Additional Profiling of applications in Sim-Profile

Computer Architecture Project
Fall 2003

Presented By,
SridhMya Balakrishnan
Ramya Dharshini Chandrasekaran
Sunita Garg
Table of Contents

1) Introduction
2) Approach
3) Implementation
4) Experimental Setup
5) Analysis and Conclusions
6) Future work
7) Individual work
8) References

Appendix A – Graphical Results
Appendix B- Modified Code
Appendix C-Outputs
Introduction:

The gap between the latency of accesses to main memory (off chip DRAM) and the latency of accesses to on-chip memory (SRAM) has motivated the design of several techniques for improving the performance of a computer’s memory system, including cache-memory hierarchies. To upgrade the performance of a computer’s memory system, it’s important to observe the locality required by the access patterns. Through this project, we analyze the frequency of occurrence of various memory access patterns, which crucial for the reduction of memory latency.

Approach:

The aim of this project is to perform a quantitative analysis for a number of memory access patterns and to investigate methods on how this information can be used to improve a system’s behavior. By analyzing the frequency of these memory access patterns, we propose to redesign the cache on the basis of the most recurring pattern. This enhances the performance of the system without additional hardware.

We classify the instructions into six types.

- **SA**-Memory accesses of executions of instruction X which always address a single location in memory.

- **LD**- Memory accesses of executions of instruction X which have addresses limited to at most ‘D’ distinct addresses, where D can be varied.

- **LR**- Memory accesses of executions of instruction X which have addresses within a range ‘R’ of memory addresses, where ‘R’ can be varied.

- **SC** - Memory accesses of executions of instruction X, which have addresses which follow stride pattern with a constant offset

The two classifications that we propose are

- **SP**- Memory accesses of executions of instruction X, which have addresses which follow the stride pattern with the non-constant offset.

- **TL**- Memory accesses exhibiting temporal locality with respect to A previous memory access instructions.
Implementation:

Our initial step in the project was to read the simple scalar simulator files – especially sim-profile.c and understand the working of the simulator.

A data structure was coded as a linked list of PCs, each of which point to another linked list of effective addresses. A hash table is being used for saving time during the traversal of the linked lists. This helps in faster execution of the code, by faster retrieval of data.

Our functions to modify sim-profile.c are written in a header file called “ourheader.h”. When a load instruction is decoded, the simulator sets the flags to the instruction class (eg:F_STORE). If this instruction is an memory access instruction, the instruction’s PC is searched in the data structure. If it is found, then the effective address of this instruction is searched in the list of memory instructions already accessed by this instruction. If the address is not found, then it is added to the list.

The data structure is populated in this manner. After this is done, the address classification is done by the addr_classfiy() function. The counters for the PCs and memory accesses are incremented in the getval() function and is included into the stat_reg_counter function. This displays the final value of these counters along with the other sim-profile outputs.

We are also writing the PC by PC classification into a file-“op_pc”. The 50 most frequently used PCs and their classifications are written to the “op_results” file.

Experimental Setup:

This simulation was run for three benchmarks- gcc, compress95 and go with various values of D and R. The gcc and go were run with the maximum number of instructions executed restricted to 700,000 and the compress95 with the maximum number of instructions restricted to 10,000. The values of D and R were varied 3 times each and simulation results obtained. The various values of D were 8,16 and 32 and the values of R were 8,32 and 128. Various graphs were plotted for these different conditions. These graphs are included in the Appendix.A.

We had some problems giving command line arguments for values of D and R. So we hard coded these values in the program.
Analysis and Conclusions:

We analyze the behavior of gcc, go and compress 95 for chosen memory access patterns. Related graphs are included in the Appendix. A. In case of memory accesses of execution of instruction X having addresses limited atmost ‘D’ distinct addresses i.e. LD on the variation of the value of D from 8 to 32, consistency was observed in number of memory locations accessed. But gcc showed maximum number of distinct memory locations accessed for D=16 in contrast to consistent rise in memory accesses with increase in the number of allowed distinct addresses.

In case of memory accesses of executions of instruction X having addresses within a range ‘R’ of memory addresses, i.e. LR, on the variation of R from 8 to 128, consistent rise was observed for number of memory accesses, for three the benchmarks.

Next, analysis was carried out to find the most frequently occurring memory access pattern. It was found that for benchmark gcc, the highest number of memory locations were accessed by the instruction following LD pattern, followed by LR, SA and TL in that order. For benchmark go, though LD was having the highest number of memory locations accessed, it was followed by SC and SP (discontinuous stride pattern). For benchmark compress 95, LD again had the highest number of memory locations accessed with LR and SA following. In this case SP and SC showed a minimal memory access which is similar to benchmark gcc.

