ECE 621 HOMEWORK #4

Tom Riley (20653444)

Caches and Memory Hierarchy

1. Changing cache line length

If you increase the cache line length, that will decrease the number of compulsory misses. This is because each cache request will fetch more addresses, which acts similar to prefetching those addresses. For example, with a block size of 1 byte, if you request the address 3 and then 0, both will be compulsory misses. If you increase the block size so it is 4 bytes or greater, then both addresses will be in the same block, and the second request to the block will hit.

If you increase the cache line length and keep the cache size constant, that likely increase the number of capacity misses, but it can decrease the amount of capacity misses. Since the block size is larger, there will be fewer frames in the cache. For example, say you have a cache of 4 bytes, and you increase the block size from 1 byte to 2 bytes. If you request the bytes 0, 3, 7, 4, then 0, that final request for 0 will hit if the cache size is 1 byte, since the cache will contain 0,3,4,7. It will miss if the cache line is increased to 2 bytes, since the cache will instead contain 4,5,6,7. Consider though the example 0,2,3,7,1,0. In this case the request will hit with the larger cache line size since it will be pulled back into the second request for another byte to the same block, whereas it would have been washed out by those requests with the smaller cache line size. In general, the relationship is complicated by both the fact that removing compulsory misses can uncover capacity misses, and the fact that the change can both increase and decrease the number of capacity misses in different cases.

If you increase the cache line length, and keep the ways and capacity constant, there will be fewer sets, since there will be fewer frames. The relationship with conflict misses is complicated, as removing compulsory or capacity misses can uncover conflict misses. For example, assume the same 4 byte cache from above but assume it is direct mapped with a 1 or 2 byte block size. If you assume the request sequence 4,0,5,4 that final request will be a conflict miss with a block size of 1, but the address 4 will be pulled in by the request for 5 with a block size of 2, in which case it will hit. If you consider the request sequence 0,1,5,0, with a block size of 1 the final request will hit, but with a block size of 2 it will be a conflict miss since the request for 5 will evict it.

Thus increasing the block size will definitely reduce compulsory misses, but the relationship with the other miss types is more complex and depends on other qualities, like the cache size, the access patterns, etc.

2. Simulating cache requests

I am assuming that memory locations are byte addressed, i.e. the address 2 refers to the second byte in memory. Since the highest address seen is 42, without loss of generality I will assume that addresses are 6 bits long, so we have the address stream:

2	0b000010	18	0b010010
7	0b000111	40	0b101000
10	0b001010	12	0b001100
34	0b100010	42	0b101010
5	0b000101	35	0b100011

Since the cache is 32 bytes, and the blocks are 8 bytes, the cache will have 4 frames, and since it is direct mapped each frame will form its own set. Since a block is 8 bytes, the block offset will be 3 bits (highlighted red), and since there are 4 sets the index will be 2 bits (bolded).

a) Categorizing cache requests

2	Request block 00, tag 0	Compulsory miss	-
7	Request block 00, tag 0	Hit	-
10	Request block 01, tag 0	Compulsory miss	-
34	Request block 00, tag 1	Compulsory miss,	evict tag 0
5	Request block 00, tag 0	Capacity miss (hit in victim buffer)	evict tag 1
18	Request block 10, tag 0	Compulsory miss	-
40	Request block 01, tag 1	Compulsory miss	evict tag 0
12	Request block 01, tag 0	Capacity miss (hit in victim buffer)	evict tag 1
42	Request block 01, tag 1	Conflict miss (hit in victim buffer)	evict tag 0
35	Request block 00, tag 1	Capacity miss	evict tag 0

b) Adding victim buffer

Since the victim buffer is 16 bytes, it could hold two blocks.

The request for address 5 will hit in the victim buffer, as it was evicted by the request for 34

The request for address 12 will hit in the victim buffer, as it was evicted by the request for 40

The request for address 42 will hit in the victim buffer, as it was evicted by the request for 12

c) Next-block prefetching

2	Request block 00, tag 0	Compulsory miss	Prefetch block 01, tag 0
7	Request block 00, tag 0	Hit	-
10	Request block 01, tag 0	Hit	Prefetch block 10, tag 0
34	Request block 00, tag 1	Compulsory miss	Prefetch block 01, tag 1
5	Request block 00, tag 0	Capacity miss	Prefetch block 01, tag 0
18	Request block 10, tag 0	Hit	Prefetch block 11, tag 0
40	Request block 01, tag 1	Capacity miss	Prefetch block 10, tag 1
12	Request block 01, tag 0	Capacity miss	Prefetch block 10, tag 0
42	Request block 01, tag 1	Conflict miss	Prefetch block 10, tag 1
35	Request block 00, tag 1	Capacity miss	Prefetch block 01, tag 1

By prefetching:

The request for address 10 will hit, as it was prefetched by the request for 2.

