CS 6290 Project 1: Cache Design

Sam Britt

Feb. 24, 2012

Experimental Approach

I first attempted to find the optimal cache parameters for a given trace and a budget of 1 MB total storage using any of several constrained optimization techniques. For example, the scientific Python package `scipy` provides `fmin_slsqp()`, which will minimize an objective function (in our case, the AAT from the simulation results), given a list of equality and inequality constraints. To list all the constraints takes some thought, but it can be done: the total storage must be less than 1 MB, $C_1 \leq C_2 \leq C_3$ and similar for B and S. In addition, there are some more subtle constraints given the meaning of these values. For example, $C_i - S_i - B_i \geq 0$ for $i \in \{1, 2, 3\}$, which expresses that there must be at least one set in the cache.

These constraints can be enumerated, but the issue I ran into using a packaged optimization routine was the integral nature of the parameters. In finding the gradient of the objective function, the parameters are modified by small values to find a numerical derivative. However, the parameters must be cast to integers for the cache simulation, the perturbation in the parameters vanishes due to rounding, the result of the cache simulation is identical to that before the perturbation, and the optimization algorithm thinks it found a minimum (the gradient appears to be zero). I could not force the algorithm to vary the parameters as integers, and did not know how or have the time to implement my own.

So for these reasons I took a brute-force search approach. The nine-dimensional space of the input parameters is far to large to search exhaustively, even given the constraints, so a search pattern was chosen to hopefully give the best results early in the search. The basic assumption taken was

- A bigger total cache is better than a smaller cache.

This simply expresses the idea that, the more data we can keep in the cache, the less we are forced to go to memory, which should result in better performance. This drives the search process: instead of searching through the possibilities from small caches to large caches, we start with very large caches ($C_3 \approx 20$) and work our way down.

Since the total cache area must be less than 2^{20}, certainly C_i for any i must be no greater than 19 (if there is more than one cache and there is nonzero

overhead). This provides an upper bound for any cache. Furthermore, since the L2 cache's parameters must be between those of the L1 and L3 caches, it is actually easier to specify these last and let them vary the fastest. That is, the pseudocode for the entire algorithm is

```
Let the L1 cache parameters run from larger caches to smaller
    Let the L3 cache parameters run from larger caches to the L1 parameters
        Let the L2 cache parameters run from the L3 to the L1 parameters
            if the resulting set of parameters forms an invalid cache, ignore it.
            else simulate the cache, record the AAT if it is smallest yet seen.
```

Note that the space tested is larger than the valid cache parameter space. Any set of parameters resulting in invalid caches are not simulated. This algorithm was run for a reasonable amount of time (around two hours on an Intel 2.4 GHz Core 2 Duo) and the minimum average access time seen so far was taken as the solution. If that value was not unique, the solution was taken to be the one most efficient in space as well; that is, the one with the lowest total cache size.

Results and Discussion

Table 1 shows the optimization results for each of the provided traces, including the minimum average access time, cache size, and the cache parameters for each cache. Cache size is reported both in total kilobytes, and as a percentage of the 1 MB budget.

All of the simulation results show an average access time of 2 ns. Since AAT must be greater than zero, and the measurement has a precision of 1 ns due simply to rounding error, this seems like a reasonable lower bound, and shows that the brute-force search pattern described above is reasonable. There were many solutions of 2 ns observed for each trace; the solution taken was the one with with the smallest total cache size. Note that the total cache size ranges from 75 % down to as low as 10 % of the total budget. It seems that 1 MB is more than enough cache space, at least for these traces.

Table 1: Optimal Cache Parameters for Each Trace

Trace	AAT	Cache Size		L1			L2			L3		
	(ns)	(kB)	(%)	C	B	S	C	B	S	C	B	S
bzip2	2	96	9.4	15	13	0	15	13	2	15	13	2
cg	2	772	75.5	17	6	0	17	11	0	19	11	1
gcc	2	192	18.8	16	11	0	16	12	2	16	12	3
go	2	384	37.5	17	14	0	17	15	0	17	15	1
mcf	2	384	37.5	17	15	0	17	15	2	17	15	2
parser	2	768	75.0	18	13	0	18	13	2	18	13	5