
Consider two alternate caches, each with 4 sectors holding 1 block per sector and one 32-bit word per block. One cache is direct mapped and the other is fully associative with LRU replacement policy. The machine is byte addressed on word boundaries and uses write allocation with write back.

1a) (7 points) What would the overall miss ratio be for the following address stream on the direct mapped cache? Assume the cache starts out completely invalidated.

```
read 0x00    M
read 0x04    M
write 0x08   M
read 0x10    M
read 0x08    H
write 0x00    M  Miss ratio = 5/6 = 0.8333
```

1b) (6 points) Give an example address stream consisting of only reads that would result in a lower miss ratio if fed to the direct mapped cache than if it were fed to the fully associative cache.

```
read 0x00
read 0x04
read 0x08
read 0x0C
read 0x14 ; victim 0x00 on associative but 0x04 on direct
read 0x00 ; miss on associative, but hit on direct
```
2. Virtual Memory and Cache Organization.

The 742LX is a uniprocessor having up to a maximum of 64 MB of addressable physical memory. The cache, virtual memory, and TLB have the following attributes:

<table>
<thead>
<tr>
<th>Cache</th>
<th>Virtual Memory</th>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>unified</td>
<td>virtual page size is 4 KB</td>
<td>unified</td>
</tr>
<tr>
<td>physically addressed</td>
<td>virtual address space is 1 GB</td>
<td>fully associative</td>
</tr>
<tr>
<td>cache holds 20 KB</td>
<td></td>
<td>40 entries</td>
</tr>
<tr>
<td>5 way set associative</td>
<td></td>
<td>1 byte control/entry</td>
</tr>
<tr>
<td>32 Byte block size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sector size = block size</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LRU replacement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>write back</td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte addresses on word boundaries</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2a) (10 points) Sketch a block diagram of how the virtual address is mapped into the physical address (assuming a TLB hit). Be sure to label exactly which/how many of the address bits go where, and how many bits are in each of the 3 fields in a TLB entry.

1 GB MEANS 30-BIT VIRTUAL ADDRESS

0 KB PAGE SIZE MEANS LOW 12 BITS ARE UNTRANSLATED

TLB

ASSOCIATIVE LOOKUP

18 bit PAGE NAME 8 bits CONTROL 14 bit PAGE ADDRESS

14-BIT PAGE ADDRESS (BIT 12-26 OF PHYSICAL ADDRESS)
2b) (14 points) Given that you have the address output of a TLB and the original virtual address, sketch a block diagram of how the cache is accessed to determine whether there is a cache hit (you may ignore data access -- just indicate enough to say whether a hit or miss occurs; also include only tag fields in your picture of the cache organization). Again label exactly which/how many address bits go where and how big an address tag is.
3. Multi-Level Caches.

You have a computer with two levels of cache memory and the following specifications:

CPU Clock: 200 MHz  Bus speed: 50 MHz
Processor: 32-bit RISC scalar CPU, single data address maximum per instruction
L1 cache on-chip, 1 CPU cycle access
  block size = 32 bytes, 1 block/sector, split I & D cache
  each single-ported with one block available for access, non-blocking
L2 cache off-chip, 3 CPU cycles transport time (L1 miss penalty)
  block size = 32 bytes, 1 block/sector, unified single-ported cache, blocking, non-pipelined
Main memory has 12+4+4+4 CPU cycles transport time for 32 bytes (L2 miss penalty)

Below are the results of a dinero simulation for the L1 cache:

```
CMDLINE: dinero -b32 -i8K -d8K -a1 -ww -An -W8 -B8
CACHE (bytes): blocksize=32, sub-blocksize=0, wordsize=8, Usize=0,
              Dsize=8192, Isize=8192, bus-width=8.
POLICIES: assoc=1-way, replacement=l, fetch=d(1,0), write=w, allocate=n.
CTRL: debug=0, output=0, skipcount=0, maxcount=10000000, Q=0.
```

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Access Type:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(totals,fraction)</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>Demand Fetches</td>
<td>100000000</td>
</tr>
<tr>
<td></td>
<td>1.0000</td>
</tr>
<tr>
<td>Demand Misses</td>
<td>52206</td>
</tr>
<tr>
<td></td>
<td>0.0052</td>
</tr>
</tbody>
</table>

