CS 520: Programming Assignment #1

Zachary Seymour

October 21, 2013

1 Problem 1

As can be seen in Figure 1, for each of the benchmarks, the performance (measured by instructions per cycle) increases as we increase the length of the issue queue. After it exceeds between sixteen and twenty entries in length, though, the performance increases essentially level out. Because of instruction latency and dependencies, there will reach a point where only a finite number of instructions will be available to execute in each cycle. For the given benchmarks, it is likely this point has been reached where we seem the performance gains drop off. Therefore, no matter how many instructions we allowed to enter the issue queue, we have maxed out the amount that can become ready to execute per cycle.

2 Problem 2

In Figure 2, we can see the effects of changing various cache parameters on the performance of the processor and its success on finding data in the cache memory. When we alter the cache size, there is a hit to IPC when the cache is very small, under 8K or so. We also see the hit rate jump dramatically for most of the benchmarks past this point, from under 80% to 95% or more. Taking these two facts together it would seem the IPC performance takes a hit from the latency of directly accessing memory rather than getting data from the cache, since small cache size means high turnover.

Altering cache associativity does not have as dramatic an effect on either performance or cache hit rates. Increasing associativity likely benefits throughput at first because it is lowering cache miss latency, as in the first part. However, the gains are likely offset by the latency caused by searching all possibly locations the data could have been stored due to associativity. As expected, we seem tapering but constant gains in hit rate from increasing the associativity, as there are more places in the cache to potentially store a given piece of data.

Finally, changing the cache line size, while seemingly have little effect on performance caused great gains in hit rate. With fixed cache size, increasing the cache line size decreases the number of blocks that can be stored in the cache. However, it does increase the amount of data stored and each block, as

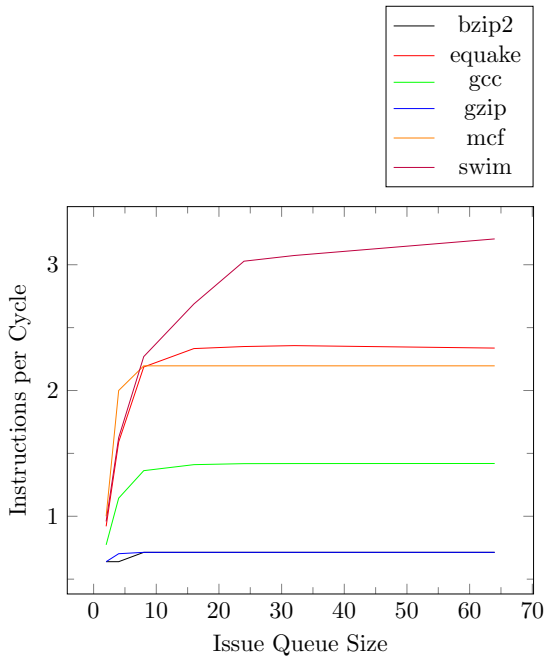
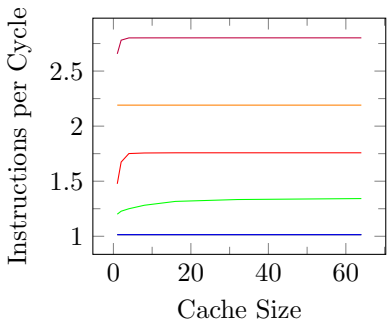


Figure 1: Issue Queue Length vs. IPC for various benchmarks

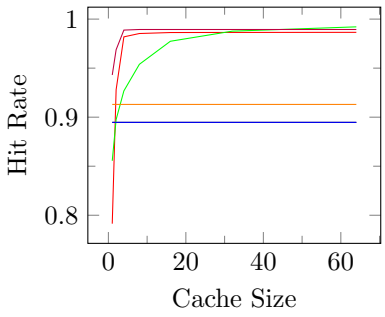
well as the amount of extra data stored in the cache that may be needed in the near future. Consequently, we get much higher hit rates with long cache lines, since, when we have a miss, the cache also brings in more data around the missed data, which is also likely to be used soon.

3 Problem 3

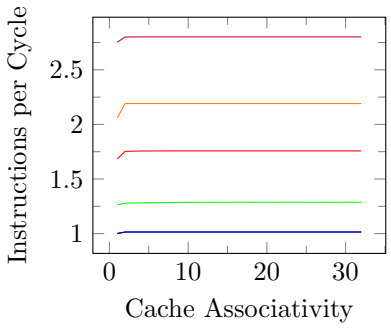
The data in Table 1 shows the performance results of implementing the various priority algorithms. The first column shows the data from the oldest-first method, which is the one implemented by default and, thus, our baseline for comparison. On average, every other algorithm performed worse in terms of instructions per cycle, although the difference between oldest-first and loads-first is negligible. This is most likely because, when a `LOAD` was not preselected to be enqueued, the algorithm proceeded as normal. The longer-latency algorithm performed the worst out of the four. By giving priority to instructions with long latency, we likely clogged up the pipeline waiting for lengthy instructions to execute. Similarly, the position-based algorithm probably underperformed because it did not take into account how long an item had been in the issue queue. For older items, there is a high likelihood another instruction is awaiting their completion, so better performance can be obtained by executing these earlier.



