Paul Mercurio
SID: 22929618
Login: cs152-ch

Lab 2 Report: Directed Portion

2.2

(Q1)

Bzip2

L1 I$ Miss Rate: 0.000%. L1 D$ Miss Rate: 1.952%. L2$ Miss Rate: 13.749%.

Mcf

L1 I$ Miss Rate: 0.000%. L1 D$ Miss Rate: 23.933%. L2$ Miss Rate: 26.851%.

Sjeng

L1 I$ Miss Rate: 0.079%. L1 D$ Miss Rate: 2.837%. L2$ Miss Rate: 40.909%.

Lbm

L1 I$ Miss Rate: 0.000%. L1 D$ Miss Rate: 1.008%. L2$ Miss Rate: 35.882%.

Overall, the Bzip2 bechmark has the best cache performance, with Sjeng having the worst, except for the L1 D$ case where Mcf is far worse. However, Sjeng is the only benchmark to display any L1 I$ misses and has by far the highest L2$ miss rate.

(Q2)

Cache access time for both L1 D$ and L1 I$ is 0.61ns. Cache access time for D2$ is 0.99ns.

(Q3)

The L1 access time is 10ps slower than the critical path length of non-memory instructions, therefore memory instructions determine the cycle time: 610ps.

(Q4)

Bzip2 CPI = 1.2 + (1898 + 3604709/1096862410) x 100/.61 = 1.739 cycles/instruction.

Mcf CPI = 1.2 + ((851 + 3228836 + 89161627)/2306949412) x 100/.61 = 7.765 cycles/instruction.

Sjeng CPI = 1.2 + (736981 + 4405218 + 648851)/992050671) x 100/.61 = 2.157 cycles/instruction.

Lbm CPI = 1.2 + ((798 + 6879063 + 17439203))/11793984289) x 100/.61 = 1.538 cycles/instruction.

(Q5)

Bzip2 L2$ AMAT = .99 + (.13749 x 100) = 14.739ns.

Mcf L2$ AMAT = .99 + (.26851 x 100) = 27.841ns.

Sjeng L2$ AMAT = .99 + (.40909 x 100) = 41.899ns.

Lbm L2$ AMAT = .99 + (.35532 x 100) = 36.522ns.

(Q6)

Bzip2 CPI = 1.2 + ((1898 + 3604709)/1096862410) * 14.739/.61 = 1.279

Mcf CPI (with L2$) = 1.2 + ((851 + 3228836 + 89161627)/2306949412) * 27.841/.61 = 3.028

Sjeng CPI (with L2$) = 1.2 + ((736981 + 4405218 + 648851)/992050671) * 41.899/.61 = 1.601

Lbm CPI (with L2$) = 1.2 + ((798 + 6879063 + 17439203)/11793984289) * 36.522/.61 = 1.323

(Q7)

The L2$ cache improves performance for all of the benchmarks, with an average CPI reduction of 1.499 cycles/second.

2.3

For bzip2 the miss rate converged to .213% once associativity reached 512-ways in a (2048:512:128) configuration. From 1-way to 256-ways, miss rate kept decreasing as conflict misses converged and stopped at 512-ways, implying that the size of the cache at this point is an upper-bound on the working-set size for the bzip2 benchmark. Multiplying out, we get 2048*512*128 = 128MB working set size for bzip2.

Following similar reasoning, we get the following working set sizes for the other benchmarks:

MCF: Miss rate converges to 0.259% at configuration of 8192*4096*128 = 4GB working set size.

