ANALYSIS OF CACHE ARCHITECTURES

Nikolay Pavlovich Laptev
Department of Computer Science – University of California Santa Barbara
Santa Barbara, CA, 93030
nlaptev@cs.ucsb.edu

ABSTRACT
The growing processor performance is hampered by the ever growing penalties caused by cache misses. Furthermore the gap between CPU and memory continues to widen causing novel processors to not perform at their full potential. There have been various solutions proposed to the caching problem which include prefetching, victim and miss caches as well as stream buffers. The right combination of these technologies still remains uncertain. This paper will examine different caching techniques as well as different variations of different sizes in cache and analyze performance effects on a mixture of benchmarks. We will also examine how code structure affects performance and cache hit/miss ratios. We also examine Athlon-specific prefetch instructions.

1 INTRODUCTION

The gap between CPU and memory performance continues to grow and this continuing trend poses a serious problem for future processor development and performance. Due to cache misses the processor has to stall thus losing valuable CPU time and degrading performance. This is true especially for scientific programs where cache access is vast which thereby causes high miss rates for all cache hierarchies. Several solutions have been proposed to this problem. One of the more robust ones includes software prefetches where we speculate on future memory accesses and prefetch needed data to cache. Prefetching must be accurate and timely. If we prefetch too soon, then our prefetch will be of no use (prefetched data will get replaced before we have an opportunity to use it) and it must be accurate in a sense that we must prefetch data that we will indeed use. Another way to decrease the amount of cache misses in a program is to acknowledge memory layout of several programming languages. The common case, C, follows a row major memory layout where we access rows first. Fig 1 demonstrates how easy it is to significantly reduce miss rates by simple loop interchange. Note that in the first loop is much faster than the second because in the second, the array does not fit in memory [since we are performing a column-major access] This simple operation reduces L1d miss from 19.9% to 1.2%.

```
// fast.c
int main(void)
{
  int h, i, j, a[1024][1024];
for (h = 0; h < 10; h++)
     for (i = 0; i < 1024; i++)
        for (j = 0; j < 1024; j++)
           a[i][j] = 0;     // !!
             return 0;
   }
```

```
// slow.c
 int main(void)
{
    int h, i, j, a[1024][1024];
      for (h = 0; h < 10; h++)
      for (i = 0; i < 1024; i++)
        for (j = 0; j < 1024; j++)
        a[j][i] = 0;     // !!
          return 0;
   }
```

Fig1. *C follows row-major order, thus arranging loops in that fashion will cause a decrease in both L1 and L2 misses on the order of 17-fold Credit: CacheGrind documentation*

In order to profile such subtle affects on loop interchange we need sophisticated tools. In this paper we use Cache Grind which is part of Valgrind available freely on the net. Cache Grind is a cache simulator which simulates the cache hierarchy during execution of a program. It also has the ability to dump statistics into a separate file. It is important to note that Cache Grind simulates inclusive L2 cache which replicates what is in L1. This can cause a greater amount of misses reported by Cache Grind. The downside of Cache Grind is that it only considers a program as a sole process running on the machine, in other words it ignores any number of background processes that are currently running on the system. It also ignores misses from the TLB and it has no prefetching, nevertheless with the above mentioned shortcomings Cache Grind still captures over 98% of cache behaviour based on hardware measurements [1].

In this paper we also make distinctions between L1i L1d and L2 misses.

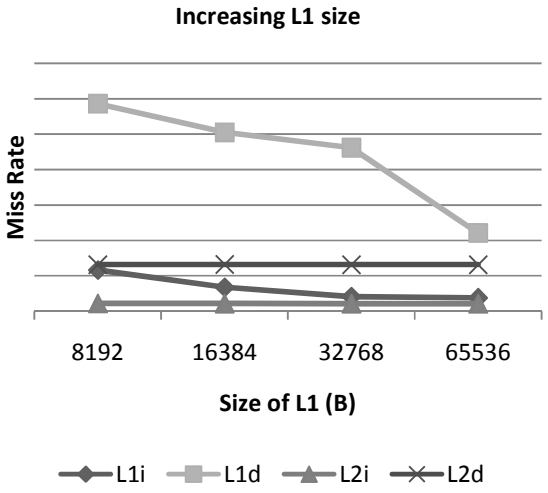

Fig 2: Benefits from increase in size of L1. (Bioinformatics benchmark)

As indicated in Fig 2 improving L1d misses provides the most performance benefit. L1i posses a sequential flow thus miss rates from them is minimal.

In Fig 3 we present results capturing the miss rate derived after increasing the L2 Cache.