The request for address 18 will hit, as it was prefetched by the request for address 10.

The request for address 40 will still miss, but it will no longer be compulsory as it was prefetched by the request for 34, but was evicted by the prefetch after the request for 5.

3. Why not strict LRU?

In order to implement LRU, you would need to know the ordinal age within its set of each block in the cache. If there are n frames in the cache, it is s way set associative and if the tag length was m bits, you would need at least min(n*m, n*lg(s)) memory. If m < lg(s), then you could essentially have a queue with each cache block holding a pointer to the next younger tag in its set. If lg(s) < m, then you could store a counter for each frame in the set that you increment above the largest value each time you add a new cache block. The easiest way I can see to do this would be to let the values wrap around and just keep a pointer to what the lowest/highest value is.

Both of these solutions require extra memory and have a performance cost. To put numbers to things, the textbook discusses 64 KB two-way set associative cache with 64 B blocks. This cache uses a 40 b physical address, and thus has a 25 b tag (=40 – lg(64K/2)). Clearly 25 >> lg(2). Strictly following the plan laid out above would require setting up a one bit counter for each frame, and then incrementing it and having a second counter that keeps track if 0 or 1 is bigger. Since the counters are just one bit, it is probably simpler to skip the one bit counters and just keep a single one bit counter for each set that says if frame 0 or 1 is older. With this low associativity, implementing strict LRU would thus only require 512b=64B extra memory or the equivalent of a single cache block. If instead the associativity were increased to fully associative, the set would be 1024 blocks. Thus we would need 1024 ten bit counters to track the ages, as well as two extra to record the oldest and youngest age if we want to let the counters wrap around for performance. This means we would need 10260 b = 1282.5 B =~ 21 additional cache blocks, approximately extra 16% overhead. Clearly the overhead is much greater for higher associativity, even ignoring the performance overheads.

An alternative is pseudo LRU, which is similar to my counter idea above. In the case of a two way set associative memory, the two are almost identical, but even for larger associativity only a 1 b counter is used. This means the overhead will always be 1/cache block size. On access, the age of the access block is set (incremented without wrapping). On assertion of the last block in a set, all the other blocks in that set have their ages reset. When a block needs eviction, one of the oldest blocks is evicted, using some other strategy to pick from them, such as just randomly picking one. This method clearly is less memory intensive for caches with high associativity.

4. Speculative memory

The Intel Itanium processors implement the IA-64 architecture. These processors and this architecture implement memory disambiguation through the use of advanced loads. In this scheme, ambiguous loads are allowed to be speculatively scheduled ahead of older stores. Since the dependency between the instructions through memory is ambiguous, the Itanium processor features a table called the ALAT, which stores the destination register and the address (see page H-40 of the textbook), as well as a valid bit that is initially set by the load. If a store is executed while the load is still speculative, it will look up its address in the table, and invalidate any matching entries. A special check instruction is placed later to check for mis speculation, by evaluating the entry in the ALAT. The simplest instruction can be used if just the load was speculative, and it just reexecutes the load on a misspeculation. If there are also speculative dependant instructions, then a separate check is used which includes a subroutine to fix up the misspeculation.

An alternative approach uses a load queue and a store queue. The basic idea is that in flight loads and stores are written to local respective queues rather than going directly to the data cache. On a load, a new entry is allowed in the load queue at the tail of the queue, and the current tail of the store queue is recorded in the entry. The same is done in the store queue, meaning the core has a full record of the ordering of all loads and stores. Loads are allowed to execute speculatively by taking the value of the youngest store to a matching location in the store queue (or the data cache if there are no entries), while recording its address in the load queue. Once all younger instructions than the load are committed, the load can be removed from the queue indicating the speculation was successful. On a store, the data is written to the store queue, but the load queue is also checked for speculative loads to the same address. If there are any speculative loads to the same address that are younger than the store, that indicates a memory violation, and the reorder buffer must be walked back to the oldest speculative load with a matching address younger than the store.

