Skewed-Associative Caches:  
CS752 Final Project

Professor Sohi

Corey Halpin  
Scot Kronenfeld  
Johannes Zeppenfeld  

13 December 2002

Abstract
As the gap between microprocessor performance and memory performance widens, the penalty in going off-chip for a memory (or higher level cache) request is becoming increasingly severe. Therefore, the miss rate of the cache is becoming ever more important in keeping average memory access time down. Since the size of the caches are often constrained so as not to make the processor chip too large (L2 caches already make up about 40% of a current AMD Athlon processor die\(^1\)), it is important to find ways of increasing the hit rate without simply increasing the size of the L2 cache. One way to do this is through a different mapping function, where a certain address is mapped to different blocks in each set, thereby reducing possible collisions. This approach was coined Skewed-Associative Caches by Andre Seznec\(^1\).

We will explore the claim by Seznec that a two-way skewed-associative cache will perform as well as a four-way set associative cache. Additionally, we will explore the claim that for large cache sizes, direct mapped caches perform as well as fully-associative cache. If this is true, skewing would be unnecessary at large cache sizes.

1 Introduction

Over the past decade, the performance gap between microprocessors and memory has increased dramatically. In a modern microprocessor, memory requests have latencies in the tens to hundreds of cycles. In order to decrease this latency, a small amount of fast memory, called a cache, is placed between the processor and the memory. This cache can only hold a small fraction of the address space; however, due to temporal and spatial locality of memory references, it often holds a large portion of the working data set. A cache that operates at speeds comparable with the processor’s clock rate can often deliver requested data within a few clock cycles, decreasing the effective memory

\(^{1}\) 15 million / 37 million = 0.405 http://www.lostcircuits.com/motherboard/asus_a7v/
access latency by a significant amount. In order to alleviate the disparity between the fast cache and the relatively slow memory, many systems have introduced a second level of caching (and recently, even a third level). This second level often works at a slower rate than the first level; however it is both larger and has higher associativity. In systems with multilevel caches, the first level is often designed to provide a fast access time, at the cost of a higher miss rate, and the second level is designed to provide a very low miss rate. One prime example of this is in the recent Intel Pentium 4 processor, whose L1 cache size is half that of the Pentium 3 (8kB instead of 16kB). The designers chose to do this in order to provide a very fast hit time to the processor, which is running at clock rates upwards of 2GHz. The second level cache then aims to maintain a low miss rate through its 256-512kB size and 8-way associativity.

Skewed-associative caches were proposed as a way to decrease the miss rate, while not further increasing the size or associativity. In a single level cache system, skewing functions are useful to help decrease the miss rate, and cause fewer requests to main memory. In multilevel systems, skewed caches would be useful at the second level or higher. They could be used in the L1 cache, but since they add a small amount of logic to the critical path of cache access, they do not meet the criteria of fast L1 cache access stated above.

2 Theory

The principle behind skewed caches is to change the mapping function for each set in a cache. Blocks that are mapped to one spot in a certain set should not map to the same spot in any other set. One example skewing function is a shuffle function with xor. The address is split, as below:

<table>
<thead>
<tr>
<th>Tag</th>
<th>A1</th>
<th>A2</th>
<th>Block Offset</th>
</tr>
</thead>
</table>

This division is not strict, as A1 can potentially overlap the tag, and A2 can potentially overlap the block. The basic requirement is that \( A1 \oplus A2 \) results in values appropriate to address the number of blocks in a bank. For each successive level of the cache, the bits of A1 are shuffled as follows:\[4\):

- bank0: \( A1 \oplus A2 \)
- bank1: \( \text{shuffle}(A1) \oplus A2 \)
- bank2: \( \text{shuffle}(\text{shuffle}(A1)) \oplus A2 \)
- bank3: \( \text{shuffle}(\text{shuffle}(\text{shuffle}(A1))) \oplus A2 \)

The shuffle function is simply a reordering of the wires. Any skewing function that uniquely maps each block in a bank is acceptable; however, using this method to skew the mapping function only adds one xor gate to the critical path of indexing into a bank. An example of a skewing function at work is shown in the appendix.

Skewing reduces the miss rate by helping to reduce conflict misses. Specifically, when there is an \( n+1 \) collision in an \( n \)-way associative cache, skewing can potentially avoid the problem by mapping them to different locations in other banks. In his 1993 paper, Seznec claims that a 2-way skewed-associative cache has the same hardware.
complexity as a 2-way set-associative cache, but performs as well as a 4-way set-associative cache. In addition, we will explore the statement that for a large enough cache size, direct mapped caches perform as well as fully associative caches.

3 Method

When searching for simulation tools to use, we found two of note: SimpleScalar and the "CACHESKEW" simulator written by Seznec for his first paper on skewed associative caches[5]. SimpleScalar was attractive because we already had Spec95 and Spec2000 binaries that would run on that simulator. CACHESKEW was attractive because it already modeled skewed associativity caches.

We decided to modify SimpleScalar to handle cache skewing. The primary reason was that this would not require us to generate large address traces. We also noted that with CACHESKEW it is more difficult to model the L1/L2 cache interaction, whereas SimpleScalar does this out of the box.