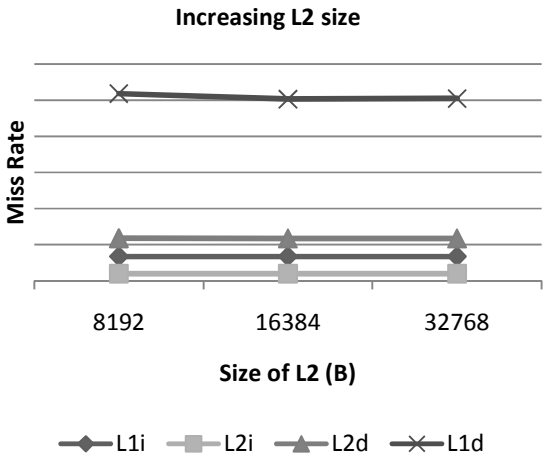

Fig3: Virtually no benefit if we increase the size of L2 (Bioinformatics benchmark)

Again, in the figure above we see that modifying L1 gives us the most benefit, and thus we will focus our attention on it in this paper.

Before we continue, it is also important to characterize four types of cache misses that hurt performance. Conflict, compulsory, capacity and coherence. Conflict misses are misses that would not have occurred if the cache was fully associative and had LRU replacement. Compulsory misses are misses required in any cache organization because they are the first reference to an instruction or piece of data. Capacity misses occur when the cache size is not sufficient to hold data between references, and coherence misses are misses that occur as a result of invalidation to preserve multiprocessor cache consistency [2].

Clearly for the reasons mentioned above studying cache hierarchy is extremely important to derive the most performance out of today's machines. This paper is organized as follows, section 2 presents background and related work, section 3 presents our methodology, section 4 presents our cache architecture, section 5 presents our results and section 6 concludes.

2 BACKGROUND AND RELATED WORK

The research carried out in order to study the performance improvements possible by cache hierarchy is truly vast. In this paper we will implement and analyze miss cache, victim cache and prefetch techniques, thus we will focus on discussing work that has been done in these areas.

In [3] the authors describe a *Markov Prediction* scheme which essentially acts as an interface between the on-chip and off-chip cache. The key idea behind the design is that it prefetches the necessary data and prioritizes it to better take advantage of the processor. It mentions important ideas such as prefetch coverage, accuracy and timeliness. A good prefetcher must have large coverage (number of addresses supplied by the prefetcher), it must be accurate (prefetched data usefulness) and it must be timely (make sure that prefetched data will not get displaced before CPU uses it). The key assumption that they make, which might skew their data a little bit is that the processor has on-chip prefetch buffer which is essentially examined simultaneously with on-chip cache (i.e. the prefetched data does not actually displace data which is located in cache).

In [4] the authors describe the implementation of additional fully-associative cache and prefetch buffers. Important vocabulary is introduced which we will use in our paper: miss cache – small associative cache which stores all of the missed addressed. Victim cache – is cache which is sued to hold blocks which were evicted, due to capacity or compulsory misses, from the CPU. Stream buffers start prefetching data starting from the

miss address [in a stream buffer the data is stored in a buffer, not in cache, thus this avoids cache pollution – the stream buffer technique will greatly reduce compulsory and capacity misses]. A straight-forward extension of the stream buffers are the multi-way stream buffers which are useful for prefetching along multiple intertwined data reference streams [4].

In [4] the authors are careful to point out also that most programs/benchmarks lose over half of their performance in L1 cache misses and a relatively small percentage of performance is lost in the second level of cache. This again confirms our original assumption at the beginning of the paper mainly that, L1 is overly important when it comes to improving the overall performance.

[4] Also states that direct mapped cache has a disadvantage of being prone to more conflict misses, it is faster to access. This is indeed a logical statement. However as Fig 4 shows performance impact when we increase the associativity of L1 cache (both i & d) is not improved by a tremendous amount when ran on our bioinformatics benchmark.

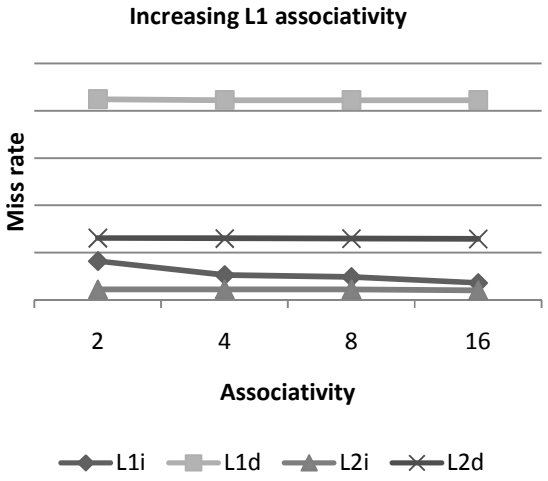

Fig4. Increasing associativity produces virtually no performance increase in our bioinformatics program.

