Classification of cache misses in a SMP using the SESC simulator

Rohit Sud
College of Computing
Georgia Institute of Technology
rohit@gatech.edu

ABSTRACT

Cache misses impose a significant performance barrier in modern day computing. This work attempts to classify the cache misses that occur in a Symmetric Multiprocessor (SMP) based system into compulsory, conflict, capacity or coherence misses. For this, the SESC simulator[1] (Superscalar Simulator) is used to run the benchmark WATER which is a part of the SPLASH2[2] suite. The simulator is modified to classify the number of read and write misses into compulsory, conflict, capacity and coherence misses. The results of the simulation are presented in the work. The work also explains the reason for the trends observed for cache misses

General Terms

Cache Miss, Compulsory Miss, Capacity Miss, Coherence Miss, Conflict Miss, Multiprocessors

1. INTRODUCTION

Cache performance plays an important role in determining an application's performance. With time, microprocessors are becoming faster and faster while the speed of the system memory is increasing at a slower rate. Caches provide an important means to bridge this gap by providing a high bandwidth and low latency store for data used by the processor.

Cache misses impose a significant performance barrier in modern day computing. Cache misses can occur in both L1 and L2 caches. But due to the difference in the speed and size of the two caches, the performance implications are quite different for them. A cache miss in a L1 cache takes only a few cycles to service while a cache miss in a L2 cache can take around 100 cycles to service. Therefore it becomes essential to analyze the cause for these misses, so that they can be reduced in practice. Not only this, classification of types of misses helps in deciding the most appropriate cache system for a system with respect to its size and associativity.

For a multiprocessor system, these misses become even more important as servicing a miss could mean fetching data from a remote processor which can take a lot of time. In recent years Chip Multiprocessors (CMPs) have become popular. Off-chip communication to a L2 cache can take a lot of time which can degrade the performance of the CMP based machine.

In this paper, we attempt to identify and classify various types of cache misses that occur in a MIPS based multiprocessor system. We use the SESC simulator to simulate the WATER benchmark. The WATER benchmark is available as a part of the SPLASH2 suite of multiprocessor applications.

We define the four types of misses as follows. *Coherence misses* are misses that were caused due to invalidation of a cache block by another processor. *Compulsory misses* are cache misses that are caused when the cache block was never ever present in the cache and was brought in for the first time. *Capacity misses* are misses that occur due to finite capacity of the cache and wouldn't have been there if the cache was infinite. *Conflict misses* are misses that are not capacity misses and occur due to limited associativity of the cache and would not have been there if the cache was fully-associative.

"WATER performs an N Body molecular dynamics simulation of the forces and potentials in a system of water molecules in the liquid state. The overall computation consists of calculating the interaction of the atoms within each molecule and of the molecules with each other after a number of time steps." (1) For the purpose of this project, we run the WATER benchmark with 512 molecules.

Similar work had been done by Dubois et al (1). In their work, they had classified misses as cold, true and false misses. They regarded the cold and true misses as actual misses and false misses as useless misses. Instead of comparing the reults with the number of processors, they had compared it with the block size of the cache.

2. METHODOLOGY

The simulation was performed on the 10/16/2007 build of the SESC simulator with some modifications. The WATER benchmark binary was provided from the course webpage[3]. The input for it was also predefined.

The WATER benchmark was simulated using the SESC simulator on 1, 2, 4, 8, 16 and 32 thread. The test system was a 32 processor system with 512KB 8 way set associative, write-back L2 cache. The replacement policy for the cache was Least Recently Used (LRU) and the architecture followed the MESI protocol for cache consistency. Miss classification statistics were collected for each thread configuration. For the purpose of miss classification, we consider the read and write misses only in the L2 cache.

The core of the simulator was not modified but code was added to the simulator to count and classify all the misses that were occurring in the execution of the WATER benchmark. For this work, both read and write miss were accounted for.

[1] Available from http://sourceforge.net/projects/sesc

[2] Available from http://www-flash.stanford.edu/apps/SPLASH/

[3] http://www.cc.gatech.edu/~milos/CS6290F07/

The order used for classifying the cache misses was coherence, compulsory, capacity and finally conflict. The SESC simulator already classifies the read and write misses for a SMP. The addition that was done to the simulator was that it could now classify each miss into one of the four types, compute the sum-total for each miss for each processor and output that information. The algorithm used for classifying each miss into one of the four categories is described below.

