++)
    cerr << "Bin " << i << " contains " << b[i] << " numbers" << endl;

  return 0; // return success code
}
```

sort.cc

```
const int listSize = 10000;
int l[listSize];

void XSort(int list[], int n);
void QSort(int list[], int lo, int hi);
int Partition(int list[], int lo, int hi);

int main()
{
  int i, tmp, i1, i2;

  // Generate a list with no repeated numbers
  for (i=0; i<listSize; i++) l[i] = i;

  // Scramble the numbers thoroughly
  for (i=0; i<listSize; i++) {
    i1 = rand() % listSize; i2 = rand() % listSize;
    tmp = l[i1]; l[i1] = l[i2]; l[i2] = tmp;
  }

  // Sort the list by exchange sort or QuickSort - comment out the
  // one you don't want to use
  XSort(l, listSize);
  // QSort(l, 0, listSize-1);

  return 0; // return success code
}

void XSort(int list[], int n)
  // Exchange sort
{
  int min, tmp, i, j, min_j;

  // Scan the list from the left to the right
  for (i=0; i<n-1; i++) {

    // Remember the item at position i
    min = list[i]; min_j = i;

    // Check the list to the right of position i for any smaller items
    for (j=i+1; j<n; j++) {
      if (list[j] < min) {
        // Remember where this smaller item is
        min = list[j]; min_j = j;
      }
    }
    // Swap the item at position i with the smallest item found to the right
    tmp = list[i]; list[i] = list[min_j];
    list[min_j] = tmp;
  }
}
```

sort.cc (continued)

```
void QSort(int list[], int lo, int hi)
  // QuickSort - this simple version assumes no repeated items in the list
{
  int k;

  if (lo < hi) {

    // Partition the list into two sub-lists
    k = Partition(list, lo, hi);

    // Now every item left of position k is smaller than the item at k,
    // while every item right of position k is larger than the item at k
    QSort(list, lo, k-1); // sort the sublist to the left of k
    QSort(list, k+1, hi); // sort the sublist to the right of k
  }
}

int Partition(int list[], int lo, int hi)
  // Partition function for QuickSort
{
  int x, tmp;

  // Pick an arbitrary key, say half way through the list
  x = list[(lo+hi)/2];

  // Now swap items until every item to the left of the key is smaller than
  // the key, and every item to the right of the key is larger than the key
  while (lo < hi) {

    // Scan from the right until we find an item smaller than the key
    while ( (lo < hi) && (x < list[hi]) ) hi--;

    // Scan from the left until we find an item larger than the key
    while ( (lo < hi) && (x > list[lo]) ) lo++;

    // Swap the two items we've discovered on the wrong side of the key
    tmp = list[hi]; list[hi] = list[lo]; list[lo] = tmp;
  }

  return lo; // this is where the key is now
}
```

blocking.cc

```
const int matrixSize = 10;
const int blockingFactor = 5;
int a[matrixSize][matrixSize];
int b[matrixSize][matrixSize];
int c[matrixSize][matrixSize];

int main() {

  int i, j, k, jj, kk, r;

  // Initialise the matrices arbitrarily
  for (i=0; i < matrixSize; i++)
    for (j=0; j < matrixSize; j++) {
      b[i][j] = i + j; c[i][j] = i - j; a[i][j] = 0;
    }

  // Work out a = b * c, using a blocking algorithm
  jj = 0;
  kk = 0;
  while (jj < matrixSize) {
    while (kk < matrixSize) {
      for (i=0; i < matrixSize; i++)
        for (j=jj; j < jj + blockingFactor; j++) {
          r = 0;
          for (k=kk; k < kk + blockingFactor; k++) {
            INST_R(b[i][k]);
            INST_R(c[k][j]);
            r += b[i][k] * c[k][j];
          }
          INST_R(a[i][j]);
          a[i][j] = a[i][j] + r;
          INST_W(a[i][j]);
        }
      kk += blockingFactor;
    }
   kk = 0;
   jj += blockingFactor;
  }

  // Display the product
  for (i=0; i < matrixSize; i++) {
    for (j=0; j < matrixSize; j++) cerr << a[i][j] << ' ';
    cerr << endl;
  }

}
```

B The cache simulator help page

VisiCache OpenGL Cache Simulator — copyright Andrew Gee, November 2002

usage: cache s:b:a:rgh

where command line arguments may be:

-s size[k]	size of the cache in [k]bytes (default 128 bytes)
-b number	number of words per block (default 1, must be a power of 2)
-a degree	degree of associativity (default 1)
-r	use random block replacement (default is to use LRU)
-g	enable graphical display (default is no graphical display)
-h	this help message

The specified cache size (-s argument) must result in a whole number of sets which is a whole power of two: otherwise, the program will not run.

The cache simulator reads a sequence of addresses (in binary, not ASCII) from the standard input. These addresses may conveniently be piped from another running process, or redirected from a file.

When the graphical display is enabled, the program responds to several key strokes and mouse actions:

click and drag mouse	pan display left/right and up/down
up/down cursor keys	zoom display in/out
space	perform a single cache access, halt continuous access
enter	perform continuous cache accesses
'd' or 'D'	cycle the address display through the three options
'3'	toggle 3D mode on/off (if supported by display)
'q', 'Q' or escape	quit the graphical display

In the graphical display, occupied cache blocks are shown shaded blue. The last block to be accessed is shaded green (for a hit) or red (for a miss). Addresses are displayed only when zoomed in sufficiently to be able to read them. The default is to display the byte address of each word: use the 'd' key to display instead the block address or the set index.

In 3D mode, hit and miss counts are displayed as vertical bars on either side of each cache block. Click and drag with the middle and right mouse buttons to rotate the cache in 3D. This mode is available only on displays which support depth buffers.