SJENG: Miss rate converges to

2.4

Given that the cycle time for this pipeline is 610ns, we don't want to increase this value because it will then affect all instructions and stages, not just instruction fetches. With that in mind, most configurations can be immediately thrown out, leaving only: (1024:1:64), (512:1:64), (256:1:64) and (128:2:64) with access times of 530ps, 410ps, 340ps, and 610ps, respectively. Of these remaining four configurations, only one exhibited a miss rate of effectively 0% across all benchmarks, and therefore not increasing CPI at all. This

configuration is: (1024:1:64), so this is the configuration we want to use for the L1 I$. The low miss rate generated by this direct-mapped configuration makes sense because of the low associativity between different instructions. Conflict misses are rare because most instructions are only used once, in which case their being evicted from the cache is relatively harmless because we probably won't need to access it again. In the few cases where instructions are revisited (branch and jump instructions) this might pose a problem, but even then the number of misses is small because in a loop, for example, when the conditional branch is not taken, it simply jumps back to an instruction that is close to itself (possibly even in the same cache line, where the larger cache of 64KB and larger line size of 1024B increases those odds in this case) and will almost always hit in this case. As for branches being taken and unconditional jumps, whether or not a miss occurs is relatively random, but in the scheme of things this number is extremely small in comparison to the large number of total instructions fetched and executed. The drawback to this configuration, however, is the larger amount of silicon (.73mm^2) used. By just switching to the (512:1:64) configuration, the amount of silicon decreases by almost 53% to .33mm^2 with the only catch being a 0.014% miss rate added to the sjeng benchmark. Taking cost and die size into account, this change to the 512:1:64 L1 I$ cache configuration is probably worth it, but physical size configuration is not important here as it is for the D$.

2.5

Again, taking into account the fact that a configuration, which increases cycle time, will slow down ALL instructions (not just loads/stores) we want to choose from the same four configurations we chose from in part 2.4. Assume we are using the I$ recommended without consideration of physical size. Again, the 1024:1:64 configuration has the smallest average miss rate across the four benchmarks at: (1.421+16.454+2.626+.833)/4 = 5.3335% (effectively zero I$ misses at our level of accuracy so only D$ misses appear in this calculation). Now, average CPI can easily be calculated with the following formula: CPI = 1.2 + (5.3335/100)*(100/.61) = 9.943. Of course because we are using the largest cache size (64KB) again, physical size becomes an issue. Additionally, unlike the I$, data cache accesses don't happen on almost every cycle so performance isn't quite as important. For example, for the bzip2 benchmark, only 16.83% of the instructions are memory instructions: 107884368 D$ read accesses + 76759267 D$ write

accesses divided by 1096862410 I$ read accesses (total instructions) = .1683 = 16.83%. Because of this, it is probably worth it to use a smaller cache at the expense of a small increase in CPI for cost purposes.

2.6

Normally, the miss penalty for an L1 D$ miss without the L2$ is 100ns to access DRAM, so any configuration of the twelve listed will satisfy this constraint.