When implementing cache techniques, it is important not to overlook the space constraint. All exogenous factors being equal, victim cache makes better use of the space, because instead of adding essentially the same block that was added to our cache (on a miss) it only adds blocks that were evicted from cache due to a conflict miss. In our case we are given a small 8K L1 cache, and by experimentation small, directly mapped caches benefit the most from victim cache.

We want to know, what the limitation of our cache improvement is. In other words, what is the limiting factor that we are constrained by? To answer this question we can observe that as size grows, a larger percentage of cache misses are due to conflict and compulsory misses. Conflict misses are usually easily resolved by increasing the size of the cache and compulsory misses can be countered by prefetching. The most common technique is the tagging where we set each block's tag bit to zero when it is prefetched and to 1 when it is being accessed. During the transition we begin prefetching the next block. Stream buffers improve on this idea by beginning prefetching successive lines starting at the miss target.

In this paper we implement Athlon prefetching. In other words, we modify our code and issue specific assembly prefetch instructions which are supported by Athlon processors to prefetch a specified address.

3 METHODOLOGY

In this paper, as stated before, we implement miss cache, victim cache, modify the code of our benchmark and experiment with Athlon prefetching.

We use two benchmarks: *Pac-man and Bioinformatics*

To implement victim and miss cache we had to modify Cache Grind to carry our appropriate actions on a cache miss. Fig 5 shows a snippet of the code used to implement miss cache in Cache Grind, mainly if data was not found in cache, we add the missed tag to the buffer [i.e. our simulated victim cache].

```
/* For the miss cache we do not need to worry about
 * copying data over to our L1, since in a miss we duplicate
 * entries
 */
   if (enableMiss)
     for (i = 0; i < SIZEMC; i++)
       if (tag == mc[i])
         return;

   /* A miss;  install this tag as MRU, shuffle rest down. */
   for (j = L.assoc - 1; j > 0; j--) {
     set[j] = set[j - 1];
   }

   set[0] = tag;

   /* For miss cache we insert the missed data [simulated by tag]
    * for VC we insert the thrown out data
    */
    if (enableMiss)
      mc[countermc]=tag;
    if ((countermc++) == SIZEMC)
      countermc = 0;
```

Fig 5: Sample Cache Grind implementation of miss cache

Victim cache was implemented in a similar way, but only instead of adding blocks that were missed, we add blocks that were evicted.

We also modified the code of our Pac-Man benchmark in order for it to be more cache friendly. We know that a C programming language uses row-order memory layout. We noted that when or benchmark program was loading the levels it was using a column order layout, mainly:

```
for(b = 0; b < 28; b++) {
   for(a = 0; a < 29; a++) {
      fscanf(fin, "%d", &Level[a][b]);
            if(Level[a][b] == 2) { Food++; } ….
```

Fig 6: Column-major order.

As Fig 6 shows this code in Pacman caused a huge amount of L1d misses. By switching the two for-loops we are able to decrease amount of cache misses substantially.

Lastly we implement a prefetch mechanism relaying on Athlon specific prefetch instructions (prefetch(%) and prefetchw(%) [for write]) There are commercial compilers such as VectorC available that will be able to dynamically insert prefetch instructions into assembly, however we decided to simply use GCC assembly hints to add support for Athlon prefetches. Fig 6 illustrates a sample implementation of inlined GCC functions that upon compilation will result in assembly generated prefetch instructions at specified lengths.

```
#if defined(AMD_PREFETCH)
static __inline__ void CPU_prefetchwR(const void *s) {
      __asm__ ("prefetchw 64 (%0)" :: "r" ((s)) );
}
static __inline__ void CPU_prefetchR(const void *s) {
      __asm__ ("prefetch 64(%0)" :: "r" ((s)) );
}
#endif
```

Fig 7 By inserting this into our code, we can call prefetch(%) to prefetch 64 bytes ahead (in the above example) [Credit: David Mathog]

4 CACHE ARCHITECTURE

The basic background of what this paper implements has already been discussed in previous sections, so it will not be discussed again here. If details are necessary please refer to previous sections.

For our design we have implemented a 64B victim and miss cache, which according to cacti will take approximately *0.01456 mm^2*.

