Analysis of Sub-block Placement and Victim Caching Techniques

Umang Choudhary Pratik Phadke
Vasundhara Puttagunta Supreeth Udayashankar
{umang, pphadk1,vputta1,suadaya1}@umbc.edu

Abstract

Rapid advances in computer technology have led to the development of processors with peak performances of the order of GHz. Since it is not feasible to have unlimited fast-memory, the performance of these processors is handicapped if the performance of the memory-hierarchy is poor. Caching techniques have been developed with this in mind. This paper presents the analysis of the performance of two such techniques.

Sub-block placement: This technique reduces the miss penalty by reducing the bandwidth between the cache and it's next level. Our results show that sub-block placement enhances the performance both for L1 and L2, more significantly in the former. The performance improves with the increase in the number of sub-blocks.

Victim caching: This technique reduces the miss rate by adding a small, fully associative cache between a cache and the next level in the memory hierarchy. The results show that victim caches reduce the miss rate in L1 caches, but the reduction achieved depends on the structure and configuration of the cache and it's victim cache. Our study of the performance of victim caches as the block size, cache size and associativity of the caches was varied showed that there can be a significant improvement in performance. However, as cache sizes increase or associativity becomes higher, victim caches do not greatly enhance performance.

Keywords: memory hierarchy, primary cache, secondary cache, associativity, victim caching, sub-block placement, miss rate, miss penalty.

---

1. Introduction

The ideal memory would be large enough to hold all code and data for even the worst-case program and also fast enough so all memory references are satisfied without stalling the CPU. The improvement in memory access times has not kept up with the improvement in CPU performances. Large memories are slow so most computers rely on the principle of locality and use a memory-hierarchy for performance close to the ideal memory. In such a design, the farthest from the CPU is a very large memory (a disk) to satisfy the size requirement. Closest to the CPU is a very fast S-RAM memory in which recently referenced memory items and their neighbors are cached: the principle of locality says that these items are most apt to be referenced in the near future. There has been an increasing need for faster memory over the years. The effectiveness of a memory-hierarchy is a function of

i. ***Miss rate:*** The fraction of references that are missed in the cache.
ii. ***Miss penalty:*** The additional time required to service such a miss.
iii. ***Instruction count:*** This is program and instruction set architecture dependent and memory-hierarchy cannot affect it.
iv. ***Memory references per instruction:*** This is CPU instruction set architecture dependent. Memory hierarchy design does not alter this factor.
v. ***Hit time:*** This is the time taken for the CPU to get the contents of it's request when it's a hit in the cache.

The equation is given by
(Memory stall cycles) = (Instruction Count) * (Memory References per Instruction) * (Miss Rate) * (Miss Penalty)

Another equation that is relevant here is the equation for average memory access time.

Average memory access time = Hit time + MissRate * MissPenalty.

The design of a cache [7,12] involves many choices, such as whether to have instructions and data in separate caches or both in one cache, the size of the cache and the number of bytes in each cache block. Other

considerations are whether the cache is fully-associative, direct-mapped, or N-way set-associative for (N = 2, 4, 8, etc) and the replacement policy. The design choices are very important as they determine the final performance of the cache.

2. Previous Work

There have been several approaches proposed to attack the different factors affecting memory hierarchy performance. [13] provides an excellent discussion on each of these techniques.
Techniques focussed on reducing miss rates include:

i. Larger Block size
ii. Higher Associativity
iii. Victim Caches[1]
iv. Pseudo-associative Caches
v. Hardware pre-fetching of instructions and data
vi. Compiler controlled pre-fetching
vii. Compiler Optimizations

Techniques focussed on reducing miss penalty include:

i. Giving priority to read-misses over write
ii. Sub-block Placement
iii. Early restart and critical word first
iv. Non-blocking caches to reduce stalls
v. Multi-level caches[2,3,6,9,10]

Techniques focussed on reducing hit time include:

i. Small and simple caches
ii. Avoiding address translation during Indexing of the Cache
iii. Pipelining Writes for Fast Write Hits [1]

We attempted to analyze the performance improvement that is achieved by using two of these techniques namely sub-block placement and victim caching.

Sub-block placement*:* This technique helps to reduce miss penalty. The reduction in miss penalty is achieved by splitting up blocks into smaller units of transfer called "sub-blocks". Suppose you try to create a cache that must fit on a single chip, you may find that your tags are too large, either because they don't fit on the chip or because they are too slow. The cache tags are address tags that are put on each cache block and indicate block address. All tags searched in serial to check if they contain valid bit. This will indicate whether the entry is a valid address A simple solution is to go to large blocks, which reduces tag storage without decreasing the amount of information you can store in the cache. The miss rate will likely improve, but the increase in miss penalty could make the larger blocks a bad decision. The solution to this dilemma is called sub-block placement. In this technique,

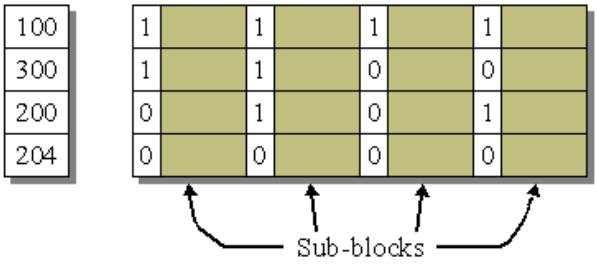

Figure 1: A Sub-block Cache

a valid bit is added to units smaller than the full block, called sub-blocks. Only a single sub-block need be read on a miss. The valid bits specify some parts of the block as valid and some parts as invalid, so a match of the tag doesn't mean the word is necessarily in the cache, as the valid bit for that word must also be on. Thus, sub-blocks have a smaller miss penalty than full blocks. Figure 1 shows the reduction in tag storage If the valid bits had to be replaced by full tags, there would be much more memory dedicated to tags, which is the reason sub-block placement was invented.

Victim caching: It is a technique used to reduce the miss-rate without affecting clock cycle time or miss-penalty. This is done by inserting a small (less than 16 entries), fully associative cache in the refill path of a cache. It is meant to capture and store the cast-outs (victims) from the cache to which it is appended.
Victim caching is based on the principle of temporal locality: recently accessed items tend to be accessed in the near future. By keeping items cast out from a cache,

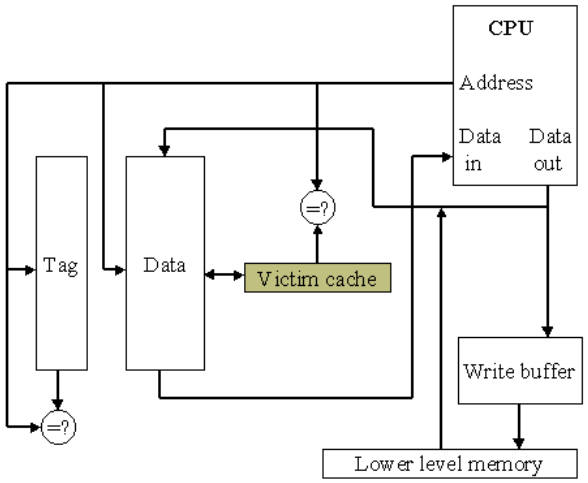

Figure 2: A Victim Cache

victim caches can improve performance by providing the items when they are needed again instead of their having to be accessed from the next level of memory hierarchy. If there is a hit in the victim cache, the entries are swapped between the cache and it's victim cache. Figure 2 shows the construction of a victim cache.

This idea of a victim cache was first introduced in [1] and their results show that there can be a 20% to 95% reduction in misses for a 4KB direct mapped data cache. However this is heavily program dependent.

3. Design

We attempt to analyze the performance of *Sub-block Placement* and *Victim Caching* on the performance of L1 and L2 caches and present the experiments and results.

```
if ( !L1_Cache.Read(addr) )
// Address was not found in primary cache
  if (Victim_Cache.Read(addr) )
  // Address was found in victim cache
  {
  swap(L1_Cache.Item(addr),Victim_Cache.Item
  (addr)) ;
   // Swaps the items between L1 and the victim
   }
    else if (L2_Cache.Read(addr) )
    // Found in L2
       {
       victim_addr ← L1_Cache.Load(addr);
        // Move the found item into L1 from L2
        // victim_addr has the castout from L1
        victim_addr ← Victim_Cache.Load
        (victim_addr) ;
         // Victim cache is loaded with this castout
         // victim_addr has the castout from the victim
        cache
        L2_Cache.Load(victim_addr);
         // Store the castout from the victim cache in L2
         }
    else {
           Invalidates ← L2_Cache.Load(addr);
           // Load into L2
           // Invalidates has the items removed from L2
           L1_Cache.Invalidate(Invalidates);
           // Invalidate the corresponding items in L1 if
           //they exist there
           Victim_Cache.Invalidate(Invalidates);
           // Invalidate the corresponding items in Victim
           // if they exist there
           victim_addr ← L1_Cache.Load(addr);
           // Move the item into L1
           // victim_addr has the castout from L1
           victim_addr ← Victim_Cache.Load
                                    (victim_addr) ;
           // Victim cache is loaded with this castout
           // victim_addr has the castout from the victim
           // cache
          L2_Cache.Load(victim_addr);
           //Store the castout from the victim cache in L2
    }
```

Figure 3 : Pseudocode for Victim caching for a unified L1 with a unified Victim cache.