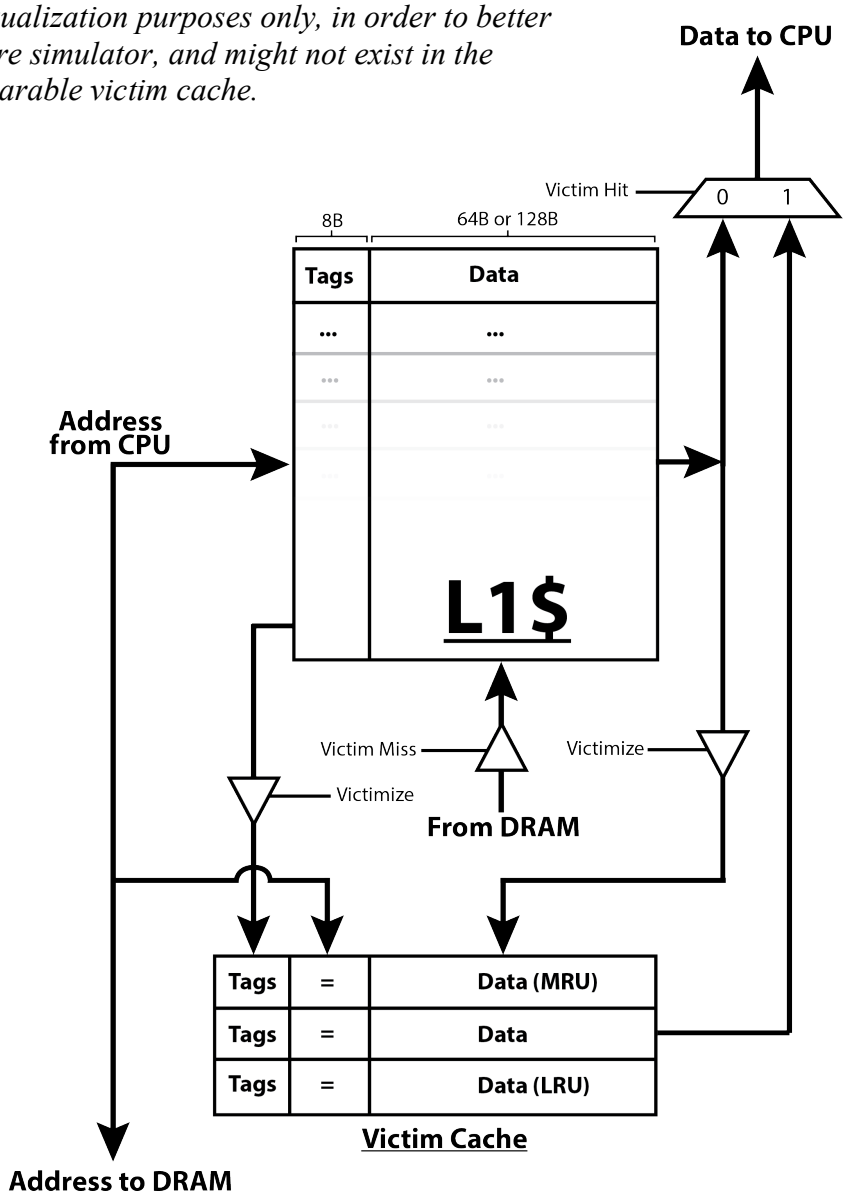
Open-Ended Portion 3.2: Victim Cache

Given the limited number of flip-flops available for this problem (only 2K, amounting to 2,048 bits of victim cache state), the only configuration that allows for more than one victim cache entry is 512:1:64. With this setup, each cache line is 64 bytes of data + 8 bytes for the tag and flag bits = 72 bytes/entry. In total, we can only use up to 256 bytes, so 256//72 = 3 victim cache lines. From here, several policies were implemented and tested. First, a list of the various implementations, along with a brief description of how each works and a hypothesis for why/how it might improve miss rate and how much it will do so by. Each implementation follows the 512:1:64 configuration unless otherwise specified:

1) No victim cache: Just ran the original cache simulator to get control data.
2) Fully associative victim cache with random replacement policy: Added a victim cache with 3 lines and randomly replaced entries when a new entry was added. I expected this to be one of the most effective implementations.
3) Fully associative victim cache with LRU replacement polity: Same implementation as #2 except each new entry was added to the "bottom" of the victim cache, thereby replacing the LRU line. Upon a victim cache hit, the "hit" entry is moved to the "top" of the victim cache, and the remaining two drop a level. I expected this to most likely be the most effective implementation of all.
4) Fully associative victim cache with MRU replacement policy: Same as #3 except the entry at the "top" of the cache is replaced with a new entry. I expected this policy to have very similar results to the random replacement policy.
5) Fully associative victim cache with random replacement and tag checker: Similar to #2 but in this case after the random index is generated, the tag of the new entry is compared to the tag of the entry it is about to replace to see if the two map to the same line in the L1$. The intuition here is to prevent two associated pieces of memory from repeated conflict misses. If they do match to the same address, a different index is chosen for the new entry. Given the limited size of the victim cache and wide range of address mappings I expected this policy to have very little improvement, if any.
6) Fully associative victim cache with random replacement and write-back to the L1$: The only difference between this and #2 is that in this case when a hit occurs on the victim cache the data is not only fed directly to the CPU, but it is also written back to the L1$ and the corresponding entry in the victim cache is flagged to be replaced next. I was not sure what effect this might have on miss rate, but upon analyzing the results it seems as if this method isn't great because the small amount of associativity provided by the victim cache is somewhat lost because out of any two tags that map to the same line, only one will be in either cache at any point in time.
7) Direct-mapped victim cache: Same 3 lines in size but direct-mapped instead of fully associative. I expected this implementation to out-perform the MRU policy, but by a small margin.

