Cache Enhancement Techniques

Team members:

1) Adarsh Gupta 18116002
2) Aditya Kumar Singh 18116003
3) Umama Alim Khan 18116080
4) Lipi Shreya 18116045
5) Aseem Verma 18114013
6) Shubhrak Prakash 18114075
7) Priyansh Agarwal 18114057

INTRODUCTION

In computing, a cache is a hardware or software component that stores data so that future requests for that data can be served faster; the data stored in a cache might be the result of an earlier computation or a copy of data stored elsewhere.

Cache helps in reducing latency. In computing, memory **latency** is the time (the **latency**) between initiating a request for a byte or word in memory until it is retrieved by a processor. If the data are not in the processor's **cache**, it takes longer to obtain them, as the processor will have to communicate with the external memory cells.

Therefore the purpose of undertaking specifically this project is to improve cache performance and therefore analyse the means to do so which includes pre-fetching techniques and modifying the memory address index.

- The various prefetching techniques that we analyse are:-
    a) **Hardware based prefetching** is typically accomplished by having a dedicated hardware mechanism in the processor that watches the stream of instructions or data being requested by the executing program, recognizes the next

few elements that the program might need based on this stream and prefetches into the processor's cache.

1) **Strided Prefetching** pattern of prefetching instructions is to prefetch addresses that are s addresses ahead in the sequence. It is mainly used when the consecutive blocks that are to be prefetched are s addresses apart.
2) **Sequential prefetch** is a mechanism that reads consecutive pages into the buffer pool before the pages are required by the application.

- Also several memory index modification techniques and search techniques will be implemented to reduce line conflicts.

PREFETCHING

Prefetching is a promising approach to reduce the number of read misses, and thus to reduce the read penalty. Non-binding prefetching does this by bringing into the cache the blocks which will be referenced in the future and are not present in cache. The value returned by the prefetch is not bound; the prefetched block is still subject to invalidations and updates by the cache coherence mechanism. Nonbinding prefetching approaches can be either software or hardware based. Software controlled prefetching rely on the user/compiler to insert prefetch instructions prior to a reference triggering a miss. By contrast, hardware-controlled prefetching schemes utilize the regularity of data access in applications, and need no software support to decide what and when to prefetch.
In this project we are primarily concerned with Hardware Prefetching techniques.

Hardware Based Prefetching

Two promising nonbinding hardware-based prefetching strategies in shared memory multiprocessors are sequential and stride prefetching. Sequential prefetching tries to exploit spatial locality across block boundaries by prefetching consecutive blocks in anticipation of future misses. By contrast, stride prefetching detects and prefetches blocks associated with strides only but does not require spatial locality to be effective. Clearly, the relative performance between the two approaches depends on the amount of spatial locality and stride accesses in an application.
We researched on both these techniques:

Sequential Prefetching

We researched about a relatively new algorithm namely we synthesize a new algorithm, namely, Sequential Prefetching in Adaptive Replacement (SARC), that is self-tuning, low overhead, and simple to implement. SARC improves performance for a wide range of workloads that may have a varying mix of sequential and random data streams and may possess varying temporal locality of the random data streams.
Initially we focused primarily on designing an effective sequential prefetching strategy along with an LRU-based caching policy for housing sequential data. We went through the entire algorithm and then turned our attention to the SARC.
SARC separates sequential and random data into two lists, and maintains a desired size parameter for the sequential list. The desired size is continually adapted in response to a dynamic,

changing workload. Specifically, if the bottom portion of SEQ list is found to be more valuable than the bottom portion of RANDOM list, then the desired size is increased; otherwise, the desired size is decreased. We tried to implement the SARC but due to its complex intricacies were not able to code it.

Strided Prefetching

Software prefetching is a promising technique to hide cache miss latencies, but it remains challenging to effectively prefetch pointer based data structures because obtaining the memory address to be prefetched requires pointer dereferences. The recently proposed stride prefetching overcomes this problem, but it only exploits interiteration stride patterns and relies on an off-line profiling method. A technique to exploit both inter- and intra-iteration stride patterns is proposed, which we discover using object inspection (ie, when dynamically compiling a method, it gathers an “execution profile” of the method by partially interpreting the method at compile time, using the actual values of the parameters and causing no side effects.)

