

[Show All Answers](#)
[Hide All Answers](#)

Cache experiment

Problem 1.

Introduction to BSim's cache simulation

In this problem, you'll be using BSim to measure the performance of different cache architectures running one of two test programs. A brief introduction to running BSim can be found in the documentation for Design Project #2.

To access the cache control panel, first load and assemble the test program (filenames are given below). When the window showing the Beta datapath appears, click on the "\$" button in the toolbar to bring up the control panel. You can select various architectural parameters for the cache on the left of the panel; cache statistics are presented on the right.

The settable cache parameters are:

- *Cache*: you can turn the cache ON or OFF.
- *Words/line*: this is the block size, i.e., how many words are read from memory when refilling a cache line and how many words are written when emptying a dirty cache line on a write-back cache.
- *Total lines*: total number of cache lines, so the total number of data words in the cache is $(\text{Words/line}) * (\text{Total lines})$. The total number of lines is divided equally among the different "ways" of the cache. For example, suppose the cache has 64 total lines. If the cache is direct mapped, it has 64 lines and 6 bits of address would be used to select the appropriate line. If the cache is 2-way associative, each subcache would have 32 lines and 5 bits of address would be used to select the appropriate line in each subcache.
- *Associativity*: direct mapped, 2-way, 4-way, 8-way or fully associative.
- *Replacement strategy*: on all but direct mapped caches, this determines which subcache is chosen to receive memory data on a cache miss. Choices are:
 - LRU: least recently used
 - FIFO: first-in, first-out (aka least recently replaced)
 - Random: choice is made at random
 - Cycle: the choice is made in a round-robin fashion
- *Write strategy*: write-through where all writes cause a memory access, and write-back where writes happen only into the cache and writes to memory happen only when a dirty cache line is replaced.

Cache statistics include:

- *Address*: information about how the address is decomposed into the tag and cache index. The cache index selects which line of the cache is checked; the tag field of the

address is matched against the tag entry for the cache line to determine if there was a hit. The data select field selects the appropriate word/byte of the cache data.

- *Cost*: breaks out the cost of each of the cache components. Register/SRAM is the storage required to hold cache data, tag information, valid bit and, if necessary, the dirty bit. Comparators are used to perform the tag comparisons, and 2-to-1 muxes are needed when the words/line > 1 to select the appropriate data word to return to the CPU. The costs correspond roughly to the relative sizes of the various components in a typical CMOS process.
- *Performance*: enumerates how the cache has performed on all the memory accesses since the simulation was last reset. The number of cycles represents the accumulated cycle count, where a cache access takes 1 cycle and misses take an additional 4 cycles to access the first word from main memory plus 1 additional cycle to fetch subsequent words when words/line > 1. When the cache is off, each memory access takes 4 cycles. Note that both instruction and data (eg, LD and ST) accesses are included in the statistics.

To run an experiment you can use the STOP, RESET, RUN, and SINGLE-STEP buttons at the top of the cache control panel. Each time you change the cache parameters the simulation is automatically RESET, so you only need to click RUN again to see what effect your changes had on cache performance.

The experiments

The first set of experiments involves the following test program, which can be found ["/mit/6.004/code/cache1.uasm"](#):

```
| Simple program to test cache configurations.
.
.include beta.uasm
. = 0          | Reset vector at 0
    BR(Start)

Start:  LD(firstadr,r1)
        LD(lastadr,r2)
        LD(increment,r3)

| 4-instruction loop, starting at location 16 = 4*4:
| Note that there are 4 words of instruction fetch
| per loop, and a single data memory read.

loop:   LD(r1, 0, r0)          | r0 <- Mem[Reg[r1]]
        ADD(r3, r1, r1)      | r1 <- r1 + increment
        CMPLT(r1, r2, r0) | if (r1 < r2)
        BT(r0, loop)        | then goto loop

        HALT()
        BR(Start)

firstadr: LONG(epgm)          | 1st adr after pgm
lastadr:  LONG(0x10000)      | end of DRAM+1
increment:LONG(16)          | The "stride"

epgm = .
. = 0x10000
```

Start bsim, load and assemble ["/mit/6.004/code/cache1.uasm"](#) and switch to the cache

control panel.