We analyze the effects of varying the cache size, block size and associativity of the cache. The cache simulations were done using a cache simulator called the "Acme Cache Simulator" (ACS). The simulator is written in C++ as a single object, thus making it usable as part of a larger simulator. The simulator uses the LRU replacement algorithm, i.e. blocks that have not been used for longest time are replaced first. The Acme Cache simulator was modified to simulate memory configuration with victim caches as well as sub-blocks.

By specifying cache size, block size, associativity, victim cache configuration, number of sub-blocks and whether it is split or unified we can configure the memory system. Our present implementation allows victim caches for L1. It can be easily extended for L2 also, though we do not expect a significant improvement from this. Figure 3 lists the pseudo-code that explains the working of a victim cache simulator in case of a read in a unified L1 with a unified victim. Likewise, a write miss in L1 is dealt with similarly. The logic is not very different in case either the L1 cache or the victim cache or both are split.

```
if ( !Cache.Read(address) ) {
// address not found in the cache, it means that either
// the entire block was not in the cache or the
// required sub-block was not valid.
if ( !blockFound)
{
// entire block not in cache, get it from memory
// find a block to move out, make space for the block
// that has to be moved in
victim = Cache.FindVictimBlock();
// move the victim out of the cache and the required
// block in
Cache.LoadBloack(victim, address);
}
```

Figure 4: Pseudocode for Sub-block

4. Experimental setup

We used the address traces of instructions and data for the SPEC92 benchmarks shown in Figure 5 for testing the performances of the memory configurations. They contain a mix of both integer and floating point benchmarks. [13] rightly points out that for realistic evaluation of the cache performances we need to have simulations on several millions of references. Some benchmarks contain almost a hundred million references.

We tested the sub-block placement and victim caching techniques for the following cases:

1. For a system with only L1 cache:
   i. Varying associativity: direct mapped , 2, 4 and 8 way set associative.

ii. Varying cache size : 2KB, 4KB, 8KB, 16KB, 32KB, 64KB

2. For a system with both L1 and L2 caches
   a. Unified instruction and data caches for both L1 and L2. L1 has a fixed size of 32 KB
   b. Split instruction and data caches for L1 and unified cache for L2. L1 data and instruction cache sizes are fixed at 16KB each.

   For the above systems we had the following configurations with:
   i. varying associativity for L1: direct mapped, 2, 4 and 8 way set associative
   ii. varying associativity for L2: direct mapped, 4 and 8 way set associative
   iii. varying L2_CacheSize: 128KB, 512KB, 1MB, 2MB

For the Sub-block case, we considered blocks to comprise of 2, 4 or 8 sub-blocks.

For the victim cache scheme we had 1, 2, 4, 16 entries in the victim cache. Victim caches were used only in association with the L1 cache. We also had a special configuration where only the L1 data cache had a victim cache.

Plat-form	**Program name**	**Bench-mark**	**No. of mem. ref. (10^6)**	**No. of inst. ref. (10^6)**	**No. of data ref. (10^6)**
Sparc	Espresso	SPEC int92	10	8.2	1.8
	Gcc	SPEC int92	100	78.8	21.2
	Tomcatv	SPEC fp92	10	7.4	2.6
R2000	Comp	SPEC int92	10.5	8	2.5
	Hydro	SPEC fp92	11	8.2	2.8
	Nasa7	SPEC fp92	99.7	65	24.7
R3000	Espresso	SPEC int92	1	0.809	0.191
	Tomcatv	SPEC fp92	1	0.615	0.385
	Nasa7	SPEC fp92	1	0.802	0.198

Figure 5: SPEC92 benchmarks used in the experiments

We also had a cache configuration where the L1 cache is split and only the data cache of L1 has a victim cache.

5. Results

Sub-block Placement:
For the L1 cache -
From figures 14 and 15, we observe that there is a notable decrease in the bytes transferred as we move from a system without sub-blocks to one with sub-blocks. As we increase the number of sub-blocks, lesser bytes are transferred in and out of the cache.

From Figure 16, we can see that increasing the associativity of a cache decreases the miss rate (conflict misses are decreased). The effect on miss rate is most significant when changing a direct-mapped cache to a 2-way set-associative cache: changing to a 8-way set-associative cache has very little effect. Doubling the associativity doubles the number of tags to be searched so hit time may be worse. A direct-mapped cache has the advantage that search time can be overlapped with data fetching.

For the L2 cache-
Figures 17, 18 and 19 show that the improvement in performance due to sub-blocks follows a similar pattern as that for L1. However, the gain in performance as the number of sub-blocks increase is not as remarkable as that of L1. This is because the bytes transferred from an L2 cache is essentially a low value and therefore the performance gain is limited.

Victim Caching:
Figures 6, 7 and 8 show that as the block size increases, the reduction in misses due to the victim also increases and also as the number of entries in the victim cache are increased. The effect is more dominant for a split L1 than a unified victim. This is because split L1 is known to have lower miss rate than a unified one and as a result the victim cache has to deal with lower number of references.

There is not much improvement as we move from a unified victim cache to a split victim cache for a split L1 (figures 7 and 8).

Figures 9, 10 and 11 show the performance of the victim caches as the L1 cache sizes increase. We find that as the L1 cache sizes increase, the effect of the victim cache becomes less striking. The fact that L1 cache is large implies that it is capable of having more hits leaving the victim cache useless in most cases.

From figures 12 and 13 we realize that as the associativity of L1 increases, the victim cache becomes less effective.

The results of the experiment of having the victim cache only for the L1 data cache for the nasa7 benchmark are plotted in figure 19. We can conclude from these graphs that having a victim cache for just the L1 data cache is justified.

6. Conclusion

We presented several results on the performance of various cache configurations with sub-block placement and victim caching.

In systems where bandwidth plays a critical role in the performance of the system (like Distributed Shared Memories) sub-blocking becomes essential.
Our results show that,

i. Bandwidth requirements are cut down by approximately 10 to 15 percent for caches with low associativity.
ii. It improves performance for both L1 and L2 caches.
iii. As associativity increases the effectiveness of sub-block placement diminishes.

In addition to reducing bandwidth sub-block placement can also be used to reduce false sharing.

We made the following observations from our results from the simulations of caches with victim caching:

i. Victim caches with 4 to 16 entries for primary caches can reduce the miss rate by about 15 to 25 percent. In particular, small primary caches with low associativities are benefited the most.
ii. As the cache size increases, the effectiveness of victim cache declines.
iii. It is more beneficial to have a victim cache for a data cache rather than an instruction cache.
iv. We did not observe as much improvement as recorded in [1]. This is because we tested the performance as an average of the benchmarks mentioned in section 4 and not for a specific program.

As the CPUs clocks become shorter, processes are bounded by the memory access rate and thus the need for faster cache hit times becomes more prominent. This encourages small and simple cache (direct mapped) design for primary caches. Victim caches can be made best use of in this scenario.

It would be interesting to examine the performance of caches that make use of both these techniques to reduce both the miss penalty and miss rate.

Acknowledgements

Parallel Architecture Research Laboratory (PARL), Klipsch School of Electrical and Computer Engineering at New Mexico State University, support TraceBase a trace database from where we obtained the Acme-cache simulator and the traces for Spec92 benchmarks. <http:\\tracebase.nmsu.edu>

References

[1] Jouppi, N. P. *Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers.* Proceedings of 17th Annual Int'l Symposium on Computer Architecture, 1990, pp. 364-373.

[2] Przybylski, M. Horowitz, and J. Hennessy. *Characteristics of performance-optimal multi-level cache hierarchies*. Proceedings of the 16th International Symposium on Computer Architecture, 1989, pp. 114 –121.