2.1 Classification of Coherence Misses

The SESC simulator handles coherence misses internally but in a way not suited for our purpose. The reason is that it overwrites the tag of the replaced block with 0 to indicate that it is invalid. This prevents us from checking if the block that is missing in the cache now was ever present in the cache.

The solution to this is to add a new field for each SMP cache line to store the previous tag of the line after it has been invalidated. Whenever a cache line is invalidated, before resetting the tag, the old value of the tag is stored in a variable. Now if a memory reference refers to the invalidated block, it will miss in the cache. When the cache miss is detected, we check the list of all invalidated tags in the cache (Stored as C++ STL library Set) to find a match for the memory access. If we find a match, it means that the block could still have been there in case no invalidation occurred and so this miss is a coherence miss.

The tag of the invalidated cache block is removed when another block writes to the invalidated cache line. This means that even though the access would have been a hit earlier for a single processor system, from now onwards, when the cache line has been written to by another block, the data could no longer be found in the given location in the cache.

2.2 Classification of Compulsory Misses

Compulsory misses are classified trivially after it is determined that the miss is not a coherence miss. Theoretically the ordering of compulsory and coherence miss detection doesn't matter as a coherence miss can never be detected as a compulsory miss as the tag would have already been in the cache. For this purpose, we maintain a set of all tags that was in the cache at least once. The set is implemented using the C++ STL library Set class. Whenever a tag arrives, it is first compared against the tags already in the set. If it is not found there, the tag is inserted into the set and the miss is marked as a compulsory miss. If the tag is found, then the access is not a compulsory miss and is further passed down to check if it is a capacity miss or a conflict miss.

2.3 Classification of Capacity Misses

Once it has been determined that a cache miss is not coherence or a compulsory miss, the address is tested for being a capacity miss. For this purpose a stack is used which implements the Least Recently Used (LRU) algorithm for replacement. The stack is implemented using a C++ STL vector. The stack stores the tag for the block. The number of lines in the actual SMP cache is the upper bound on size of the vector. The ordering of tags in the stack indicates the recentness of tag access.

When a tag arrives and is already classified as a miss, the LRU stack is searched for a match. If the tag is not found in the LRU stack, then the miss is classified as a capacity miss.

In addition accessing any block in the cache also updates the LRU stack for both hits and misses. Care is taken so that we update the LRU stack *after* we determine the type of the miss and not before that.

LRU Stack Replacement Algorithm

1. Check if the stack contains the tag.
   a. If tag exists:
      i. Erase the tag from its current position.
   b. If the tag doesn't exist
      i. If stack is full
         Remove the element at the front of the stack.
2. Insert the tag at the back of stack.

Figure 1: LRU Stack Replacement algorithm

2.4 Classification of Conflict Misses

If it is determined that the miss is none of the three misses discussed above then it is classified as a conflict miss. In other word, if the tag is found in the LRU Stack after the previous procedure, then the miss is classified as a conflict miss.

3. RESULTS

Figure 2 shows the execution time (in this case, the simulation time) for the application versus the number of threads. Figure 3 shows the variation of Instructions per Cycle (IPC) with the number of threads. The IPC for the first processor is listed as the IPC for all the other processors remains roughly the same (~1.6) for all the simulations.

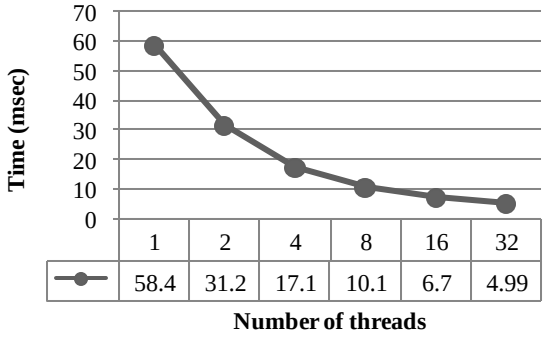

Figure 2: Simulation time versus number of threads

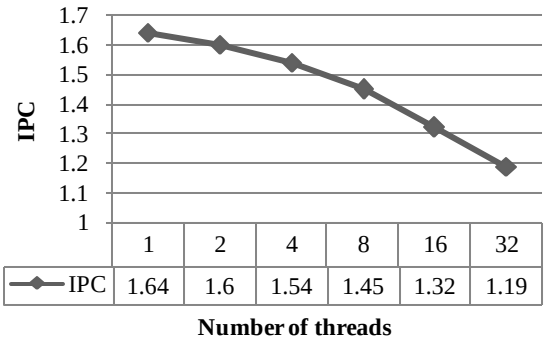

Figure 3: Instructions per cycle (IPC) versus number of threads

Figure 4 shows the total number of misses, classified by type, versus the number of threads. Figure 5 shows the percentage of different misses classified by type, versus the number of threads in the simulation.