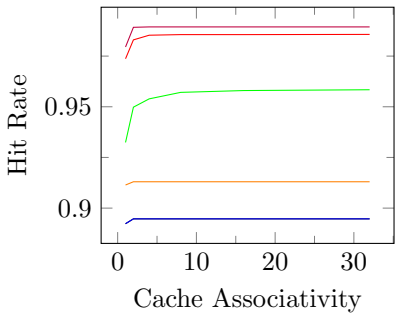
(a) Impacts of changing cache size on IPC.



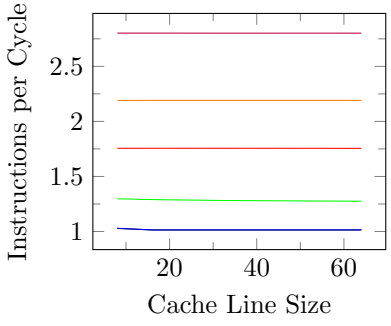
(b) Impacts of changing cache size on hit rate.



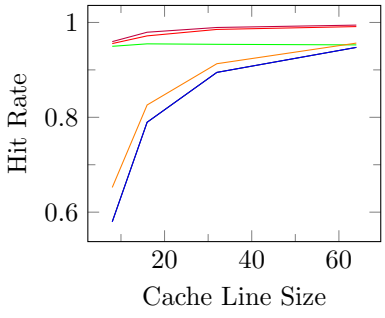
(c) Impacts of changing cache associativity on IPC.



(d) Impacts of changing cache associativity on hit rate.



(e) Impacts of changing cache line size on IPC.



(f) Impacts of changing cache line size on hit rate.

Figure 2: Performance impacts of various changes to the cache. (Legend as in Figure 1)

Benchmark	Oldest-first	Position-based	Longer-latency	Loads-first
bzip2	1.0144	1.0291	1.0290	1.0144
equake	1.9094	1.5693	1.5660	1.9030
gcc	1.4399	1.3533	1.3297	1.4398
gzip	1.0144	1.0291	1.0290	1.0144
mcf	2.1908	2.1486	2.1486	2.1908
swim	3.0173	2.7756	2.7497	3.0173
Average	1.7644	1.6508	1.6420	1.7633

Table 1: Results of different priority algorithms for ready queue (shown in instructions per cycle).

4 Problem 4

The chart shown in Figure 3 shows the performance impact of pipelining the wakeup/select stages in the processor. I implemented this in simulation by reversing the calls to `wakeup()` and `selection()` so that `selection` was called first. This introduced the necessary bubble cycle between the stages. As can be expected, across most of the benchmarks, this change degraded performance. With instructions not being selected until a cycle after they are woken, the throughput of the processor drops across all benchmarks. The bzip2, gzip, and mcf benchmarks all just rely on integer arithmetic, so this is likely why there were no noticeable hits to their performance.

A Code Listings for Problem 2

A.1 Oldest-first

```
1  static void
2  readyq_enqueue(struct ROB_entry *rs)        /* RS to enqueue */
3  {
4    struct RS_link *prev, *node, *new_node;
5
6    /* node is now queued */
7    if (rs->queued)
8      panic("node is already queued");
9    rs->queued = TRUE;
10
11
12   assert(all_operands_spec_ready(rs));
13
14
15   /* get a free ready list node */
16   RSLINK_NEW(new_node, rs);
17   new_node->x.seq = rs->seq;
```