Words From Memory      | 180920        |
( / Demand Fetches)    | 0.0181        |
Words Copied-Back      | 766845        |
( / Demand Writes)     | 1.0000        |
Total Traffic (words)  | 947765        |
( / Demand Fetches)    | 0.0948        |

3a) (6 points) What is the available (as opposed to used) sustained bandwidth between:

- L1 cache bandwidth available to CPU (assuming 0% L1 misses)?

  \[200 \text{ MHz} \times 2 \text{ caches} \times 32 \text{ bytes} / 1 \text{ clock} = 12.8 \times 10^9 \text{ B/sec} = 11.92 \text{ GB/sec}\]

- L2 cache bandwidth available to L1 cache (assuming 0% L2 misses)?

  \[200 \text{ MHz} \times 1 \text{ cache} \times 32 \text{ bytes} / 3 \text{ clocks} = 2.133 \times 10^9 \text{ B/sec} = 1.98 \text{ GB/sec}\]

- Main memory bandwidth available to L2 cache?

  \[200 \text{ MHz} \times 32 \text{ bytes} / (12+4+4+4) \text{ clocks} = 267 \times 10^6 \text{ B/sec} = 254 \text{ MB/sec}\]
3b) (9 points) How long does an *average* instruction take to execute (in ns), assuming 1 clock cycle per instruction in the absence of memory hierarchy stalls, no write buffering at the L1 cache level, and 0% L2 miss rate.

7362210 instructions = 7362210 clock cycles @ 1 clock effective access time

52206 demand misses @ 3 clocks = 156618 clocks delay penalty.

\[
\frac{(7362210 + 156618)}{7362210} = 1.0213 \text{ clocks} / 200 \text{ Mhz} = 5.1065 \text{ ns}
\]

3c) (7 points) A design study is performed to examine replacing the L2 cache with a victim cache. Compute a measure of speed for each alternative and indicate which is the faster solution. Assume the performance statistics are
L2 cache local miss ratio = 0.19
Victim cache miss ratio = 0.26; and its transport time from L1 miss = 1 clock

Given fixed L1 cache performance, it is fair to compare these head-to-head (but the comparison might not stay the same if L1 were changed):

\( t_{ea} \) for L2 cache beyond the L1 access time is:

\[
3 + 0.19 \times (12+4+4+4) = 7.56 \text{ clocks in addition to L1 delay}
\]

\( t_{ea} \) for L2 cache beyond the L1 access time is:

\[
1 + 0.26 \times (12+4+4+4) = 7.24 \text{ clocks in addition to L1 delay}
\]

So, in this (contrived) case the victim cache is a slight win in speed, and a whole lot cheaper.

You have instrumented only data references from the subroutine “sum_array” in the following program using Atom on an Alpha workstation (“long” values are 64 bits). The resultant data reads and writes have been run through dinero with a particular cache configuration. In this question you’ll deduce the cache configuration used.

```
#include <stdio.h>
#include <stdlib.h>

void sum_array(int N, long *a, long *b, long *c, long *d, long *e)
{ int i;
  for (i = 0; i < N; i++)
  { *(a++) = *(b++) + *(c++) + *(d++) + *(e++); }
}

int main(int argc, char *argv[])
{ int N, offset, i;
  long *a, *b, *c, *d, *e;    /* 64-bit elements in each array */
  if (argc != 3)
  { fprintf(stderr, "Usage: test <size> <offset>\n"); exit(-1); }
  sscanf(argv[1], "%d", &N);      sscanf(argv[2], "%d", &offset);
  a = (long *) malloc(5 * ( (N*sizeof(long)) + offset) );
  b = a + N + ( offset/sizeof(long) );
  c = b + N + ( offset/sizeof(long) );
  d = c + N + ( offset/sizeof(long) );
  e = d + N + ( offset/sizeof(long) );
  sum_array(N, a, b, c, d, e);
}
```

The program was executed with a command line having successively higher values of N from 1 to 2100, and an offset value of 0. The below graph shows the number of combined data misses for each value of N.