[3] Baer, Jean-Loup, and Wang, Wen-Hann. *On the inclusion properties for multi-level cache hierarchies*. 25 years of the international Symposia on Computer Architecture (selected papers), 1998, pp. 345 – 352

[4] Gee, et al.. *Cache Performance of the SPEC92 Benchmark Suite.* IEEE Micro, Vol. 13, Number 4, August, 1993, pp. 17 – 27. http://www.cs.wisc.edu/~markhill/spec92miss.html

[5] Johnson, Teresa L., and Hwu, Wen-mei W. *Run-time Adaptive Cache Hierarchy management via Reference Analysis.* ISCA '97, Denver, CO, USA, pp. 315 – 326.

[6] Jouppi, Norman P, and Wilton, Steven J. E. *Tradeoffs in Two-Level On-Chip Caching.* Research Report 93/3, October 1993, Compaq Western Research Laboratory http://www.research.digital.com/wrl/publications/abstracts/93.3.html

[7] Steven Przybylski, Mark Horowitz, John L. Hennessy. *Performance Tradeoffs in Cache Design.* ISCA 1988: Honolulu, Hawaii, USA, pp. 290 – 298.

[8] Cheriton, D. R., Goosen, H. A. and Boyle, P. D. *Multi-level shared caching techniques for scalability in VMP-M/C.* Proceedings of the 16th annual International Symposium on Computer Architecture, 1989, pp. 16–24.

[9] Short, Robert L., Levy, Henry M. *A Simulation*

Study of Two-Level Caches. Proceedings of the 15th Annual Symposium on Computer Architecture. May 1988, pp. 81–88.

[10] Wan, Marlene, and George, Varghese. *Effect of Second Level Cache Parameterising Overall Cache Performance.* http://infopad.eecs.berkeley.edu/~varg/CS252_report/Final.html

[11] Hill. *A case for Direct Mapped Caches*. Computer, 21:12. 1988, pp. 25-40.

[12] Smith, A. J. *Cache Memories*. Computing Surveys, 14:3. September, 1982, pp. 473-530.

[13] Hennessy, J.L., Patterson, D.A. *Computer Architecture A Quantitative Approach*, 2nd edition, 1996. Morgan Kaufmann Publishers, Inc.

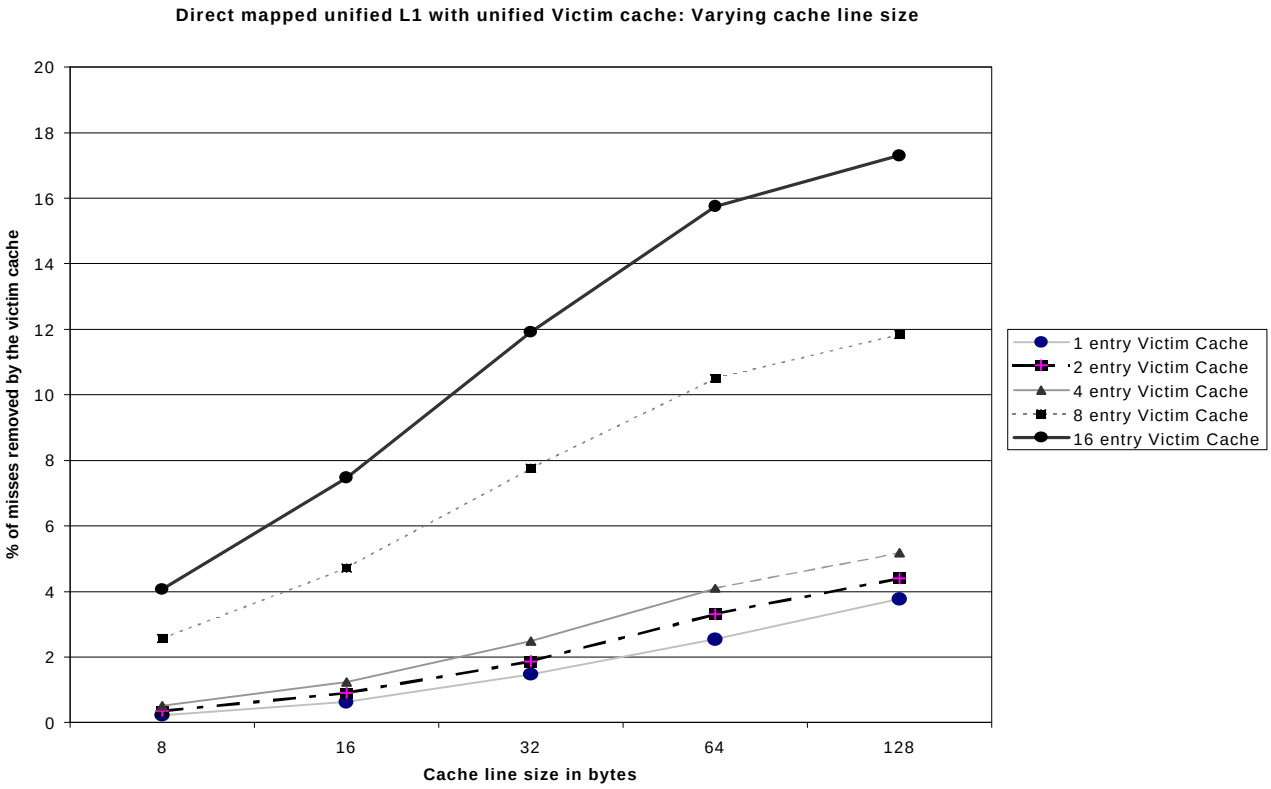

Figure 6: A plot of the % of misses removed by a unified victim for a unified L1 cache as the cache line size (i.e. block size) is varied. (L1 cache size = 32KB)

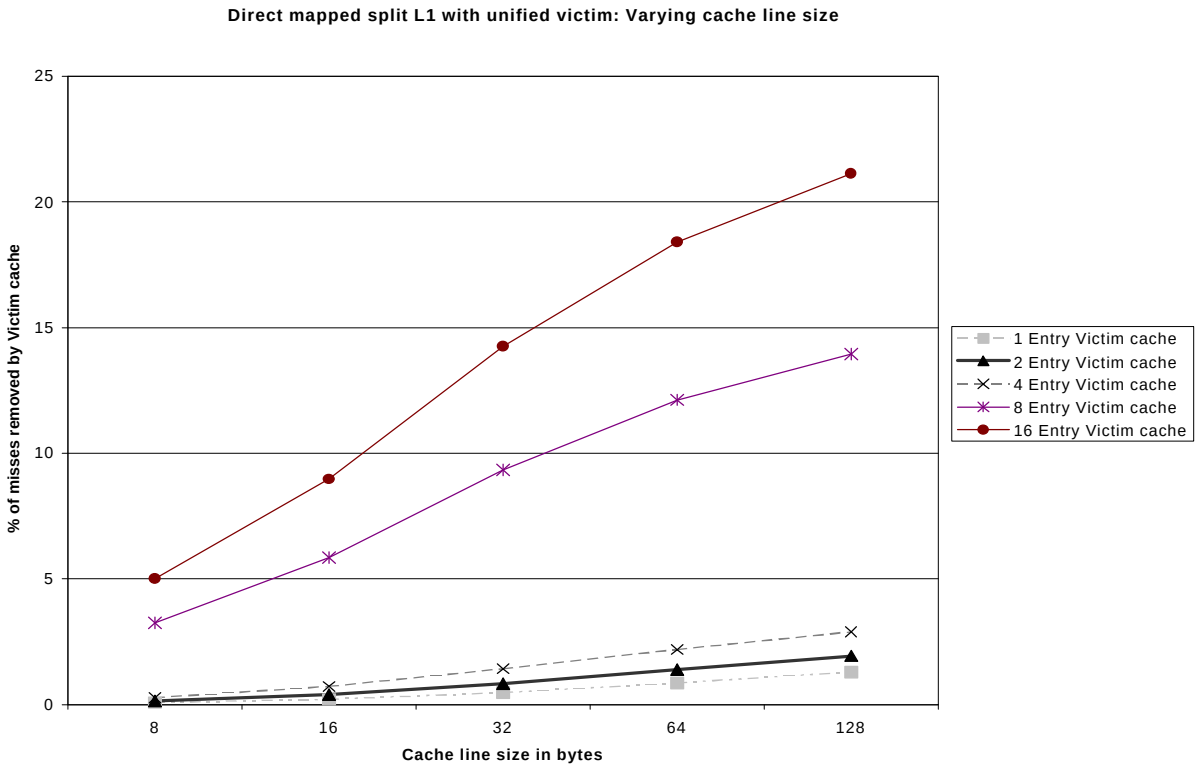

Figure 7: A plot of the % of misses removed by a unified victim for a split L1 cache as the cache line size (i.e. block size) is varied. (L1 data & instruction cache size = 16KB each)