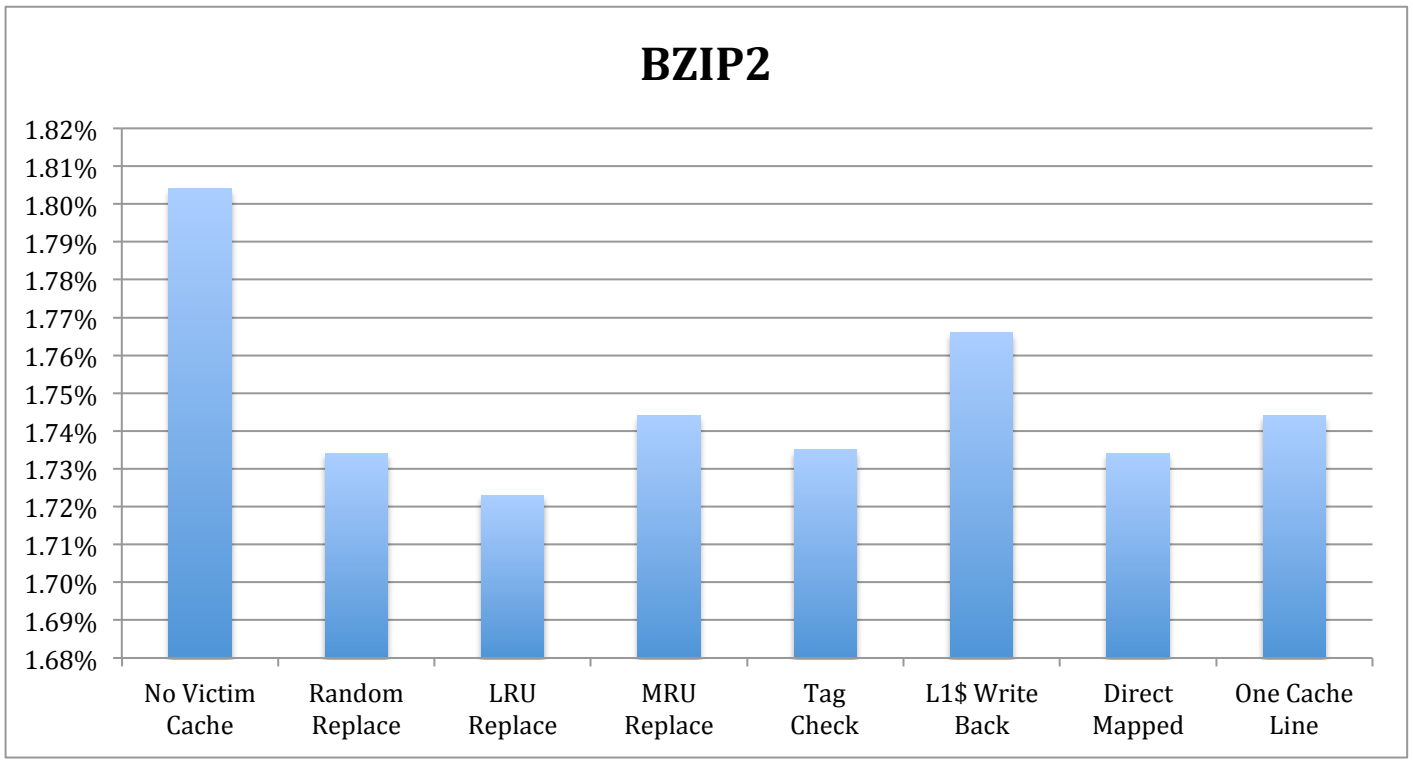
8) Victim cache with one line: mostly did this to compare against the 256:1:128 configuration but the idea here is that the victim cache is of size one, so I expected the results to be significantly worse than any of the implementations listed above.
9) L1$ configuration of 256:1:128 with no victim cache: A control to compare to this configuration having a victim cache. I expected this to perform similarly to #1
10) L1$ configuration of 256:1:128 with a victim cache that is one line in size. There is only one line to meet the 2K flip-flop constraint because the configuration involves a much bigger line size. I expected this to have very similar results to #8.