- Bus size and word size are both 8 bytes.
- The cache has one block per sector and a block size of 128 bytes (16 words).
- Assume a completely invalidated cache upon entry to sum_array.
Some data that might be of interest are:

<table>
<thead>
<tr>
<th>N</th>
<th># Misses</th>
<th>Total Traffic (words)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>626</td>
<td>2516</td>
</tr>
<tr>
<td>501</td>
<td>628</td>
<td>2533</td>
</tr>
<tr>
<td>502</td>
<td>629</td>
<td>2534</td>
</tr>
<tr>
<td>503</td>
<td>630</td>
<td>2535</td>
</tr>
<tr>
<td>504</td>
<td>630</td>
<td>2520</td>
</tr>
<tr>
<td>505</td>
<td>632</td>
<td>2537</td>
</tr>
<tr>
<td>506</td>
<td>633</td>
<td>2538</td>
</tr>
<tr>
<td>507</td>
<td>696</td>
<td>3531</td>
</tr>
<tr>
<td>508</td>
<td>1020</td>
<td>8700</td>
</tr>
<tr>
<td>509</td>
<td>1277</td>
<td>12797</td>
</tr>
<tr>
<td>510</td>
<td>1597</td>
<td>17902</td>
</tr>
<tr>
<td>511</td>
<td>1853</td>
<td>21983</td>
</tr>
<tr>
<td>512</td>
<td>2079</td>
<td>25584</td>
</tr>
<tr>
<td>513</td>
<td>1862</td>
<td>22097</td>
</tr>
<tr>
<td>514</td>
<td>1606</td>
<td>17986</td>
</tr>
<tr>
<td>515</td>
<td>1288</td>
<td>12883</td>
</tr>
<tr>
<td>516</td>
<td>1032</td>
<td>8772</td>
</tr>
<tr>
<td>517</td>
<td>717</td>
<td>3717</td>
</tr>
<tr>
<td>518</td>
<td>651</td>
<td>2646</td>
</tr>
<tr>
<td>519</td>
<td>653</td>
<td>2663</td>
</tr>
<tr>
<td>520</td>
<td>652</td>
<td>2632</td>
</tr>
<tr>
<td>521</td>
<td>655</td>
<td>2665</td>
</tr>
<tr>
<td>522</td>
<td>655</td>
<td>2650</td>
</tr>
<tr>
<td>523</td>
<td>657</td>
<td>2667</td>
</tr>
<tr>
<td>524</td>
<td>658</td>
<td>2668</td>
</tr>
</tbody>
</table>

Answer the following questions, giving brief support for your answer of (unsupported answers will not receive full credit).
4a) (5 points) Ignoring overhead for the subroutine call, what is the theoretical minimum possible data total traffic (in 8-byte words) that this program has to move (combined into and out of the cache) for N = 500?

Each array element touched once. $500 \times 5 = 2500$. Actual: 2516 so this makes sense

4b) (6 points) Does the cache perform write allocation?

With write-allocation for N = 500 you would expect to move extra words to fill the cache block when missing on a write to a[i]. In fact, 512 extra words, making it 3012 words of traffic; which is more than we observed. Actual is 2516 so it must be write-no-allocate.

4c) (6 points) Assuming it is not direct mapped, does this data look like it came from an LRU replacement policy or a random replacement policy? Why?

LRU - data is very “clean” and miss rate is nearly monotonic with increasing array sizes except for conflict areas; with random replacement one would expect to see accidental mini-spikes even with non-conflicting data at “random” places on the graph.

Notes:
- an important thing to note about the program is that only touches the data array once
- a random replacement policy still has significant spikes because all the action is concentrated in a single set when the arrays conflict -- there is going to be a loser no matter what the replacement policy is
4d) (6 points) Assume that you have a direct mapped cache. What is the best value for the input parameter offset if you want to improve performance for N=512 ("best" means the smallest value guaranteed to have 100% effectiveness for N=512).

offset=128 = 1 block size which offsets arrays by exactly 1 set in cache. Array is visited only once, so there is no issue of capacity misses -- only conflict misses matter.

4e) (10 points) What is the actual associativity of the cache that produced the data given? (and how did you figure that out?)

Traffic drop-off at offset of 6 words away on both sides of 512 means that at this distance you have only enough integral blocks of offset to fit into the associativity. So block size/associativity =~ 6 words; associativity =~ $128/(6\times8) = 2.66 \Rightarrow 3$ way set associative.

4f) (8 points) How many bytes does the cache hold (data only, not counting control+tag bits)?

Array size of 512 introduces conflicts. With 4 data sets and 3 way set associativity this means that each sector in cache holds 512*8 bytes = 4K. 4K*3 = 12 KBytes.