The design of victim cache still needs to be improved because at this point it is not clear how Cache Grind deals with cache entries that are thrown out and thus more research is needed in that area.

5 RESULTS

For performance analysis we used a perceptron based branch predictor and cycle latencies were estimated with Cacti. The resulting numbers were then entered into a simple performance simulator to generate an estimate for approximate performance given parameters such as cache misses, branch predictor misses and L2 latencies.

First we show the result when using victim and miss cache. We ran our test bioinformatics program and below show the performance increase by using both concurrently.

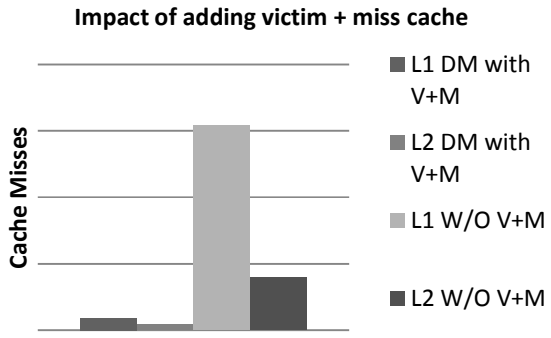

Fig 8. Clearly Victim and Miss cache together greatly improve performance of our basic benchmark (Bioinformatics program)

Next we show performance increase by modifying our second benchmark program (Pac-man) as described in section 3.

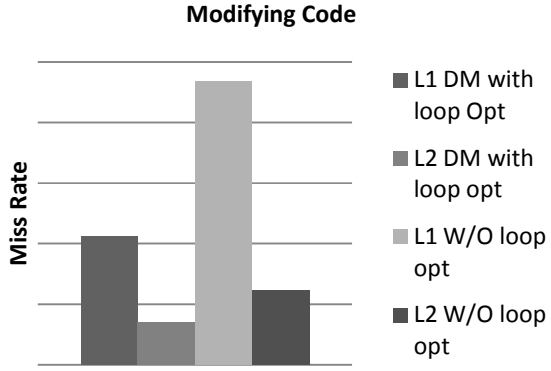

Fig 9: By simple loop rearrangement we are able to improve performance by 2x!

Athlon prefetching: our results regarding Athlon prefetching were inconclusive and need more testing. The reason for this is because prefetching by exactly the right amount is very tough in assembly and more time is needed to provide very accurate model for Athlon prefetching boost.

The basic idea is simple, we put calls to our prefetch function described in section 3 such as *(CPU_prefetchR(&alph[i]);)* where *&alph* is the next address we want to prefetch. We put this call in a loop (as it is the place where most misses occur), however it is not clear to us how to precisely determine the exact timeliness of the needed prefetch length, however this problem is very interesting to us and we will continue working on it.

6 CONCLUSION

This paper discussed the great potential that can be achieved by making small modifications to existing cache hierarchy. Performance increase is highly significant, thus this reason greatly motivates further study in this area. We have implemented victim, miss caches as well as modified a simple loop in our benchmark program. The performance gains we saw were significant, ranging from 2 to 4x performance increase which resulted from simple modifications. Athlon prefetching is a promising technology and we experimented a little bit with it in this project, however due to time constraints we were unable to fully utilize it, nevertheless the literature suggests [AMD papers] improvements of 50-100% are easily achievable with prefetch instructions.

REFERENCES

1. *Dynamic Binary Analysis and Instrumentation*", by Nicholas Nethercote.
2. *A Stateless, Content-Directed Data Prefetching Mechanism*
   Robert Cooksey, Stephan Jourdan
   Dirk Grunwald
   Intel Corporation
   Intel Corporation
   University of Colorado
   rncookse@ichips.intel.com
   sjourdan@ichips.intel.com
   grunwald@cs.colorado.edu
3. P*refetching using Markov Predictors*
   Dirk Grnnwald
   Doug Joseph
   Department of Computer Science
   IBM T. J. Watson Research Lab
   University of Colorado
   P.O. Box 218
   Boulder, Colorado, 80309-0430
   IBM. T. J. Watson Research
   grunwald@cs.colorado.edu
   Yorktown Heights, NY 10598
   djoseph@watson.ibm.com
4. *Improving Direct-Mapped Cache Performance by the Addition*
   of a Small Fully-Associative Cache and Prefetch Buffers
   Norman P. Jouppi
   Digital Equipment Corporation Western Research Lab
   100 Hamilton Ave.. Palo Alto, CA 94301
5. *Effective Jump-Pointer Prefetching for Linked Data Structures*
   Amir Roth and Gurindar S. Sohi
   Computer Sciences Department
   University of Wisconsin, Madison