Even Odd Tabulation

There exists three conventional caches:

a) Direct
b) Set Associative
c) Fully Associative

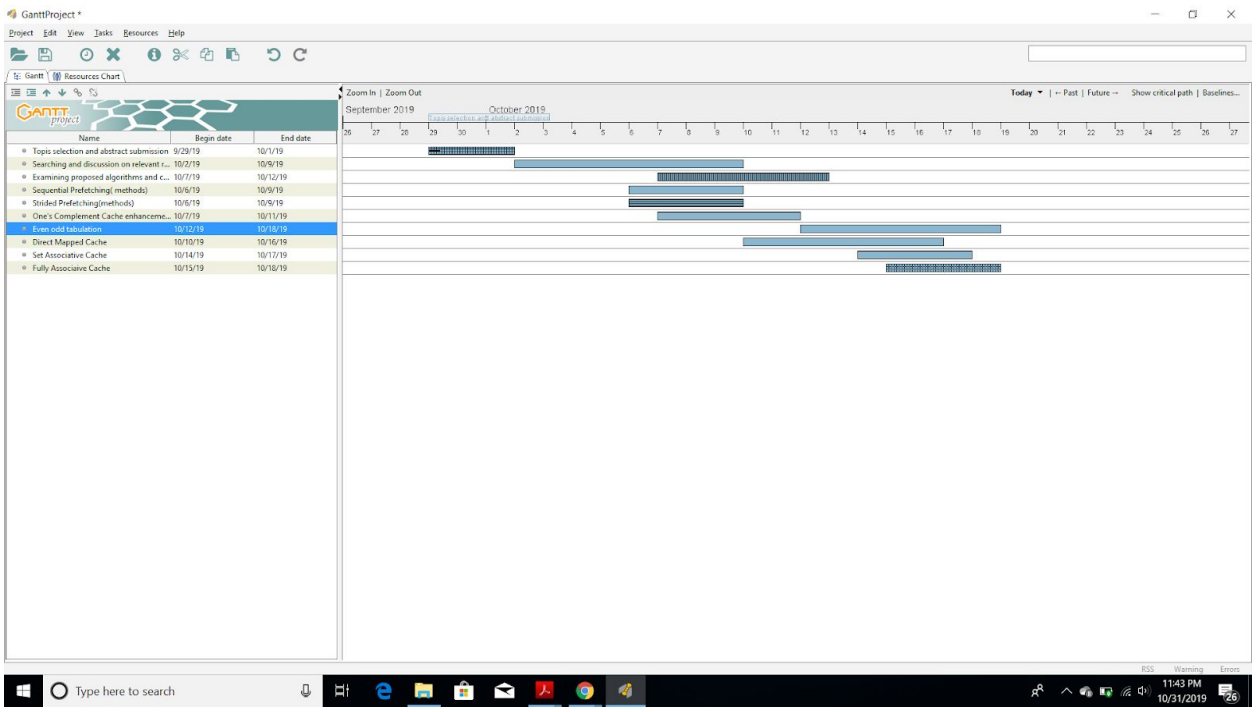
As we move from Direct to Fully Associative the miss rate decreases however on the flip side the access time increases. Thus we decided to exploit the miss rate of the Fully Associative Cache but meanwhile improve the access time.

Even Odd Tabulation tries to improve average access time in case of a fully associative cache by comparing the least significant bit of the memory location with those already present in cache,thereby reducing the number of memory locations to be searched and optimising cache performance.

One’s Complement Cache Method

Cache misses caused by line interferences are minimized by means of evenly distributing data items referenced by program loops across all sets in a cache. Evenly distribution of data in the cache is achieved by making the number of sets in the cache a prime or an odd number thereby the chance of related data being mapped to a same set is small. The proof of the same is examined in this project.

Gantt Chart



References

1. **A One's Complement Cache Memory** by Qing Yang and Sridhar Adina
2. **Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors** by Fredrik Dahlgren.
3. **Data Prefetch Mechanisms** by Steven P. Vanderwiel
4. **Sequential Program Prefetching in Memory Heirarchies** by Alan Jay Smith.