Dr. Jens Teubner, Cagri Balkesen Discussion: 2012-10-04
Department of Computer Science, ETH Zurich

Introduction

The goal of this first exercise is to become familiar with low-level caching issues in data intensive applications such as databases. We ask you to implement a simple C program that measures the access times for different memory access patterns. Then use these results to estimate the sizes of your L1 D-cache, L2 cache and the TLB cache.

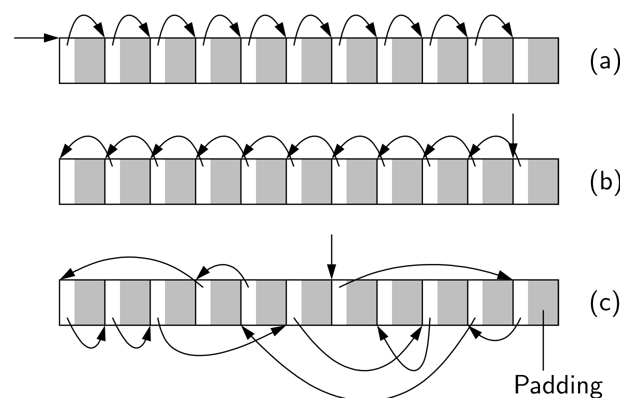

Figure 1: (a) sequential access increasing addresses, (b) sequential access decreasing addresses, (c) random links

We work with data-dependent memory access. First, we allocate an array of elements that are chained to a list. The elements are of the following type:

```
struct listelement {
   struct listelement *next;
   long int padding[NUMPAD];
}
```

We obtain data-dependency by traversing the list using the `next` pointers instead of the array index. The elements are connected in the three different orders shown in Figure 1. Two variants (a) and (b) use sequential access (in opposite directions). The third (c) uses a random order (permutation). The size of an element is varied using the `NUMPAD` parameter that controls the number of padding words in the array. The experiments can be performed both in 32-bit and 64-bit mode. In 32-bit mode `sizeof(struct listelement) = 4*(NUMPAD+1)` bytes and in 64-bit mode `sizeof(struct listelement) = 8*(NUMPAD+1)` bytes. The overall size of the working set is given by the number N of list elements. For each configuration, which consists of N, `NUMPAD` and the traversal order from Figure 1, we measure the wall clock time for a given number of runs, e.g., 100,000. Using different configurations determine the

- size of your L1 data cache (kB)
- size of your L2 cache (kB)

- size of your TLB cache (number of entries) .

Plotting your measurements will help you analyze the data. Please bring your results to the exercise class on October 4th for a lively discussion.

Tasks

Determine Cache Size

1. Implement the linked list in C and add code that measures the execution time of the memory access.
2. Vary the size of the working set by the number N of list elements in powers of two up to 64 MB.
3. Vary the size of the elements, i.e., change `NUMPAD`.
4. Run different access patterns (sequential vs. random).
5. What happens for write access (e.g., writes to the padding structures)?
6. Identify the sizes of your L1 data and L2 caches from your measurements.

Determine TLB Size

1. Modify the code such that (a) each element occupies an entire cache line and (b) that each element is located on a separate 4 kB memory page.
2. Compare the results and try to estimate the size of the TLB cache.

Hints

This assignment is based on Ulrich's Dreppers paper *What every programmer should know about memory*[1]. In particular, Section 3.3.2 "Measurements of Cache Effects" might be helpful for this exercise. [2]

Time Measurements

For your time measurements you can use one of the POSIX functions `gettimeofday()` or `clock_gettime()`. On Windows you can use the `GetTickCount()` function. Note that these functions have different resolutions[1]. Alternatively, you can use the following inline assembly code to read the *Time Stamp Counter* (TSC) register present on all x86 CPUs since the Pentium processor.

```
static inline void
getcyclecount(uint64_t* cycles)
{
  __asm __volatile(
    "cpuid            # force all previous instruction to complete\n\t"
    "rdtsc            # TSC ->  edx:eax \n\t"
    "movl %%edx, 4(%0) # store edx\n\t"
    "movl %%eax, 0(%0) # store eax\n\t"
    : : "r"(cycles) : "eax", "ebx", "ecx", "edx");
}
```

[1] The resolution can be significantly lower than the time units the functions return.

The value of the TSC register is incremented during each tick (e.g., at the CPU clock). In the assemly code `cpuid` is used as a synchronization instruction, such that all outstanding instructions retire before `rdtsc` is scheduled. Additional hints:

- Reduce noise by terminating interfering processes, e.g., updatedb. Consider using median of single runs instead of average (median is less sensitive to "outlier" noise than average).
- Watch out for frequency scaling (power management) when reading the cycle counter.
- Pin the process to a core: `sched_setaffinity()`.
- Be aware of out-of order execution when reading CPU cycle counter.
- The cycle counter is accurate enough to measure each run separately.

Generating Random Links

You can generate the random links in Figure 1 (c) using a shuffled index array. First, create an array of length N initialized to `index[i]=i`. Then, generate a random permutation of the array, i.e., shuffle it, using *Knuth Shuffle* [2]:

```
/* Knuth Shuffle */
for(i=N-1; i>1; i--) {
  j = rand()%(i+1);    /* uniformly 0 <= j <= i */
  tmp = index[j];
  index[j] = index[i];
  index[i] = tmp;
}
```

Next, build the linked list structure corresponding to the shuffled array.

References

[1] Ulrich Drepper. What every programmer should know about memory. http://people.redhat.com/drepper/cpumemory.pdf, 2007.

[2] Richard Durstenfeld. Algorithm 235: Random permutation. *Commun. ACM*, 7(7):420, 1964.