Pictured below is a victim cache diagram similar to the one in the victim cache paper by Norman P. Jouppi, but slightly more specific to my implementation, particularly the LRU replacement policy implementation. *Note: some of the visual aids (muxes, tri-state buffers, control signals) are for visualization purposes only, in order to better demonstrate what is going on in the software simulator, and might not exist in the actual hardware implementation of a comparable victim cache.*

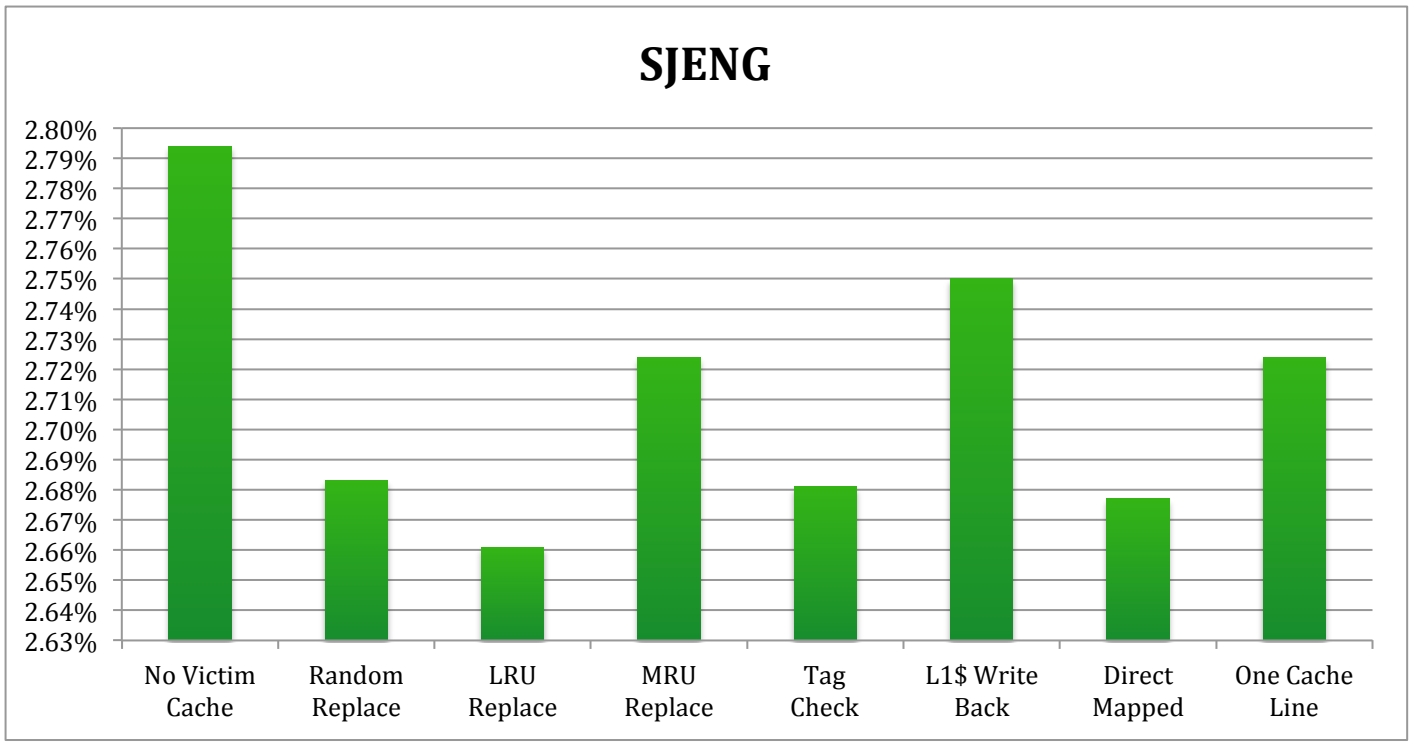


The Results

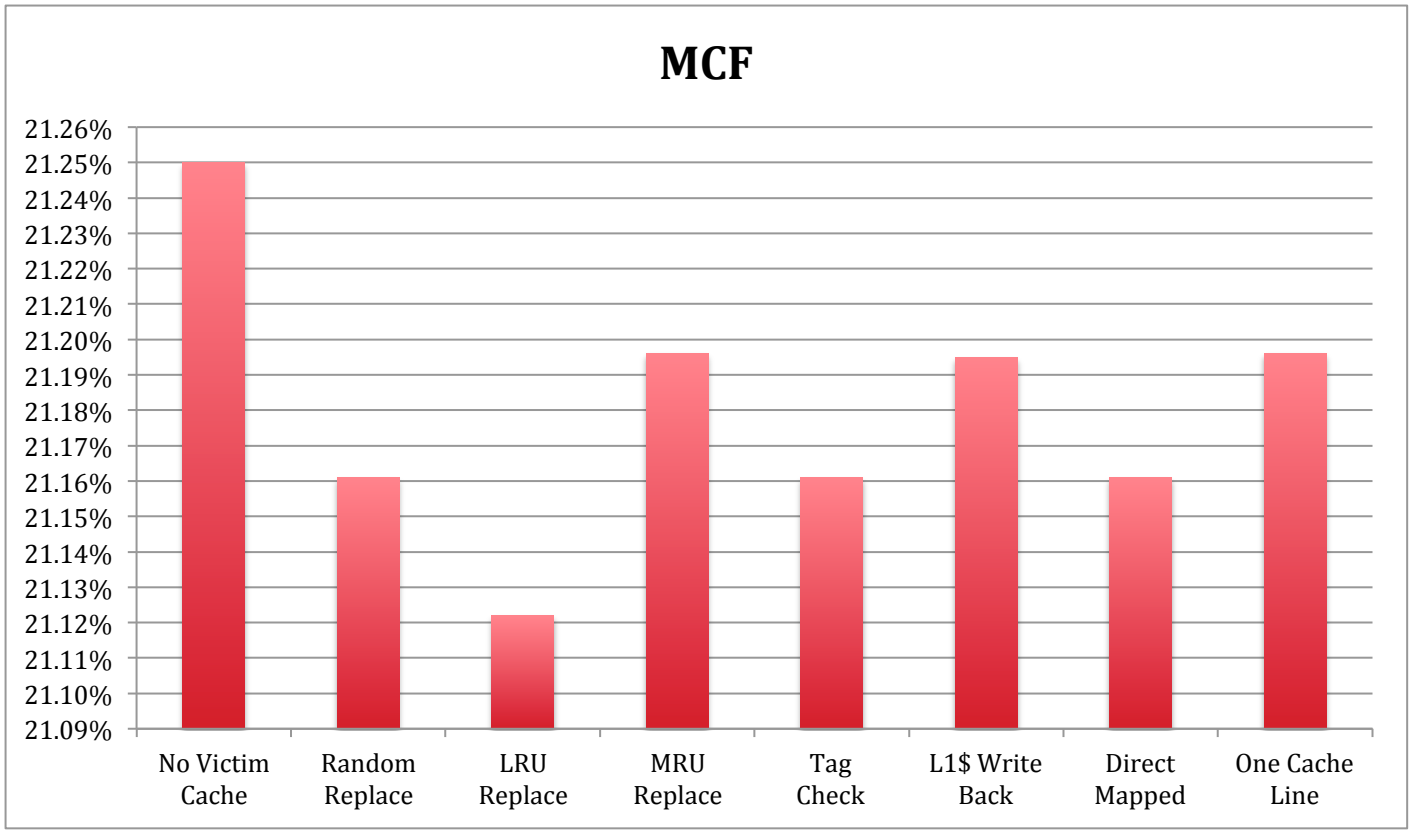
The four charts below show the performance of each implementation as a miss rate percentage for each of the four benchmarks with the 512:1:64 cache configuration. The two 256:1:128 configuration results were omitted from the charts for scaling purposes but are listed below each chart.