Note: don't peek at the answers until you've run the experiments and tried to answer the questions on your own. If you can puzzle through the questions yourself, you'll actually figure how and why caches work, which, of course, is the point of the experiment.

- A. Configure the cache as ON, 1 word/line, 8 total lines, direct-mapped, write-through. Run the "cache1" test program by clicking on the RUN button and observe the cache hit rate and the total number of cycles needed to execute the program. Repeat the experiment for 2-way, 4-way and 8-way associative cache. Finally turn the cache off and take a final measurement. You'll notice that increasing the amount of associativity past a certain point does not improve performance. Examining the program, explain why this result is what one would expect. Compare the performance for 2-way and 4-way associative caches and give an explanation for what you see.

Hide Answer

Expected results:

	hit %	cycles
direct-mapped	69.9	45072
2-way	79.9	36893
4-way	79.9	36893
8-way	79.9	36893
off	0.0	81892

Each iteration of the loop involves five memory accesses: four instruction fetches and one data access. In a direct-mapped cache, instructions will occupy four contiguous lines and data accesses will alternately map between one line of these four and one of the other unused elements. Thus, the hit rate in the steady state will be $3.5/5$ or 70%. In a 2-way set associative cache, the instructions can occupy one subcache while data accesses use one line in the other subcache for a hit rate of $4/5$ or 80%. Increasing the associativity further will not help because all data accesses only occur once and will always miss.

- B. Now configure the cache as ON, 4 words/lines, 2 total lines, 2-way associative, LRU replacement, write through. As before, run the test program and observe the total number of cycles needed to execute the program. Repeat the experiment using 4 total lines with 2 words/line, and using 8 total lines with 1 word/line. Explain why, for this program, smaller cache lines produce better performance.

Hide Answer

Expected results:

	hit %	cycles
4 words/line, 2 total lines	79.9	49152
2 words/line, 4 total lines	79.9	40973
1 word/line, 8 total lines	79.9	36893

The program is accessing every 16th data word in memory and those accesses always miss since each data word is accessed only once. Overall performance

decreases with increased block size because extra time is spent bringing in data words that aren't accessed by the program. The small improvement in hit rate with larger block size is due to the advantage of fetching the instructions all at once instead of word-by-word. Note that if the program had accessed every data word (instead of every 16th word), increasing the block size would have improved performance dramatically.

- C. Finally, configure the cache as ON, 1 word/line, 8 total lines, 2-way associative, write thru. Experiment with the four possible replacement strategies (LRU, FIFO, Random and Cycle). Explain why you might have expected the results you observed.

Hide Answer

Expected results:

	hit %	cycles
LRU	79.9	36893
FIFO	69.9	45077
Random	70.0	45025 (varies slightly from run to run)
Cycle	69.9	45077

For this test code, optimum performance is obtained when the data value is replaced in the cache. LRU guarantees this, while the others are divided 50/50.

- D. Now load and assemble the second test program, which can be found in /mit/6.004/code/cache2.uasm. Experiment with different cache configurations and report the one that has at least a 50% hit rate and the best price/performance ratio, i.e., the smallest value for (Total Cost)*(Cycles). Analyzing the program's memory access patterns will be a great help in finding the best cache configuration. Why does selecting a "write-back" write policy make such a large performance improvement for this program?

Hide Answer

With the stated cost metric and 5 minutes of experimenting, my best result was with a 2-way/LRU/write-back cache with 8 total lines and 2 words/line. This yielded a 80% hit rate and 40078 total cycles for a relatively small cost of 11756.

Even though each word is written only once, a write-back strategy is still a big performance boost. If you have multiword cache lines, a write-back cache will accumulate changes for each word in the block and then do one memory transaction to write the block back to memory when it gets thrown out of the cache. So even if you only write each word once, you still see a savings on amortizing the memory latency over each word in the block (write-thru would cost you the memory latency as each word is written individually).