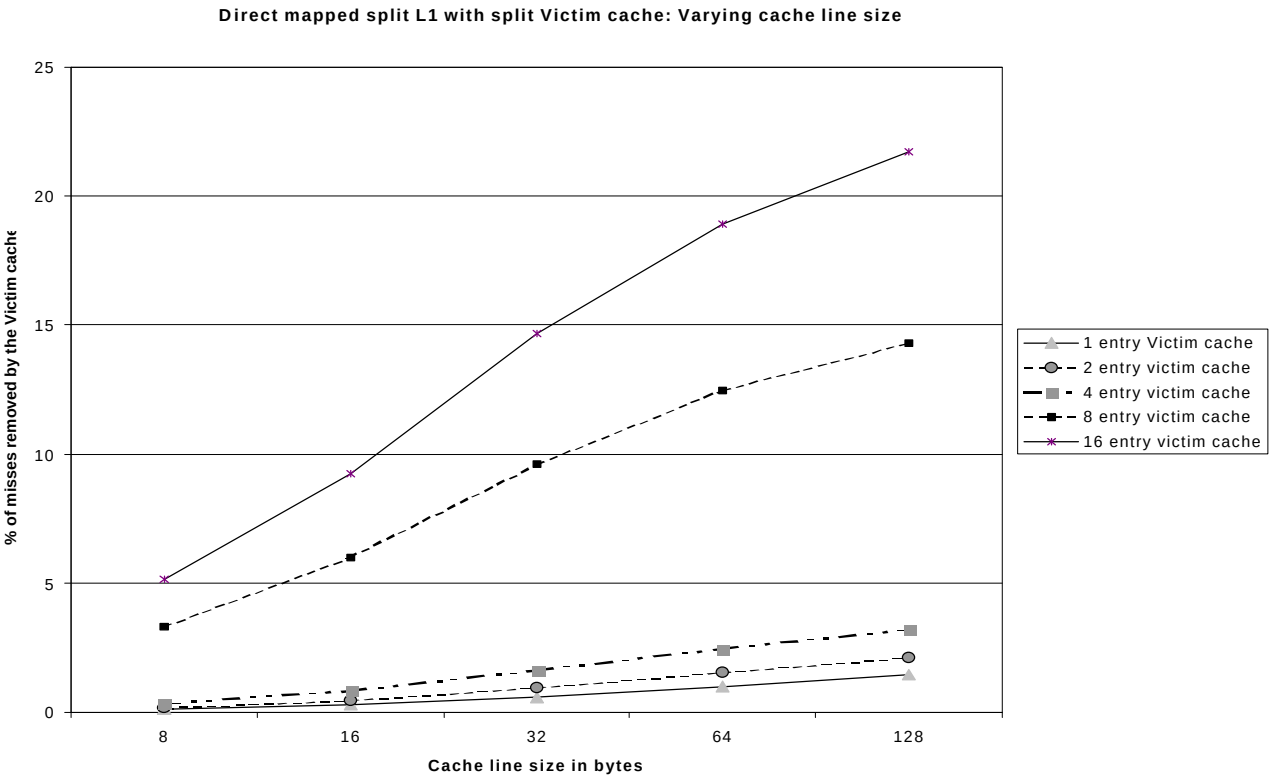

Figure 8: A plot of the % of misses removed by a split victim for a split L1 cache as the cache line size (i.e. block size) is varied. (L1 data & instruction cache size = 16KB each)

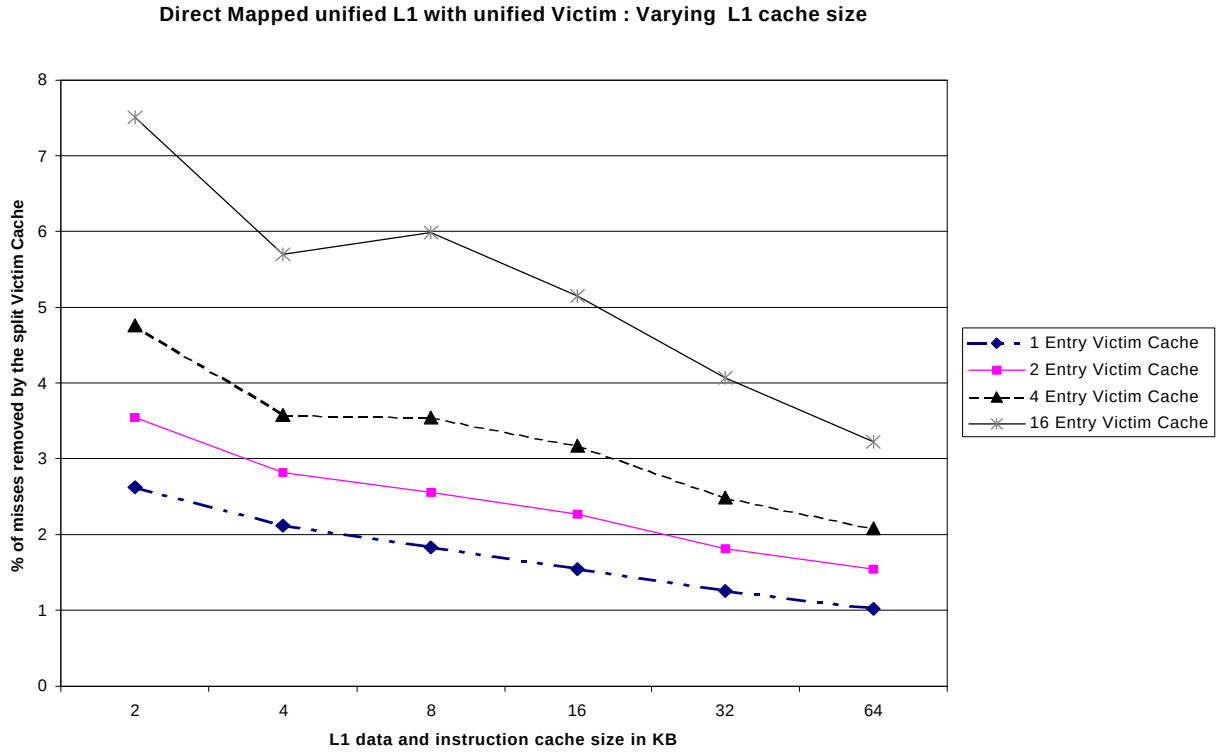

Figure 9: A plot of the % of misses removed by a unified victim for a unified L1 cache as the cache size is increased. (Block size = 32 bytes)