Both approaches are similar in the sense that they track speculative loads and then check for misspeculation on subsequently executed stores that are older in program order. In the latter approach, the memory violation is implicitly dealt with as soon as it is detected, while the IA-64 waits for explicit check instructions. An advantage of the advanced load approach is simpler hardware is needed with just a simple table compared to two queues. I suspect that this approach would also allow for faster loads and stores since less extra work is being done. This approach requires extra instructions though, even without a misspeculation.

Exploring Caches with SimpleScalar

5. SimpleScalar Cache

Experiment 1: Associativity vs hit latency

For the experiment, a 4 KB cache size was used with 32 B blocks, and an LRU replacement policy. Thus, we have the following conditions:

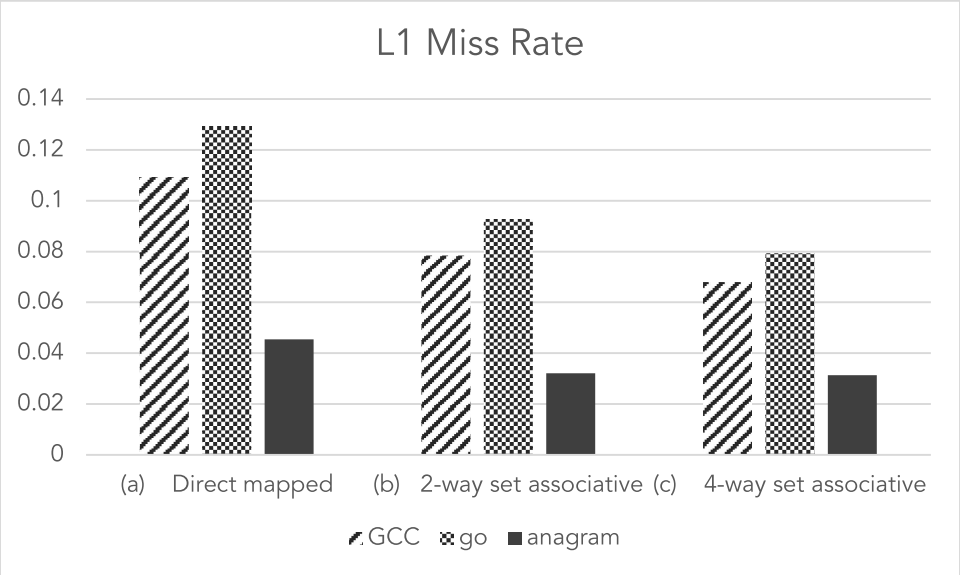
Condition	Total cache size (B)	Number of sets	block size (B)	associativity (frames per set)	replacement policy	Cache latency (cycles)
(a)	4K	128	32	1	LRU	2
(b)	4K	64	32	2	LRU	3
(c)	4K	32	32	4	LRU	4

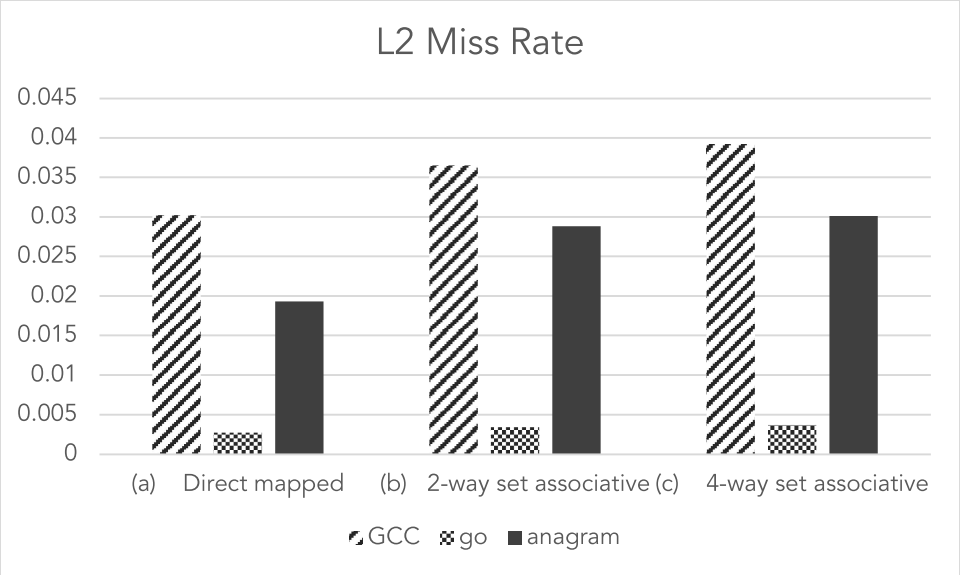
Note that it is assumed that the there is a unified instruction and data 512KB 4-way set associative L2 with 64 B blocks, and a 6 cycle hit latency, and that the main memory has a 18 cycle latency for the first chunk and 2 cycles for subsequent chunks, with each chunk being 8 B. For simplicity I will assume that L2 misses take a constant latency of 18+7*2=32 cycles, ignoring the possibility of a page fault. Thus, the total memory latency will be calculated as:

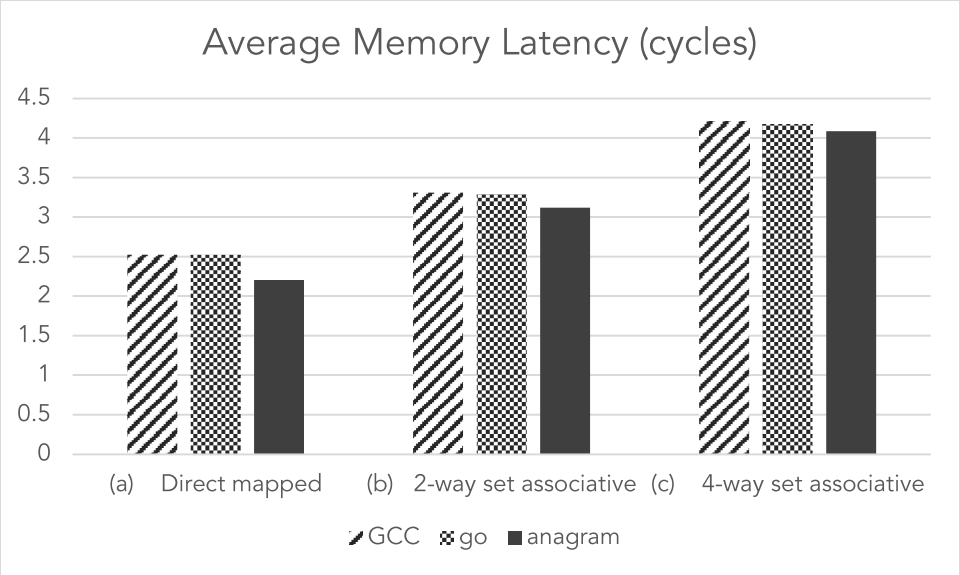
$$\text{Average memory latency} = (1 - \text{L1 miss rate}) \times \text{L1 hit latency} + \text{L1 miss rate} \times ((1 - \text{L2 miss rate}) \times 6 + \text{L2 miss rate} \times 32)$$

Where the L1 miss rate and L2 miss rate will be measured by SimpleScalar, and the L1 hit latency will be either 2, 3, or 4.

The following charts show the results:



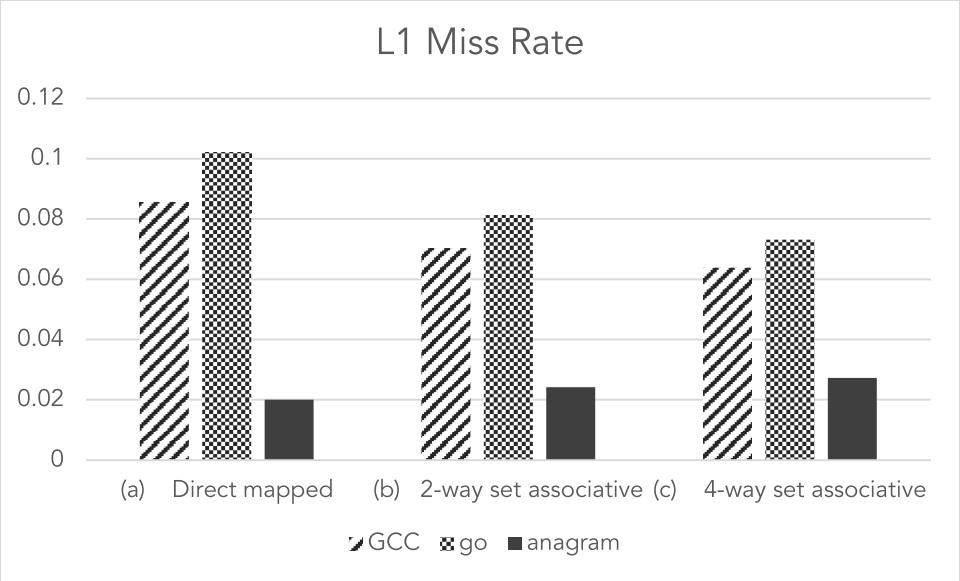


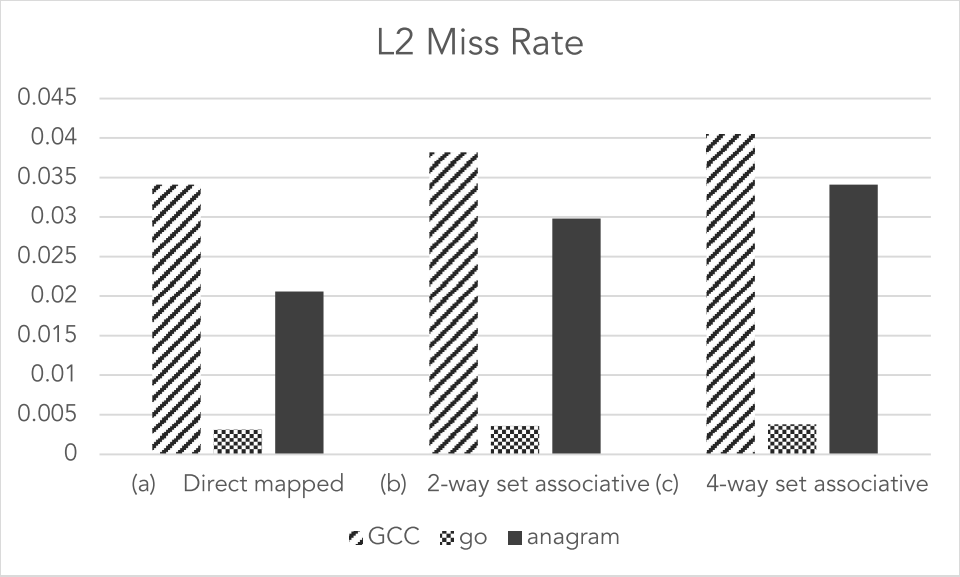


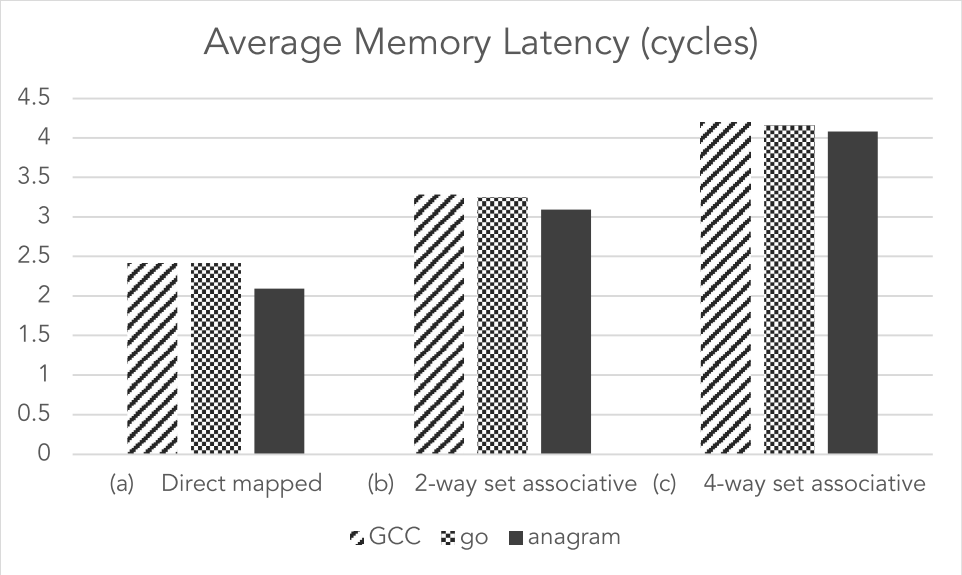
These experiments show that while the higher associativity decreased the overall L1 miss rate as we might expect, the effect was not large enough to overcome the larger latency and higher L2 miss rate. This demonstrates why it is so important to prioritize the L1 hit latency.

