CIS 629: Project No.2

Analysing an Address Trace

Sameer S. Shende

December 6, 1994

Address Stripping Program

The program to filter the large trace was written in C++. The logic for *str* deals with creating a direct mapped cache with N sets each of which points to a line having L bytes per line. The input trace file, N and L are taken from the command line and the address and type of reference fields are read. Then the address (unsigned long) is used to extract the index field (index to the Sets) andthe tag which is to be compared. The tags are compared and those addresses which generate a miss are logged onto an output trace file. Initialisation transient effects are ignored at this first stage of filtering. Multiple traces were generated using different values of N and L for different simulations.

Cache Simulator Implementation

In the second stage the reduced trace is analysed for various values of K and data is analysed. The design of the Cache is done in C++ using Object Oriented Techniques. At the lowest level is an *Item* which contains a tag and has pointers to elements in the list. A special type of *List* is developed using Items. It is used to simulate the Least Recently Used Algorithm where items can be brought to the top of the list (when accessed), the list can be searched for a tag and if the item is not in the list, the last item (Least Recently Used item) is discarded and a new item containing the tag is brought to the top of the list (Most Recently Used position).

The Cache directory is designed as an array of sets each of which contains the LRU list. The number of items in the list is K. There are N such lists in the directory. (The item has a pointer to a line in memory in case it is needed to simulate the memory in future). The Cache is a class which has its member functions like Reference, Statistics, etc. The whole software was developed module by module (item, list of items, Cache) and each was tested independently (as classes) and was integrated together to form the Cache Simulator.

In the first case the data and instructions are all in one cache and the references which produce a miss due to initialisation transients (bit in Item is negative), are discarded from the calculations. The miss ratio is calculated by using the total number of references as in the original trace. It was verified that the miss ratio was same for the reduced trace as for the original trace.

Different runs of the simulator for different traces (N,L values) produced the output which was plotted. In one run of the simulator, for a certain value of K, multiple hit ratios were calculated using the array of hits as described in Stone. So, K could be varied in a single run to generate ratios for values of K less than those specified by the user.

In the second case, two caches were used - one for data and the other for instructions. Since *Cache* was a class, these were just two objects which acted as instances of the class and the effort required to implement this was very nominal as the Cache contained its data seperately, and statistics were calculated independently. The data for different traces was plotted.

Conclusion

Having simulated the cache for different values of N, L and K, it was observed that if a 32KB cache was used (with the common data and instruction cache), it could perform well with N = 128 sets with L = 32 bytes per line and in an 8 way (K = 8) set associative cache giving a hit ratio of 0.99988.

If seperate data and instruction cache is used with the same size (total size = 32 KB), then an arrangement with N = 128 sets, with L = 32 bytes per line in a 4 way (K = 4) set associative cache resulted in a hit ratio of 1 for data cache and 0.99902 for the instruction cache. This appears to be an optimum arrangement examining the data from various runs.

If the size of the cache is not a significant constraint then N= 512 sets with L = 64 bytes per line with K = 2 with seperate data and instruction cache gives a 64KB data and 64KB instruction cache which results in Data and Instruction hit ratios of 1. This is more expensive to implement but gives better results than the above configuration.

Appendices

The Appendices consist of

1. Plots for common cache for different N,L values with K varied from 1 to 8. N=64,L=16 (8KB), different combinations for implementing 32KB cache , and the big 256 KB cache with N=512, and L = 64. In one plot the comparison of the three 32 KB options is shown.

2. Plots for separate instruction and data caches with the above configurations is presented.

3. Script file for the output of sep (separate data and inst. cache) and sim (common Cache Simulator) for different traces is included. This data contains the actual hit ratios for the different runs.

4. Source code listing for the three programs - *str* (address trace stripper), *sim* (common data and instr. cache simulator) and *sep* (separate data and instruction cache simulator) are included. Note that since sep uses the bulk of the code from sim (Cache Class implementation, LRU List implementation), only the main function is included. A significant portion of the code can be reused in similar projects in future, if need be.

   In this way, the Cache simulator was implemented and the cache design was done using this tool.