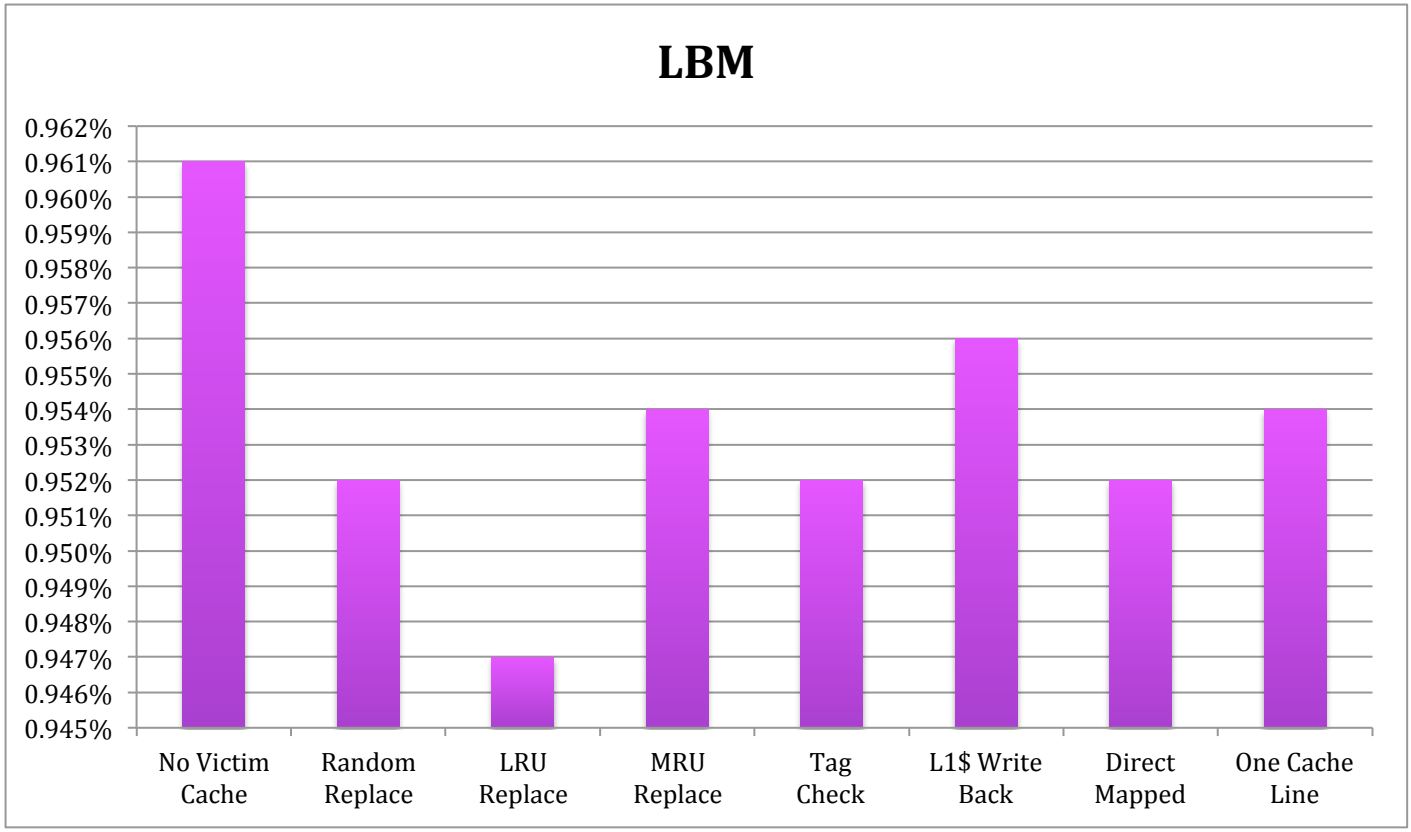
256:1:128 No Victim Cache MR: **1.849%**. 256:1:128 Victim Cache MR: **1.736%**



256:1:128 No Victim Cache MR: **1.753%**. 256:1:128 Victim Cache MR: **1.644%**



256:1:128 No Victim Cache MR: **20.446%**. 256:1:128 Victim Cache MR: **20.325%**



