Laboratory 1

Yrkkö Äkkijyrkkä yrkko@kth.se   Martin Yrjölä yrjola@kth.se

3.2 Collecting statistics from simple cache

The performance results are given in tbl. 1. It is clear that the parser benchmark has the best cache performance because it has the highest hit rate. The vortex benchmark has the worst performance only scoring a hit 56.9 percent of the time.

Table 1: Cache performance with default settings

Benchmark	Inst hit rate	Read hit rate	Write hit rate	Hit rate
parser	92.8%	87.2%	63.4%	84.3%
equake	79.3%	87.1%	80.9%	81.6%
vortex	58.7%	74.8%	28.2%	56.9%

3.3 Determining benchmark working set size

By increasing the cache size the amount of conflict and capacity misses will decrease. Because the cache has been warmed up in our benchmark, there should be very few compulsory misses. The cache size at the point when the hit rate stabilizes should give us an approximation of the program's working set. A visualization of the hit rate is presented in fig. 1. We also tested the performance by increasing the associativity to 4 with the aim of decreasing conflict misses in order to get a better approximation. The graph suggests that the parser benchmark has the smallest working set at approximately 60kB. The working set of the vortex benchmark is around 2MB. The equake benchmark has the largest working set. We estimate it is approximately 4MB.

3.4 Minimal instruction and data caches

The performance using one cache line with different sizes is presented in fig. 2. Using one cache line will give us a hint of the spatial locality of the data. A line