Experiment 2: Adding 2-entry supplementary cache

For this experiment, all parameters were kept the same as in Experiment 1, and the latency was measured in the same way. The only change was the addition of a two-entry victim cache. This was implemented by adding two blocks of storage to the L1 cache. In the simulation, after it checks for a hit to the L1 data cache it checks for if the block is in the victim cache, and if it is then that is treated as a hit. When a cache is to be replaced, the older block in the victim cache is replaced with the younger block, and the younger block is replaced with the victim block. This gave the following results:


L1 Miss Rate
0.12
0.1
0.08
0.06
0.04
0.02
0
(a) Direct mapped
(b) 2-way set associative
(c) 4-way set associative
GCC
go
anagram


L2 Miss Rate
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
(a) Direct mapped
(b) 2-way set associative
(c) 4-way set associative
GCC
go
anagram

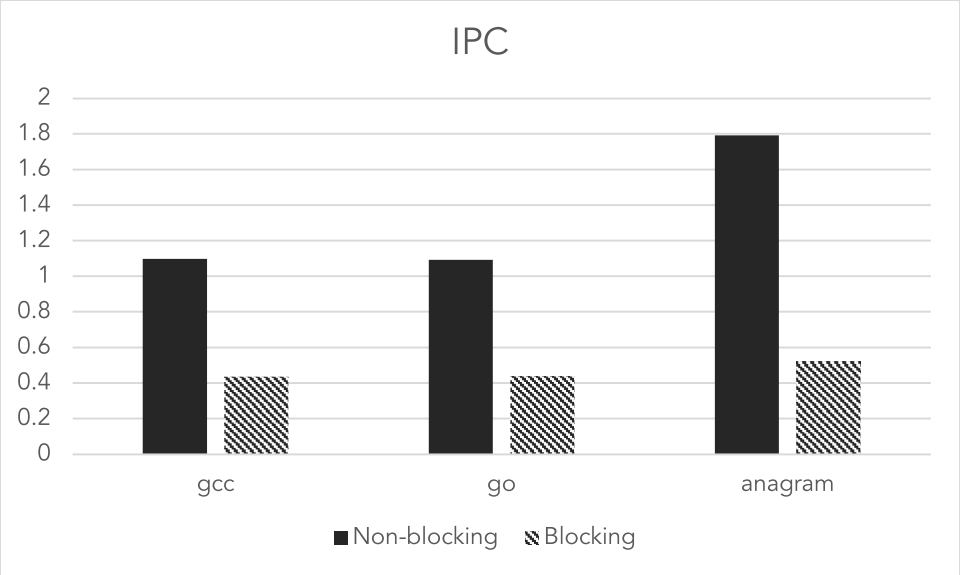


Note that the addition of the victim cache lowered the average memory latency in all cases, as you would expect since it is adding 1.5% extra cache memory. Also, as expected, the effect was much greater on the direct mapped cache, with a 4% decrease in average latency for gcc and go, and a 5% decrease for anagram. On the other hand, there was less than 1% decrease for the two-way caches, and less than a 0.3% decrease for the 4-way caches, likely not worth the extra memory. In all cases the lowest cache latency was still with direct mapped caches.

Experiment 3: Blocking cache

SimpleScalar's out of order execution model uses an event based cycle accurate simulation method. It iterates cycles by cycle, using an event queue to wait for certain operations (usually updating registers). To my understanding, events such as cache accesses happen instantly, but the registers are not updated instantly, instead having their updates added to the event queue to happen at a certain cycle.

A blocking cache can only service one request at a time, and if it is currently processing a request it will not be able to service a second request. In practice this would be implemented using busy signals. To implement this in simulation, a field was added to each cache to record the simulation cycle when the cache would be ready. When the cache is accessed and the latency is calculated, the latency is added on top of the current ready time and the latency returned is the difference between the new ready time and the current time. The results for the direct mapped cache from the previous experiments is shown below. Note that the results for other associativities are very similar.



Clearly using a blocking cache decimates the performance. This is expected, since the previous model using a non-blocking cache allowed independent memory accesses to overlap, and could thus hide a lot of the memory stall cycles.