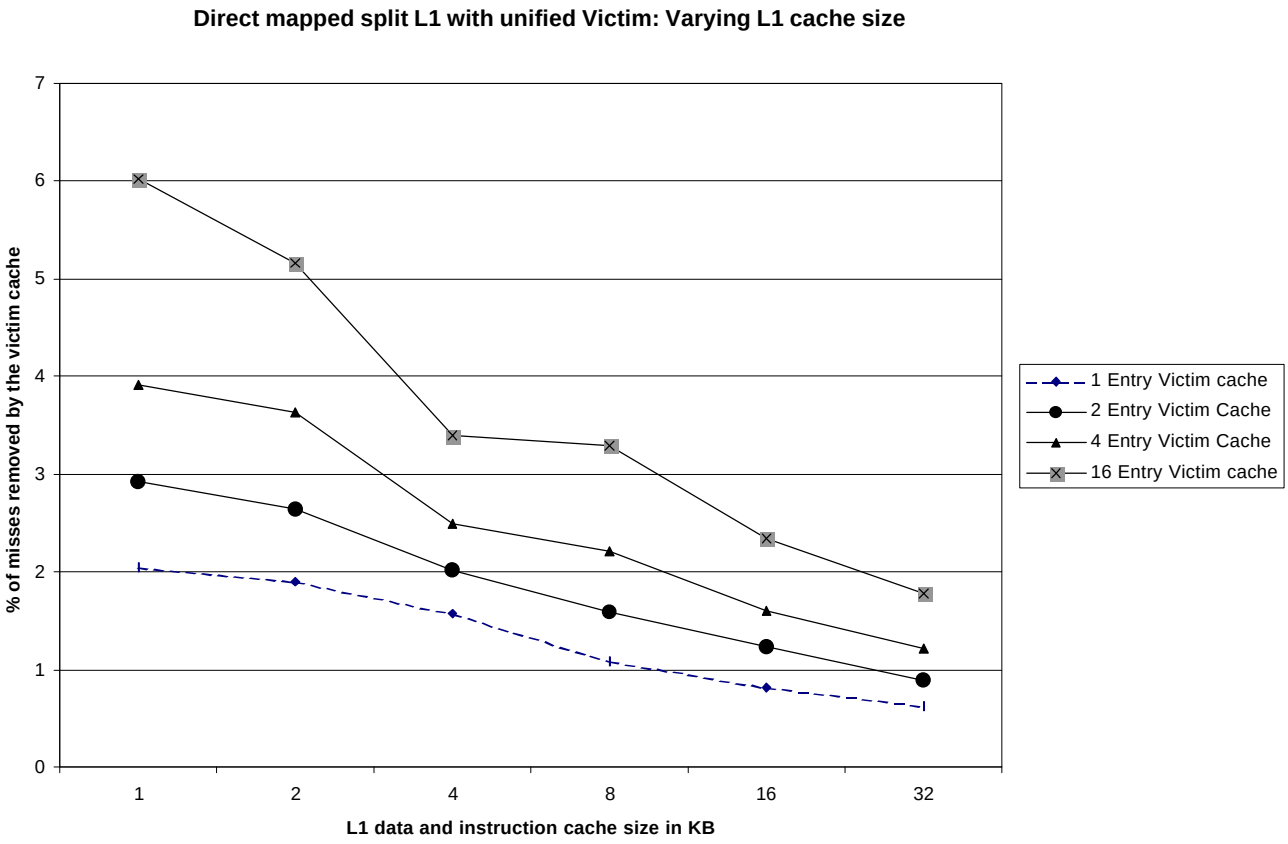

Figure 10: A plot of the % of misses removed by a unified victim for a split L1 cache with equal data and instruction caches as the size is increased. (Block size = 32 bytes)

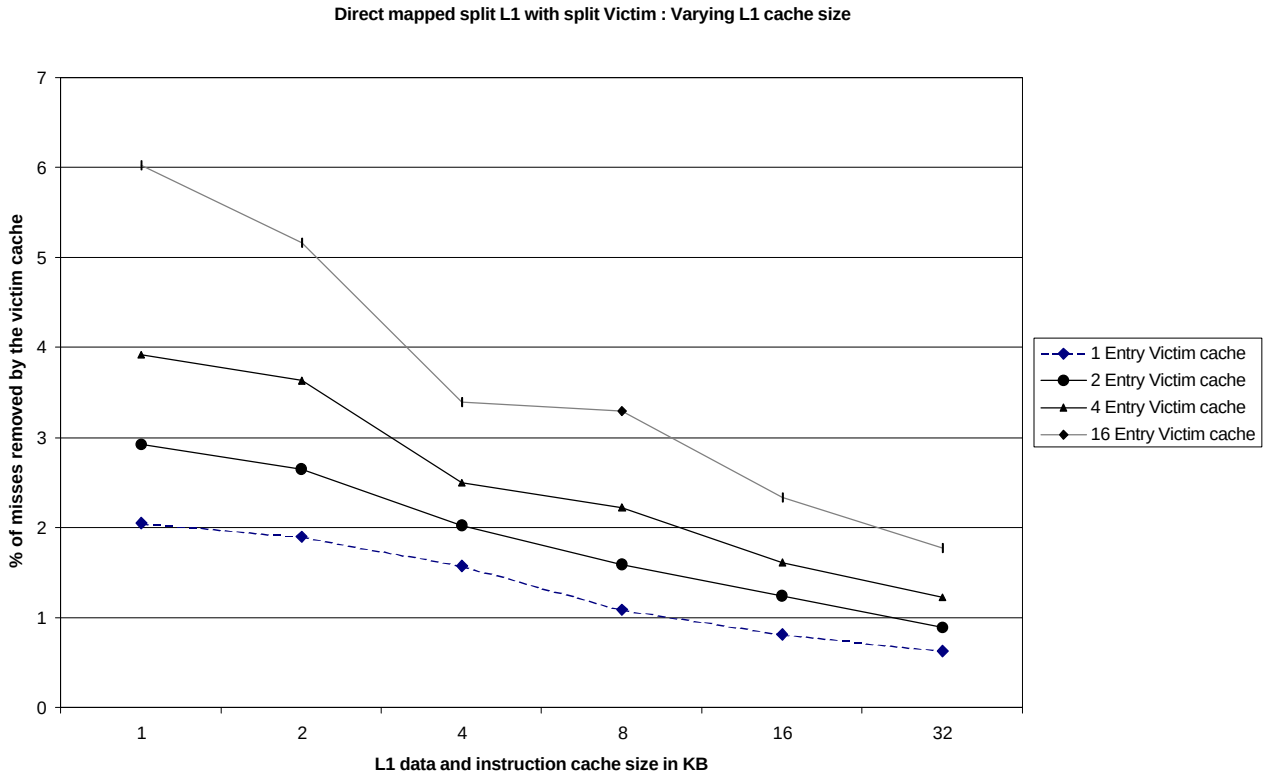

Figure 11: A plot of the % of misses removed by a split victim for a split **L1 cache as the cache size is varied. (block size = 32 bytes)**

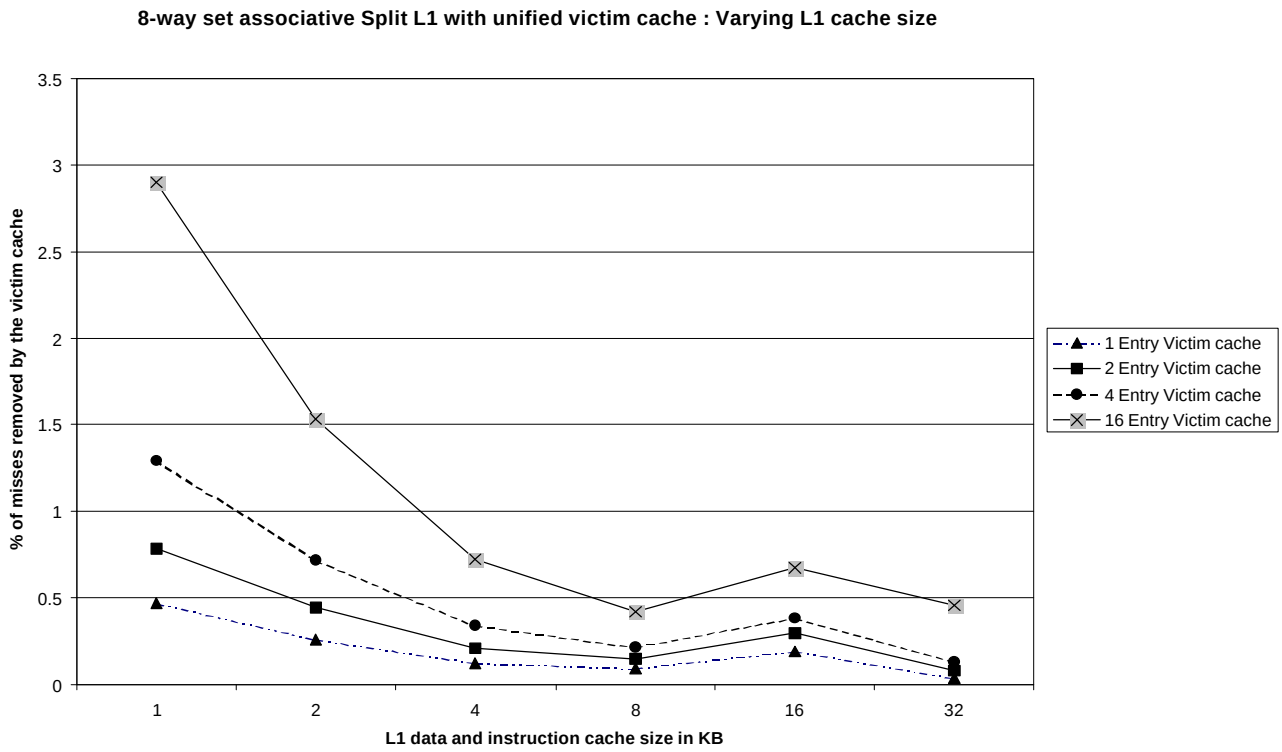

Figure 12**: A plot of the % of misses removed by a unified victim for a split L1 cache as the cache size is varied. (**block size = 32 bytes and 8-way set associative **)**