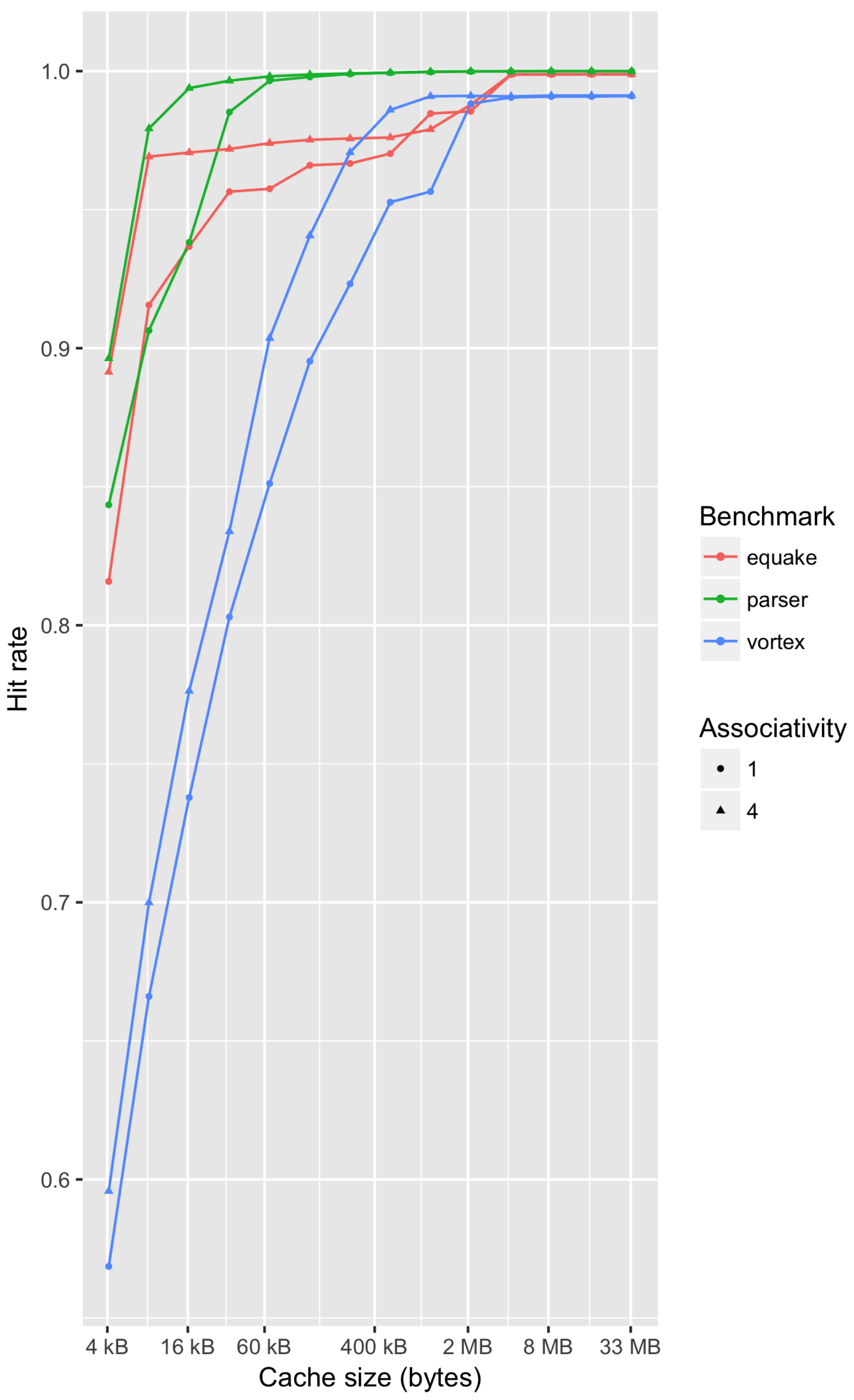

Figure 1: Hit rate for different cache sizes and degrees of associativity

size of 32 bytes is quite small and will only perform well if the data is accessed sequentially. My intuition says that code is accessed mostly in this manner. The program counter will keep on increasing until a branch happens. The UltraSparc T1 processor has instructions of 32 bits, so without branches we should see a hit rate of 7/8 with a line size of 32 bits and a 31/32 hit rate for a line size of 128 (*UltraSPARC Architecture 2005*, n.d.). However, the performance doesn't come close to these values. Maybe there's more branches in the benchmarks than in typical code. One thing to note is that the instruction fetch hit rate almost doubles with the larger line size.

The data reads and writes share the same cache, so it is very likely that they will interfere with each other. We see a big variance between the benchmarks. The parser has the most sequential reads, but the worst write performance. The equake benchmarks has the best write performance.

The question was about comparing relative locality of data against instruction accesses. The data suggests that instruction accesses tend to follow the principles of temporal and spatial locality quite well. Data access is very much dependent on the benchmark. The parser benchmark achieves higher read hit rate than the code, but both the parser and vortex suffer of bad write performance.

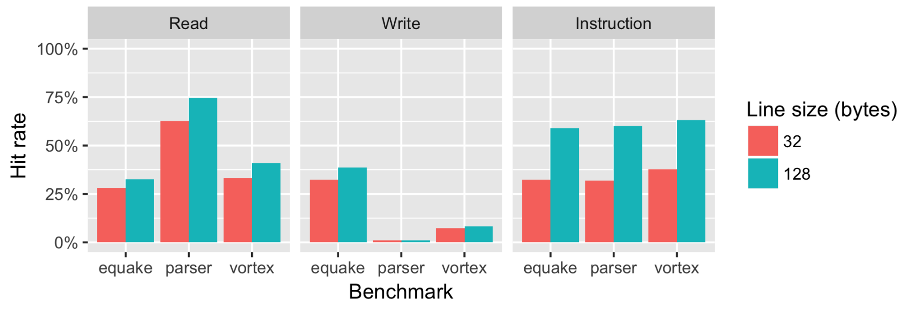

Figure 2: Benchmark performance with one cache line

3.5 Collecting statistics from a cache hierarchy

The performance for the cache hierarchy for different sizes of L2 cache is given in fig. 3. The first question was to assess how effective the L2 cache is at collecting misses from the L1 level. We would argue that the L2 cache performs very well, because the hit rate is very high for every benchmark.

The second question was related to how the size of the L2 cache changes its performance. We choose to analyze the performance in the vortex benchmark, which has the most variance in the L2 results. The small size of 8 KiB has significantly lower hit rate than the 500 kiB and 1 MiB L2 caches. The difference between 500 kiB and 1 MiB is negligible, so it doesn't make sense to invest in a larger L2 cache for this benchmark. Choosing between 8 kiB and 500 kiB is a tradeoff between cost and performance. In the next section we will give a useful metric for assessing the performance between the different sizes.

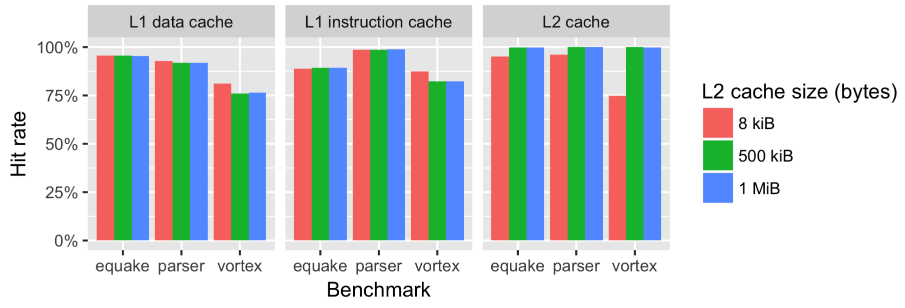

Figure 3: Cache hierarchy performance

Next we will calculate the average memory access time. We will use the following formula to calculate it.

$$3 + L1_{\text{miss rate}} * (10 + L2_{\text{miss rate}} * 200)$$

The average access times are given in tbl. 2. We calculated them by aggregating the statistics for all the benchmarks. As expected the access time gets shorter by increasing the cache size.