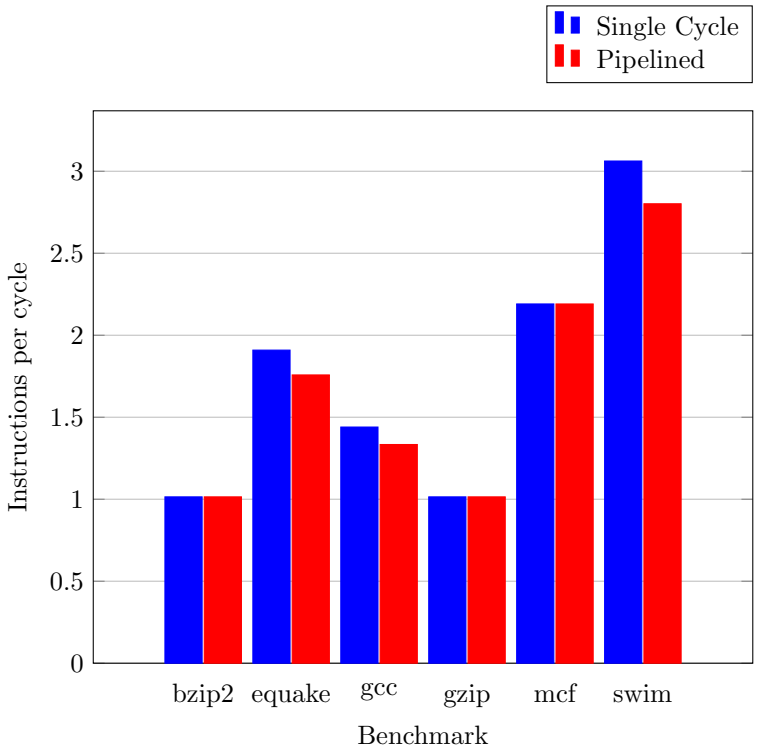


Figure 3: Performance for each benchmark with single-cycle or pipelined wakeup/select stages.

```
18
19    /* otherwise insert in program order (earliest seq first) */
20    /* OLDEST FIRST ISSUE */
21    for (prev=NULL, node=ready_queue;
22         node && node->x.seq < rs->seq;
23         prev=node, node=node->next);
24
25    if (prev)
26      {
27        /* insert middle or end */
28        new_node->next = prev->next;
29        prev->next = new_node;
30      }
31    else
32      {
33        /* insert at beginning */
34        new_node->next = ready_queue;
35        ready_queue = new_node;
36      }
37  }
```

A.2 Position-based

```
1   static void
2   readyq_enqueue(struct ROB_entry *rs)        /* RS to enqueue */
3   {
4     struct RS_link *prev, *node, *new_node;
5
6     /* node is now queued */
7     if (rs->queued)
8       panic("node is already queued");
9     rs->queued = TRUE;
10
11
12    assert(all_operands_spec_ready(rs));
13
14
15    /* get a free ready list node */
16    RSLINK_NEW(new_node, rs);
17    new_node->x.seq = rs->seq;
18
19    /* otherwise insert in program order (earliest seq first) */
20    /* OLDEST FIRST ISSUE */
21    for (prev=NULL, node=ready_queue;
22         node && node->x.seq < rs->iq_entry_num;
23         prev=node, node=node->next);
```

```
24
25    if (prev)
26      {
27        /* insert middle or end */
28        new_node->next = prev->next;
29        prev->next = new_node;
30      }
31    else
32      {
33        /* insert at beginning */
34        new_node->next = ready_queue;
35        ready_queue = new_node;
36      }
37  }
```

A.3 Longer-latency first

```
1   static void
2   readyq_enqueue(struct ROB_entry *rs)        /* RS to enqueue */
3   {
4     struct RS_link *prev, *node, *new_node;
5
6     /* node is now queued */
7     if (rs->queued)
8       panic("node is already queued");
9     rs->queued = TRUE;
10
11
12    assert(all_operands_spec_ready(rs));
13
14
15    /* get a free ready list node */
16    RSLINK_NEW(new_node, rs);
17    new_node->x.seq = rs->seq;
18
19
20    /* otherwise insert in program order (earliest seq first) */
21    /* OLDEST FIRST ISSUE */
22    for (prev=NULL, node=ready_queue;
23       node && node->rs->exec_lat > rs->exec_lat;
24       prev=node, node=node->next);
25
26    if (prev)
27      {
28        /* insert middle or end */
29        new_node->next = prev->next;
```

```
30        prev->next = new_node;
31      }
32    else
33      {
34        /* insert at beginning */
35        new_node->next = ready_queue;
36        ready_queue = new_node;
37      }
38  }
```

A.4 LOAD first

```
1     static void
2   readyq_enqueue(struct ROB_entry *rs)                /* RS to enqueue */
3   {
4     struct RS_link *prev, *node, *new_node;
5
6     /* node is now queued */
7     if (rs->queued)
8       panic("node is already queued");
9     rs->queued = TRUE;
10
11
12    assert(all_operands_spec_ready(rs));
13
14
15    /* get a free ready list node */
16    RSLINK_NEW(new_node, rs);
17    new_node->x.seq = rs->seq;
18
19    if (/* load? */
20            ((rs->op) & (F_MEM|F_LOAD)) == (F_MEM|F_LOAD))
21    {
22      /* insert at beginning */
23        new_node->next = ready_queue;
24        ready_queue = new_node;
25    }
26    /* otherwise insert in program order (earliest seq first) */
27    /* OLDEST FIRST ISSUE */
28    else
29    {
30    for (prev=NULL, node=ready_queue;
31       node && node->x.seq < rs->seq;
32       prev=node, node=node->next);
33
34    if (prev)
```

```
35      {
36        /* insert middle or end */
37        new_node->next = prev->next;
38        prev->next = new_node;
39      }
40    else
41      {
42        /* insert at beginning */
43        new_node->next = ready_queue;
44        ready_queue = new_node;
45      }
46    }
47  }
```