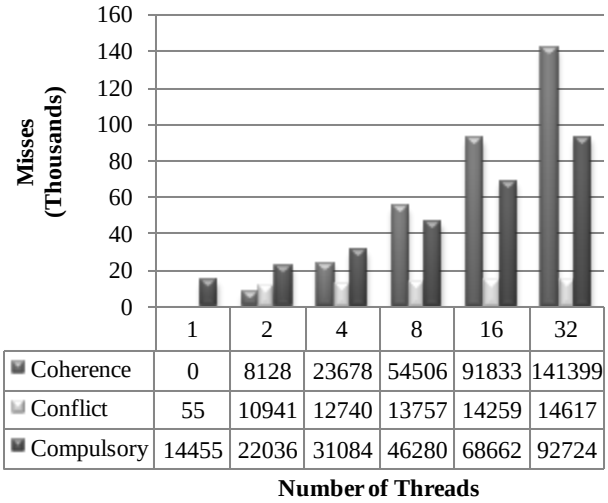

Figure 4: Cache misses by type versus number of threads. Since no capacity misses were observed, they are omitted from the graph.

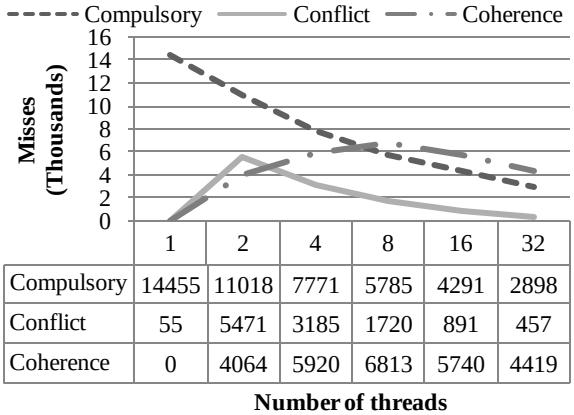

Figure 5: Cache misses per processor versus the number of threads

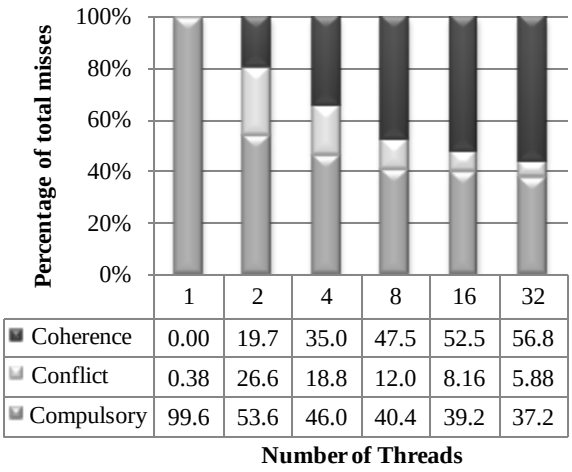

Figure 6: Percentage distribution of Cache misses versus number of threads. Note that due to large size of the L2 cache, there were no capacity misses with this simulation.

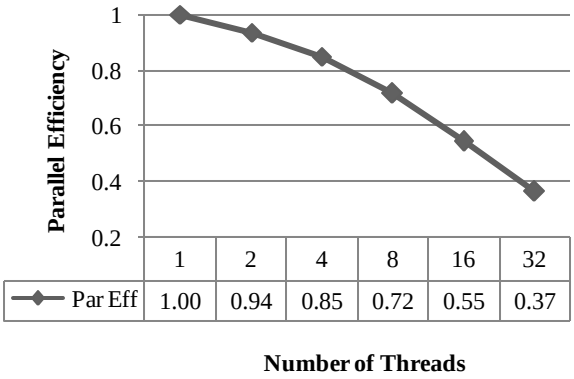

Figure 7: Parallel Efficiency versus the number of Threads. Parallel Efficiency is calculated as $\frac{Speedup over single Processor}{Number of processors}$

4. CONCLUSION

The results of the simulation show that even though the performance of the system improves as a whole when multiple threads are used across multiple processors, the improvement achieved by assigning every extra processor goes down. As a result the parallel efficiency (Figure 7) goes down.