256:1:128 No Victim Cache MR: **0.557%**. 256:1:128 Victim Cache MR: **0.546%**

Conclusion

Without a doubt, the LRU replacement policy proved to be the most effective for each policy. Perhaps more interesting, however, is the fact that each benchmark exhibits the exact same pattern in terms of effectiveness of each implementation. The random replace, tag check and direct mapped implementations all performed almost identically, per benchmark. The reason for this being the fact that, given the non-associativity of the L1 cache, the likelihood of two tags mapping to the same line is very small, therefore the tag check mostly differs to just using random replacement anyway. As for the direct-mapped victim cache, simply modding an entry's tag by 3 to choose its index in the victim cache is a random assignment in itself, and with only three cache lines in the victim cache, this implementation is not necessary. Another interesting observation stems from the fact that the MRU replacement policy and the victim cache with only one line have nearly identical results for each benchmark. This result led me to the conclusion that MRU replace is almost identical to having a one-line victim cache unless two memory accesses that aren't in the same cache line are highly-associative, in which case the MRU replace will perform slightly better. Perhaps the most interesting result (aside from the most effective one, LRU) is the performance of my L1$ write-back because it illustrates what the victim cache is intended to take most advantage of. In almost all cases, only a very small fraction of memory accesses happen to be highly associative, so the victim cache is intended to provide this associativity while keeping the fast access time of a direct-mapped L1 cache (Norman Jouppi). My version of the L1$ write-back policy's poor performance relative to the rest of the victim cache implementations is because when there is a victim cache hit, the associated (my mapping) entry is knocked out and replaced by the entry returned by the victim cache, and the association relation is now gone because only one of the memory locations remains in a level 1 cache. Because LRU replacement emerged as the victor, the following new AMAT and CPI calculations will be based on the assumption that this is the newly implemented policy. *Note: technically the (256:1:128) configuration with a one-line victim cache had the best results, but they are less interesting and less informative, so I'm sticking to the (512:1:64) configuration.*

Hit Time = **.41ns**, Miss Penalty = **100ns**, New Cycle Time: .60ns

Note: the cycle time is now set by non-memory instructions because cache accesses now take .41ns.

Bzip2

AMAT Without Victim Cache = .41 + (.01804 * 100) = **2.214ns**

AMAT With LRU Victim Cache = .41 + (.01723 * 100) = **2.133ns**

CPI Without Victim Cache = 1.2 + ((1898 + 2373587+956622) / 1096862410) * (100/.60) = **1.706**

CPI With LRU Victim Cache = 1.2 + ((1623 + 2248654 + 933270) / 1096862410) * (100/.60) = **1.684**

Sjeng

AMAT Without Victim Cache = .41 + (.02794 * 100) = **3.204ns**

AMAT With LRU Victim Cache = .41 + (.02661 * 100) = **3.071ns**

CPI Without Victim Cache = 1.2 + ((737316 + 657701 + 4319660) / 3728441696) * (100/.60) = **1.455**

CPI With LRU Victim Cache = 1.2 + ((626205 + 4283860 + 456093) / 3728441696) * (100/.60) = **1.440**

MCF

AMAT Without Victim Cache = .41 + (.21250 * 100) = **21.66ns**

AMAT With LRU Victim Cache = .41 + (.21122 * 100) = **21.532ns**

CPI Without Victim Cache = 1.2 + ((851 + 2742522 + 79291210) / 7972437872) * (100/.60) = **2.915**

CPI With LRU Victim Cache = 1.2 + ((625 + 2714533 + 78825753) / 7972437872) * (100/.60) = **2.905**

LBM

AMAT Without Victim Cache = .41 + (.00961 * 100) = **1.371ns**

AMAT With LRU Victim Cache = .41 + (.00947 * 100) = **1.357ns**

CPI Without Victim Cache = 1.2 + ((798 + 16669129 + 6529960) / 41953702120) * (100/.60) = **1.292**

CPI With LRU Victim Cache = 1.2 + ((737 + 16584281 + 6270124) / 41953702120) * (100/.60) = **1.290**

Overall, Bzip2 demonstrated the most significant improvement with an AMAT improvement of 3.66% and a CPI reduction of 1.3%.