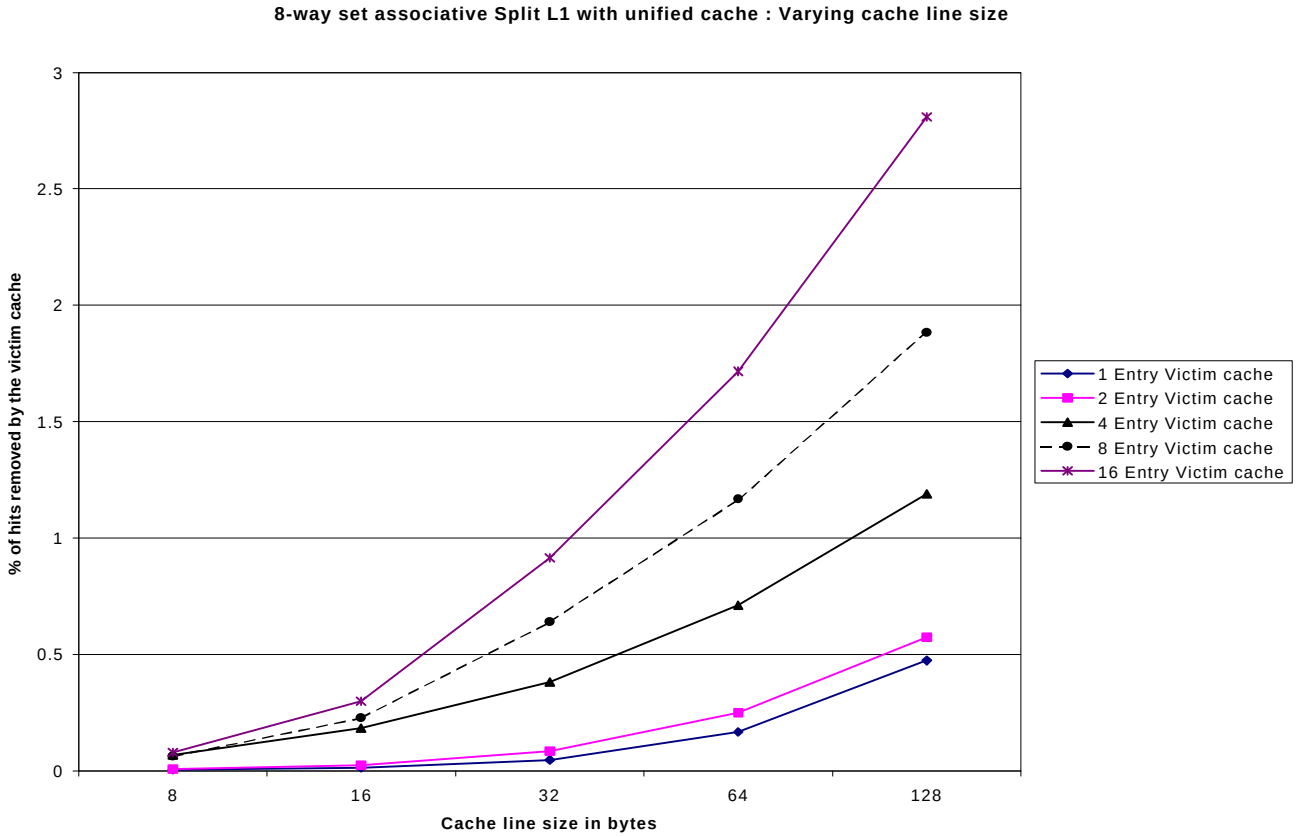

Figure 13**: A plot of the % of misses removed by a** unified **victim for a** split **L1 cache as the cache line size (ie block size) is varied. (L1 cache size** = 32 KB and 8-way set associative **)**