Table 2: Average memory access time for the cache hierarchies

L2 cache size	Average memory access time (cycles)
8 kiB	6.1996
500 kiB	4.1795
1 MiB	4.1769

4.4 Multithreaded performance study

The miss rates for the multithreaded benchmark are given in fig. 4. The results for the L1 cache are averaged over all the caches. What we can observe is that the miss rates decrease substantially for the L1 cache when we quadruple its size, this is expected because there is likely a lot of conflict misses with the smaller cache. The miss rates for L2 instruction fetch and write increases when L1 cache is enlarged. The main reason is that earlier the L1 conflict misses got served by the L2 cache but with the larger L1 cache most of the conflict misses become hits and the thus there are less hits overall in the L2 cache.

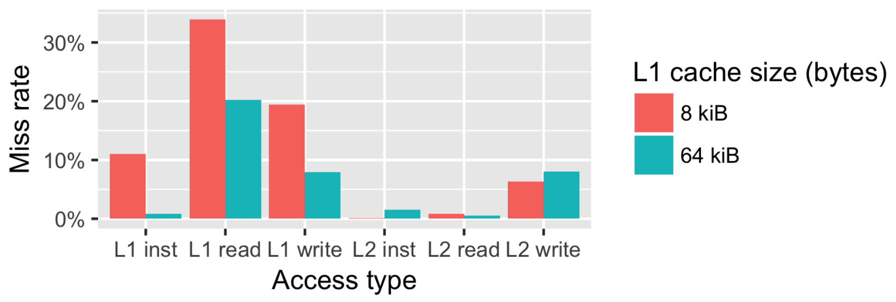

Figure 4: Multithreaded cache performance

MESI statistics for the L1 caches are given in fig. 5. The values are the arithmetic mean of all the statistics on the different processors. The L1 cache is write through, therefore there are no 'modified'-transactions. The exclusive to shared statistics are increased because it is more likely that any given address is available in the larger caches. This also leads to a dramatic increase in invalidates because more data is shared in a larger cache and thus a write has a higher probability to invalidate more data than with the smaller cache.

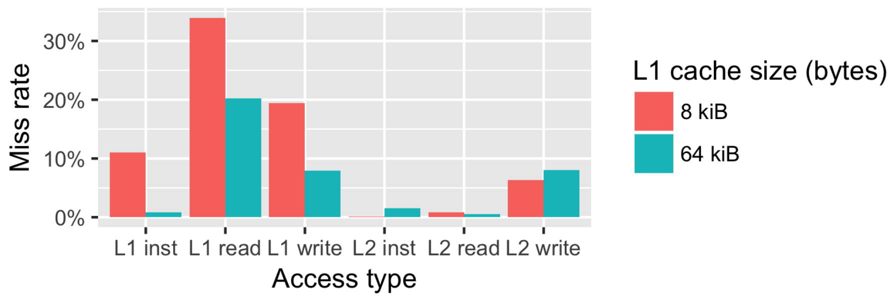

Figure 5: Multithreaded MESI statistics

5.5 Critical sections

1. The data is most of the time inconsistent when using `./lab1.5 -t critical -c pthreads`, increasingly so when adding threads to the execution. This is explained by race conditions because there is no synchronization. When one thread has read the counter value it should be locked from from reading and writing by other threads until the value has been incremented/decremented.
2. Adding the `enter_critical()` and `exit_critical()` makes the data

consistent across executions by using mutexes.

3. Dekker's algorithm
    a. The variables are marked volatile in order to protect false writes because both threads are accessing them.
    b. The CPU's out-of-order execution causes the inconsistency of data.

4. Memory fence Implementation:

```
if(thread == 1){
    flag[1] = true;
    MFENCE();
    while(flag[0]){
        if(turn != 1){
            flag[1] = false;
            while(turn != 1){
                //Busy wait
            }
            flag[1] = true;
        }
    }
}
MFENCE();
```

5. Atomic instructions

Bibliography

UltraSPARC Architecture 2005. n.d. http://www.oracle.com/technetwork/systems/opensparc/t1-06-ua2005-d0-9-2-p-ext-1537734.html: Sun microsystems.