In order to simulate our skewed-associative cache, we modified the SimpleScalar 2.0 cache[ch] modules. In SimpleScalar, replacement policy and associativity are both rolled handily into the cache data structure by representing each cache set as a linked list. This structure works wonderfully if there is a unique mapping from one cache bank to another (as there is in a non-skewed cache). However, when there is not a unique mapping from one bank to another (as occurs in a skewed cache), the linked list structure becomes impractical.

So, we removed the linked list. To the cache structure, we added a reference count, and to each block we added a last reference stamp. When any block is accessed, its last referenced stamp is updated to be equal to the reference count of the cache. Using these data members we can then implement LRU replacement.

To index into the cache, we first see if there was a hit in bank 0. If there was not, we skew our block address, and see if there was a hit in bank 1. This continues until we run out of banks, or until we have a hit.

To verify the correctness of our simulation, we compared non-skewed results to the unmodified SimpleScalar. These results matched exactly. We then compared skewed cache results to those generated by Seznec’s CACHESKEW on address traces of identical length. These results differed by only a few tenths of a percent. We believe that this small difference can easily be explained by the difference in skewing functions. Seznec’s simulator uses a number of complex (in software) skewing functions, whereas ours uses a simpler one.

We attempted to implement the shuffle-xor skewing function, but found that implementing it for arbitrary cache and bank sizes was non-trivial. Rather than spend time implementing detailed shuffle functions, we decided to approximate a shuffle function that produced similar distributions of output values but was simpler in software. The function that we chose was $f(bank\_number) = (addr + 4 \cdot bank\_number) \mod bank\_size$. In simulations that we ran versus Seznec’s simulator, this function produced miss rates accurate to three orders of magnitude.

We ran our simulator on 6 Spec benchmarks: vortex, mcf, perlbench, parser, gzip, and mesa. We used the ‘test’ inputs for most of these tests, and the ‘train’ inputs for the
others. These are full runs of the entire benchmark. We would have liked to perform runs that encompassed all of the input files, as well as more benchmarks, but we simply didn’t have the time. Some of the longer benchmarks require more than 20 hours of simulation time to complete a cache level simulation, and we require thirty five of each simulation (seven cache parameter banks * five cache sizes).

We simulated cache sizes of 16k, 32k, 64k, 128k, and 256k, and for each of these sizes we ran simulations for 2-way associative, 4-way associative, and 8-way associative organizations. For all of the associative organizations, we simulated corresponding skewed caches.

Our memory hierarchy consisted of split L1 I and D caches, both of which were direct mapped. The I cache was 8k in size using 64B blocks, and the D cache was also 8k in size, but used only 32B blocks. These parameters were the default in SimpleScalar, and seemed reasonable. The second level unified cache was the focus of our work, since in situations where hit-time is critical, associative caches are not practical as explained in the introduction above.

The sizes that we chose were representative of L2 caches in embedded or low end processing applications, up through high-end desktop processors that use 256-512k caches. We wanted to perform simulations on a continuum of sizes, to more fully investigate the effect of cache size on skewing function. Cache skewing was, according to Seznec, most effective on small caches, so we started there. We would have liked to perform simulations up 512k or larger, as that is a current value in a Pentium4; however, that would have raised the total runs from 35 to 42, or higher. We were forced to draw the line somewhere, and as can be seen in the next section, 256k did not significantly outperform 128k at high associativity. Once again, this is highly workload dependent, as the more data a program uses, the more effective a large cache becomes.

4 Data

Following are plots obtained for our data. They are all of the same format, with miss rate on the y-axis, and Cache configuration on the x-axis. The cache-configuration lists successive sized caches in order, and within those lists the associativity starting with direct mapped.
This plot is too cluttered to draw very many conclusions from, however it shows the overall behavior over all of our variables. Trivially, we observe the intuitive results that as cache size and associativity increase the miss rate decreases. Of more interest to this paper, we can see that in most cases skewed caches perform better than non-skewed caches on the spec benchmarks that we ran. For some benchmarks, like gzip, skewing actually worsens the miss rate. We also note that small, direct mapped L2 caches do not yield very impressive performance. However, as cache size increases, the direct mapped performs much more comparably to a highly-associative cache. To get data more specific than this, we will analyze the graphs for each program individually.
The vortex results appear to be close to our ideal results between skewed and non-skewed caches. The skewed cache has a better miss rate than non-skewed for most of the cache configurations, with the only exceptions being in the small 16k cache. The skewed appears to have its biggest advantage over non-skewed at 64k, after which the gap begins to close. This is to be expected, since as caches get larger, the conflict misses decrease. As mentioned above, skewed-cache helps to remove conflict misses from set-associative cache; hence, as they decrease, the two will perform similarly. Also of note is that the direct mapped cache for a 256k cache performs reasonably close to the 8-way associative. This supports our second claim that direct mapped caches perform as well as fully-associative caches for large sizes. Current processors use 256k-512k L2 cache. Had we modeled 512k caches, we expect that the direct mapped performance would have been even closer to the associative caches. Also in consideration, but out of the scope of this paper, is that hit time is reduced for direct-mapped, so direct-mapped may even result in superior performance for 256k in this benchmark.
MCF benchmark results