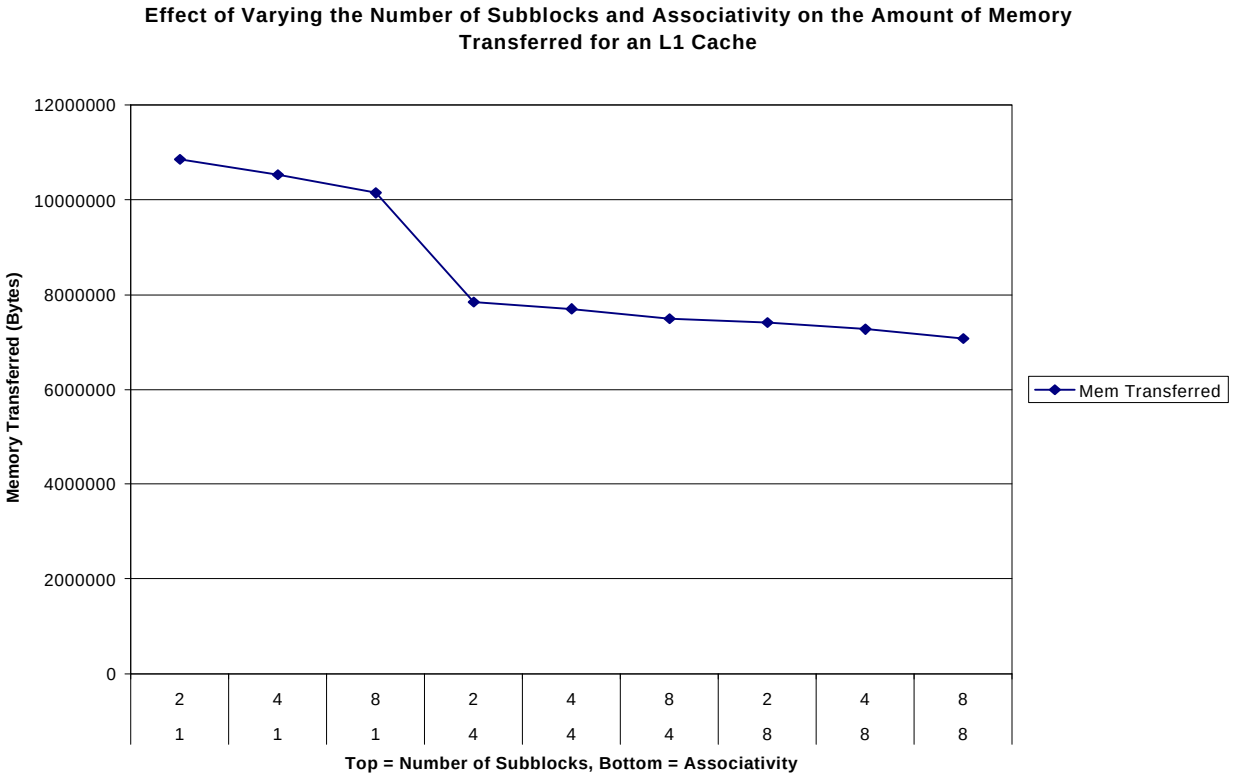

Figure 14

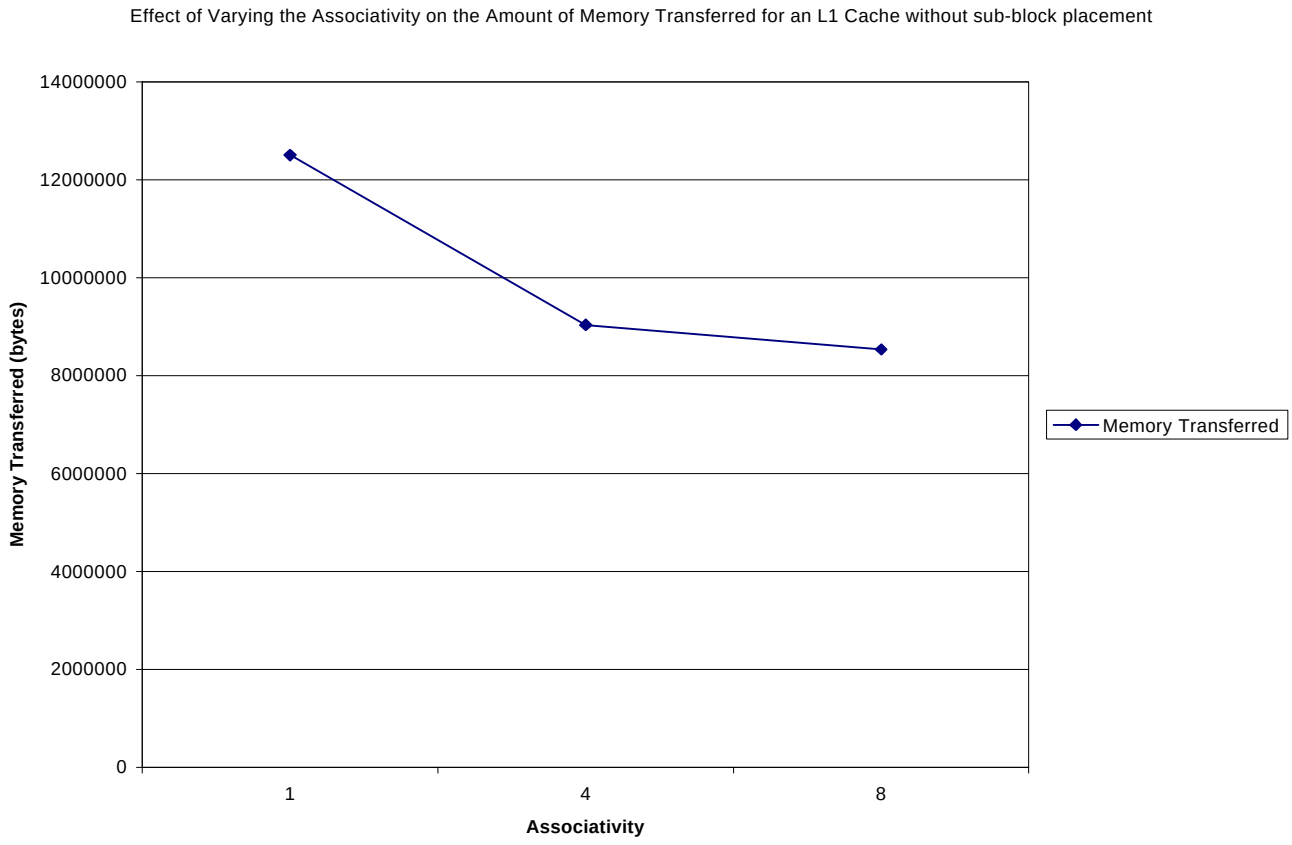

Figure 15

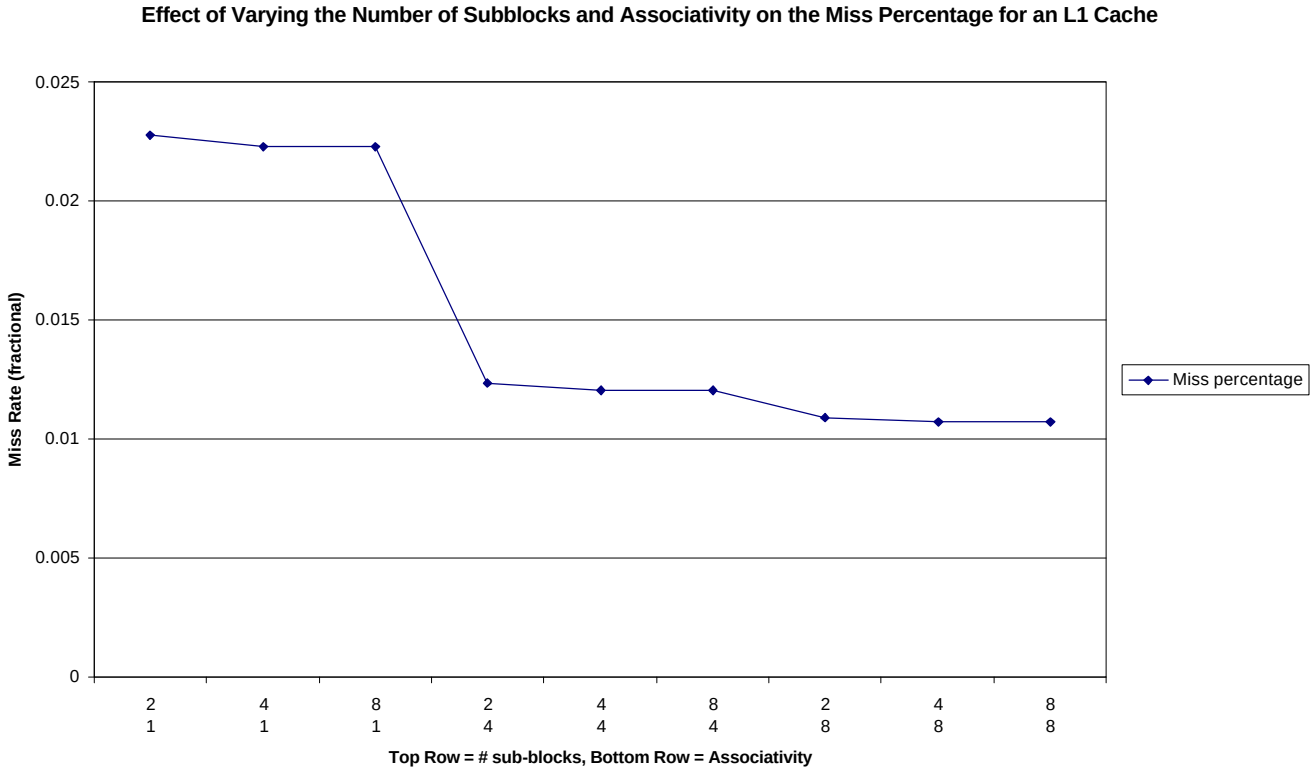

Figure 16

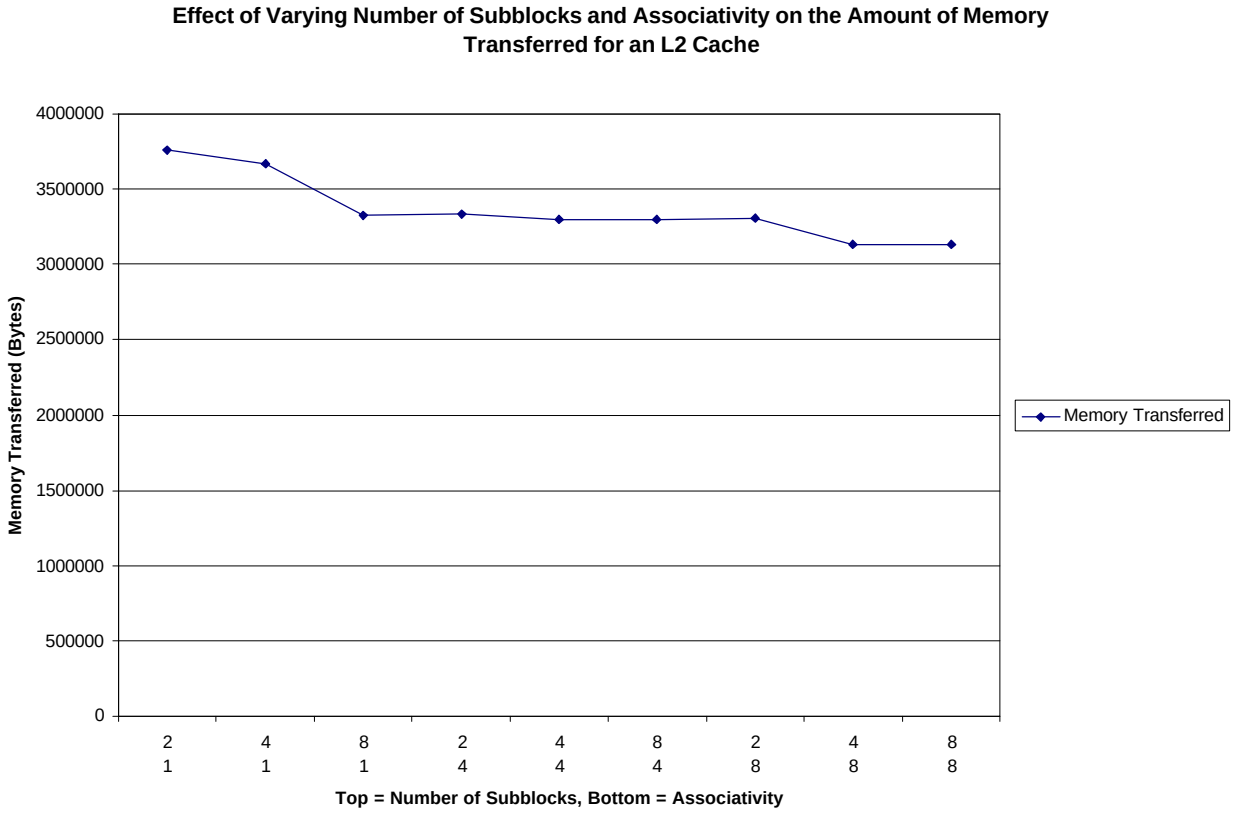

Figure 17

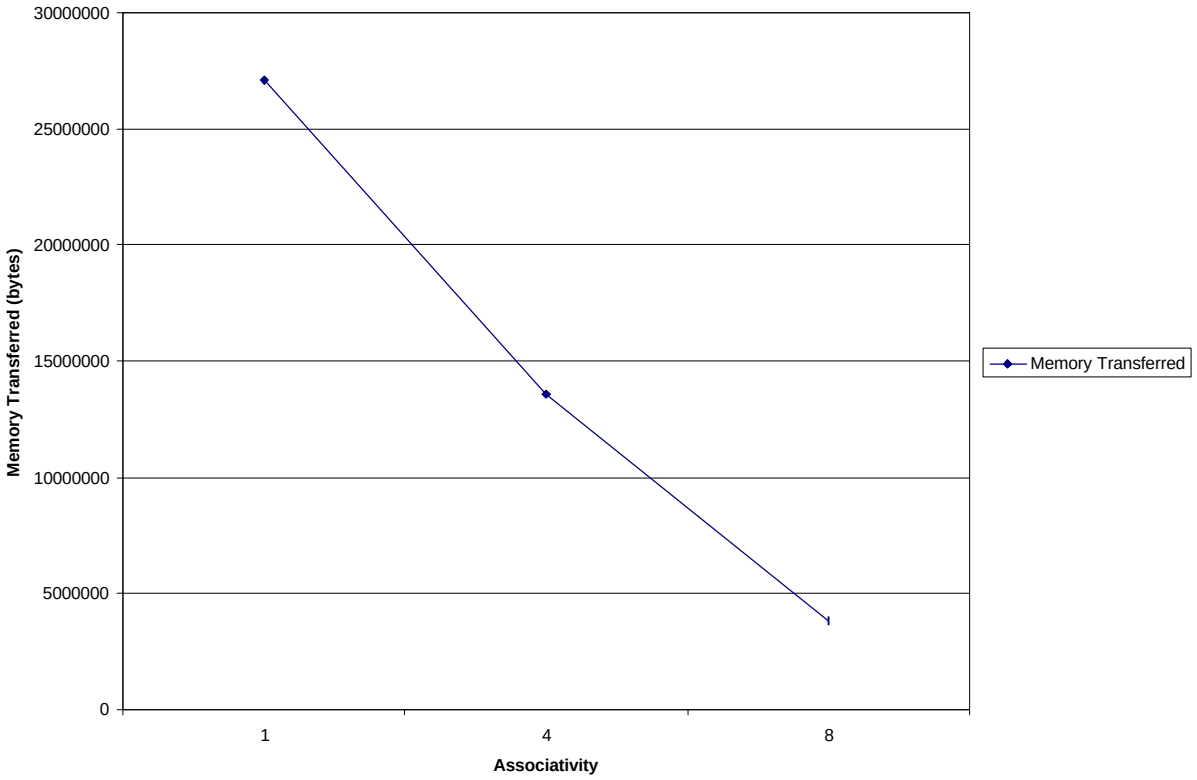

Figure 18

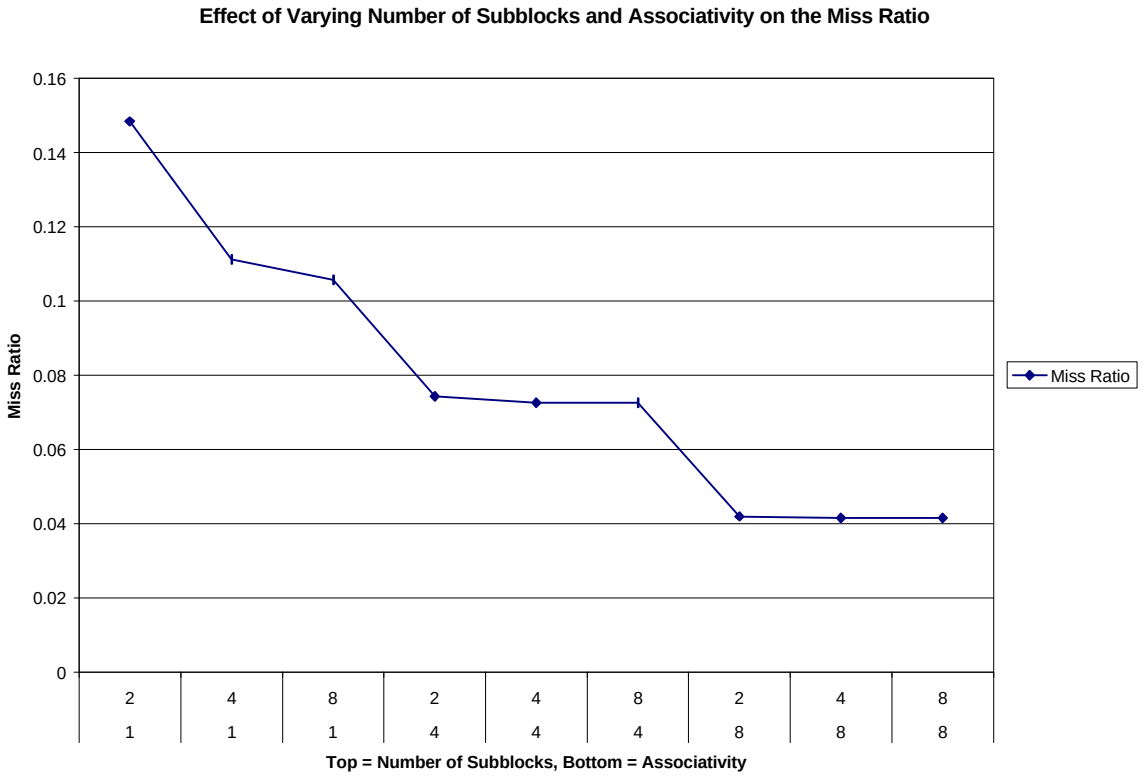

Figure 18