When frequency of memory location accessed was compared, it was found that LD is the most occurring pattern.

We then plotted the cache hit rate for the three benchmarks and found that gcc has the highest cache hit rate. From the calculation of percentage of load and store instructions in each benchmark, we observed that gcc has the highest percentage of load and store instructions and hence the highest cache hit rate.

We observed that LD is the most frequently occurring pattern in the load-store instruction scenario and hence forms the highest portion load-store percentage for any benchmark. In order to have higher cache hit rate, we should design the architecture of our cache based LD pattern which will reduce memory latency.
**Future Work:**

Locality of memory accesses is the cardinal principal guiding the design of cache hierarchies. The bottlenecks provided by the memory accesses both in power and performance are overwhelmed by observing the patterns in memory accesses. So, the design of caches, by empirically observing accesses patterns of different benchmarks, typical to a particular class of computers is sensible. This has instigated computer designers all round the world to come up with great memory designs.

In this project we implemented three benchmarks viz. Go, Compress95 and Gcc. It was found that LD forms the majority of memory accesses for all the three benchmarks. It is concluded that memory should be designed such that temporal locality is fully exploited. Compress95 is smaller in terms of instructions when compared with other two benchmarks and it has very less ‘percent conflicts’. This is could be attributed to the fact that its code would have been optimized to yield good ILP, thus better performance without any bottlenecks. At the same time, we can also say that compress95 shows poor temporal locality as well.

Also another interesting pattern we tried to implement but could not complete was to find patterns among pointers in a linked list.

Imagine a linked list of the form

```c
struct b { int x, struct b * y};
struct b* p, *q;
q = build_list(SIZE);

Traverse the list
for(p = q; p!= NULL; p=p->y)
{ n = n + p->x; }
```

The addresses in the above sequence that correspond to the updates of pointer p is of the pattern

\[ x, *(x) + y, \text{ and } *(*(x) + y) + y \]

Implementing this type of pattern and its quantitative analysis can be helpful as sometimes pre-fetching data from well predicted loads is often sufficient to mask a number of cache misses due to loads that are not predicted.
Individual Work:

Coding: Srividhya Balakrishnan
Debugging and Simulation: Ramya Dharshini Chandrasekaran, Sunita Garg.
Research for the two additional classifications: Ramya Dharshini Chandrasekaran, Sunita Garg, Srividhya Balakrishnan
Analysis, Results and Report: Ramya Dharshini Chandrasekaran, Sunita Garg.

References:
1) P Grun, N Dutt and A Nicolau, “Access Patterns based Local Memory Customization for low power Embedded systems”, Center for Embedded systems, UCI, Irvine CA
2) D. Burger and T.M Austin, The simple scalar tool set, Simple scalar web site.
4) http://www.ece.rochester.edu/~mihuang/TEACHING/OLD/ECE404_SPRING03
Appendix. A-Graphical Results

Fig. 1

Varying value of D for gcc

Value of D

Fig. 2

Varying value of D for go

Value of D

Fig. 3

Varying Value of D for compress 95

Value of D

Fig. 4

Varying R for go

Value of R

Fig. 5

Varying Value of R for gcc

Value of R

Fig. 6

Varying value of R for compress 95

Value of R
Fig. 7

Frequency of memory accesses in Benchmark gcc

Memory Access

Fig. 8

Frequency of Memory Access for Benchmark go
Frequency of Memory Access in Benchmark compress95

<table>
<thead>
<tr>
<th>Memory Access</th>
<th>npc</th>
<th>nacc</th>
</tr>
</thead>
<tbody>
<tr>
<td>SA</td>
<td>612</td>
<td>1115</td>
</tr>
<tr>
<td>LD</td>
<td>794</td>
<td>1730</td>
</tr>
<tr>
<td>LR</td>
<td>633</td>
<td>1329</td>
</tr>
<tr>
<td>SP</td>
<td>0</td>
<td>329</td>
</tr>
<tr>
<td>SC</td>
<td>0</td>
<td>122</td>
</tr>
<tr>
<td>TL</td>
<td>253</td>
<td>641</td>
</tr>
<tr>
<td>NA</td>
<td>606</td>
<td>606</td>
</tr>
</tbody>
</table>
Comparison of memory access patterns for different benchmarks

![Graph showing memory access patterns for 'compress-nacc', 'gcc-nacc', and 'go-nacc'.](image10)

Fig. 10

Comparison of cache hit rate and % of memory access of various benchmarks

![Graph showing cache hit rates and load/store instructions for 'compress95', 'go', and 'gcc'.](image11)

Fig. 11