MCF also seems to be a good candidate for cache skewing. After some very similar behavior at the 16k and 32k sizes, the skewed-cache significantly outperforms the non-skewed. This gap appears to continue increasing until about 128k. This is a higher threshold than for vortex, however, we expect that had we modeled 512k, the performance difference between skewed and non-skewed would have narrowed. The direct-mapped cache at large sizes does not perform as closely to associative caches as in our vortex benchmarks; however, it performs significantly closer at 128k and 256k than at the smaller sizes. Once again, as the size further increases, we expect this gap to decrease.

**Figure 3** - mcf benchmark results
The results for the gzip benchmark are not as easily explained as those for vortex and mcf above. This was the only benchmark that we ran where skewing was consistently worse than direct associativity. Notably, skewing is not very much worse in any case, and would not decrease performance as much as it would increase other benchmarks. In an attempt to explain the gzip benchmark, we tested a separate skewing function on gzip only. The results were very similar to the current skewing function, once again showing that the miss rate is independent of the skewing function. Our explanation is that maybe it is an artifact of the skewing function that we chose or the replacement policy (LRU throughout all examples), and the particular access pattern of gzip on this input file. For direct-mapped caches, the miss rate is very similar to associative caches at high cache sizes. For 256k, the difference is only a few percent.
The parser benchmark displayed very similar behavior between the skewed and non-skewed versions of the cache. Similarly to gzip, the parser benchmark also shows several data points that have a miss rate that is greater for a skewed cache than a non-skewed one. Even so, the values are very close and in fact data points for medium sized caches (32k and 64k) and average cache associativity show that a skewed cache performs better. General trends such as a decrease in miss rate with an increase in cache size and associativity can be observed, and the difference between direct mapped and set-associative caches also decreases.

The perl benchmark was the only SpecINT2000 one which did not yield useful results. Taking a look at the input files used for the perl benchmark, this makes some sense, as the script used as input is nothing more than a compact loop that goes through several hundred thousand iterations. The problem is likely to be that with such a small working set needed for operation, there are a few initial compulsory misses, followed by a lot of hits in the cache since the entire working set is within the cache.

This was the only floating point benchmark that we were able to run. We also attempted art and equake, but they did not finish in reasonable amounts of time, and we were forced to terminate simulations to free up workstations for other students. If mesa is prototypical of FP benchmarks, then we can conclude that FP applications tend to be more compute intensive than memory intensive, and that cache behavior is not a large factor for FP workloads.
5 Conclusions

We have not found Seznec’s claim that 2-way skewed associative caches perform just as well as 4-way set associative caches of the same size to be accurate for our benchmarks. This might be explained by the fact that Seznec used only 10,000 line address traces, whereas our specs executed up to 2.6 billion instructions, with 123 million references to the L2 cache. For our simulations a skewed cache performs erratically. Sometimes a 2-way skewed cache performs as well as a 4-way associative (vortex, mcf), and other times it performs worse than a 2-way associative (gzip, parser). The general result, however, appears to be a benefit in the miss rate. When skewed-caches work, they can work significantly better than set-associative caches, and when they don’t they work just as well. The hardware cost to implement skewing is small, much smaller than increasing the cache size.

Our second claim, that direct mapped caches of large size perform similar to fully-associative caches, is more of a clear-cut issue. All of the benchmarks support this claim somewhat, with gzip, mcf, perl, and mesa showing the strongest case. This can be explained by the fact that as cache size increases, more and more data can be stored in the cache, finally resulting in a cache that can hold the entire memory, at which point only compulsory misses are relevant and direct mapped or fully associative caches are essentially the same. As many modern microprocessors are using more than 256k of L2 cache, we would have liked to simulate cache configurations of 512k or greater. We expect that the line would have continued to flatten out, as conflict misses continue to decrease.

A further goal that we would have liked to achieve is modeling the execution times of programs, with skewed and non-skewed caches in place. Since there is potentially an extra xor-gate in the critical access path to the L2 cache, skewed caches with lower miss rates might still perform more slowly than non-skewed. We did not have time to simulate this, so we cannot determine the magnitude of the impact of this increased hit time.

Overall, skewed caches appear to be a useful and innovative way to reduce miss rate. However, as the benchmarks show, the results are largely dependent on the workload, and in the case of two benchmarks, the results are worse. Before determining whether skewing is a benefit to a given system, move exhaustive studies should be run to determine the performance. Specifically, we recommend running simulations for more benchmarks, to determine if gzip and parser are anomalies, as well as simulating larger sizes and more replacement policies. As our results were run on the ‘test’ versions of the benchmarks, which are the shortest, running on the ‘train’ or ‘ref’ inputs could also produce valuable insight into the true performance of skewed caches.

6 Appendix

Example skewing function at work:
Three addresses which map to the same block in Bank 0 map to 3 different blocks in Bank 1

References


[4] From CS752 class notes