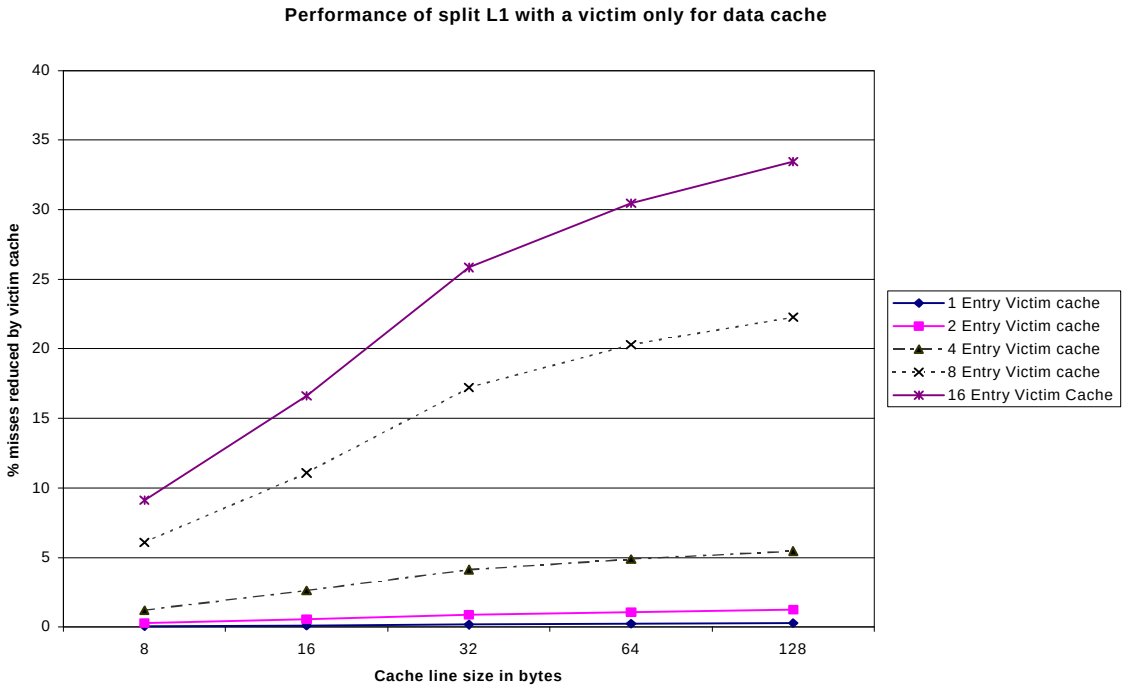
Performance of split L1 with a victim only for data cache
% misses reduced by victim cache
40
35
30
25
20
15
10
5
0
8
16
32
64
128
Cache line size in bytes
1 Entry Victim cache
2 Entry Victim cache
4 Entry Victim cache
8 Entry Victim cache
16 Entry Victim Cache

Figure 19