We measure the performance of the whole system by observing the execution time for the benchmark. Figure 2 shows that the execution time for the benchmark decreases with an increase in number of threads. This is expected as increasing the number of threads increases the number of instructions that can be simultaneously executed by the processor. It also distributes the workload amongst processors so that work can be done in parallel.

Figure 3 shows that the IPC steadily decreases with an increase in number of threads. This can be attributed to the effect of sharing data amongst processors which leads to delays. The chief cause of these delays are cache misses which increase in number as we move from a single processor to a multiprocessor system. Cache Miss Penalty, which includes the cycles to fetch data from the memory, introduces delays which leaves very little work for the

processor to do during the miss. In fact, if the miss is for a Load instruction, all instructions in the pipeline dependent on the Load will stall until the cache miss is resolved. For a Store instruction, the effect of a cache miss is lesser as usually there are no instructions that depend on the store.

Figure 4 shows that the all types of misses increase with an increase in the number of threads running on different processors.

Compulsory misses increase simply because each processor has to fill its cache with instructions and data before it can proceed. As we double the number of threads, the compulsory misses increase roughly by a factor of 1.5. The total number of conflict misses also increase but at a very slow rate. In fact, conflict misses per processor actually decrease as we increase the number of threads as shown in Figure 5. The reason for this is that each processor has to store a comparatively smaller amount of data (as the workload is distributed) but now the total cache size (and the set size) for a single processor is the cache size (and the set size) for just one processor in a multiprocessor system. This results in an improvement for the conflict miss rate *per processor* even though the total number increases.

There is an anomalous jump for conflict misses as we move from a single threaded to a multi-threaded system. The reason for this could be the introduction of the invalidation protocol. When we have a read miss for a block in Shared or Exclusive state or a write miss in the Exclusive state, the access is a conflict miss which shows up in case of a multi-threaded system but is absent for a single threaded system.

Increasing the number of threads increases the coherence misses by a large factor. This is due to sharing of data amongst different threads running on different processors. They might be accessing the same data (*true sharing*) or might just be referring to nearby data that happens to lie in the same block (*false sharing*). In this simulation the L2 cache uses the MESI protocol. So whenever one of the threads needs to write to shared, it invalidates the data for all other processors in the system, that were caching the same block. So the next time other processors want to access that data, they have to fetch it again from the memory which leads to cache misses which are coherence misses.

As we increase the number of threads, the number of processors competing for shared data increases by a factor of two. Consequently, each shared data block sees more contention for itself and there are more coherence misses for each block. This increases the number of coherence misses by a large factor for every increase in number of threads.

Figure 6 shows the plot of how much each type of miss contributes to the total number of misses versus the number of threads. For a single threaded system, almost all the misses are compulsory misses. There are no capacity misses as the cache is large enough to hold all data blocks required by the processor. The trend shows that the percentage of compulsory misses decreases with an increase in number of threads and becomes steady after 8 threads. The reason for this is that even though the number of compulsory misses increases, the increase is very less in comparison with the increase in coherence misses. Similarly, the percentage of conflict misses decreases with an increase in number of processors. This is because the increase in the number is quite small in comparison the rate at which the total number of misses increases. Coherence misses witness a huge increase in number as discussed earlier and so form a major part of the cache misses.

5. OBSERVATIONS

During the simulation, no capacity misses were observed. This was due to the fact that a cache of 512KB was big enough to contain all the data required for the simulation. But when the simulation was tried with a smaller cache of 256KB, capacity misses were observed.

6. REFERENCES

1. *The Detection And Elimination Of Useless Misses In Multiprocessors.* **Stenstr, M. Dubois and J. Skeppstedt and L. Ricciulli and K. Ramamurthy and P.** San Diego, CA : Proceedings of the 20th International Symposium on Computer Architecture, 1993.