Data Cache Parameter Measurements

Enyou Li, Clark Thomborson
Computer Science Department
University of Auckland
Private Bag 92019, Auckland
New Zealand
{ley,cthombor}@cs.auckland.ac.nz

Abstract

We extend prior research by Saavedra and Smith on designing microbenchmarks to measure data cache parameters. Unlike Saavedra and Smith, we measure the parameters by characterizing read accesses separately from write accesses; and we do not assume that the address mapping function is a bit-selection. We can measure the cache capacity C, block size b, and associativity a; we can measure the cache-hit access time and penalty for read and write; we can determine whether a cache allocates on write; we can detect write-back and write-through policies. We present experimental results for two CPU/cache structures, a 200 MHz Pentium with MMX and a 180 MHz Pentium Pro.

1. Introduction

Computer performance is, increasingly, limited by memory speed rather than processor speed, as graphically illustrated on page 619 of [8]. In 1996, Richard Sites opined that "over the coming decade memory subsystem design will be the only important design issue for microprocessors" [10]. In his paper, the CPUs could only run at less than 10% of their peak instruction issue rate.

Computer architects are not the only people who are concerned about memory systems performance. As memory bottlenecks increase in frequency and severity, performance programmers and algorithmic designers must pay more attention to memory design [5, 6]. For many computational problems, the performance of the primary data cache is a crucial consideration. Our analysis and experimental results on cache performance, presented in this paper, could be used to guide the appropriate selection, design, and hand-tuning of computer codes.

For an optimizing compiler using a fixed "cost function" or machine performance model [1], our microbenchmarks could be used to choose appropriate parameters for the cache portion of a machine performance model, guiding a compiler's optimizations of memory operations in the inner-loop computations that consume the vast majority of time in many applications.

The performance of the primary cache is of strong current interest to both the computer architects and programmer, because this cache forms an important "first layer" of the memory hierarchy in modern system design. In this paper, we show how to write a series of microbenchmarks that, in conjunction with our analysis, will estimate the capacity C, the associativity a, the blocksize b, and the cache hit access times and cache miss penalties. Our microbenchmarks are thus similar to (and indeed are inspired by) those proposed by Saavedra and Smith[9]. Our benchmarks, however, differentiate between read and write access times rather than reporting an average access time for a 50/50 mix of reads and writes, as is done by Saavedra and Smith. (Note: Smith and Von Worley have proposed microbenchmarks for parallel computers that separately estimate read and write access times [13]. These microbenchmarks do not estimate any other cache performance parameters.)

We provide two additional microbenchmarks that can be used to determine the presence or absence of the cache policies commonly known as allocate-on-write and write-through.

Our analysis indicate that our microbenchmarks will deliver correct results for a wide variety of cache architectures. In particular, and unlike Saavedra and Smith[9], we do not assume that a particular bit-selection function is used to compute the set-index for a cache reference and the cache capacity is not necessarily a power of two. We assume only that the set-index mapping function (along with the cache's associativity and replacement logic) will provide conflict-free access to any aligned array whose size does not exceed the total capacity of the cache. For example, our microbenchmarks are applicable to caching systems with "random" [11] or "EE-XOR" [7] set-indexing functions.

Smith did not discover any advantage in "random" over bit-selection functions [11], however a more recent study of a particular set-indexing function has shown great advantage on some codes, and no significant disadvantage [7]. It therefore seems prudent to design a microbenchmark that gives correct results even if the bit-selection function is not employed.

To date, we have tested the accuracy of our microbenchmarks on two workstations with different cache structures. Our experimental results, from a 180MHz Pentium Pro workstation and a 200MHz Pentium MMX workstation, are in agreement with the manufacturer's representations about their CPU chip's primary cache capacity, associativity, block size, and write policy.

2. Assumptions and notation

In order to estimate cache parameters, we must make some assumptions about the computer system of which the cache is a part. In particular, we make the following assumptions about the way in which a computer would execute a timed loop such as the one shown in Figure 1.

1. A timing routine (*e.g.* `clock()`) is available with sufficient accuracy to measure the wall-clock time for our microbenchmarks. In addition, any background tasks (*e.g.* those running on behalf of the operating system) will not introduce a systematic bias in our timing measurements.

2. Memory is byte-addressable and the memory access "strides" are measured in bytes. The starting address of an array such as `va[]` in Figure 1 will be aligned in memory. The length of a `long` data type is 4 bytes.

3. The compiler supports a `volatile` data type. When it optimize the program, it will not optimize any operation related to the `volatile` data. For example, the compiler will not remove the statements `s1 = va[j]` in Figure 1.

4. The compiler will place frequently-used scalar operands in data registers when optimization is enabled, and instructions will be cached separately from data, so that the only memory accesses in the inner loops of Figure 1 will be to the array `va[]`.

5. Sufficient CPU resources are available such that the average loop time `T4` calculated by the kernel of Figure 1 will reflect the average time required for a stride-4 read to memory and not, for example, the average time for an instruction fetch or an indexing calculation.

We must make additional assumptions about the structure of the cache itself.

```
volatile long *va;
register long s1;
double t1,T4;
*va=calloc(N, sizeof(long));
for(j=0; j<R; j++)
  s1=va[j]; /* preread loop */

start_time=clock();
for(l=0; l<LOOPN; l++)
  for(j=0; j<R;) {
    s1=va[j]; //16-unrolled
    s1=va[j+1];
    ... //13 similar lines
    s1=va[j+15]; j+=16;
  }
t1=(double)(clock()-start_time);
T4=t1/(R*LOOPN );
```

Figure 1. The timed kernel for a measurement of cache size, using stride-4 accesses.

1. The block (line) size b of the cache is a power of two in bytes, with $b \geq 4$. The block-offset function $B()$ is a bit-selection function, with $B(x) = x\&(b-1)$. (This is the typical mechanism by which a cache optimizes spatially-local accesses.)

2. The cache has associativity a. There are S sets in the cache and S is a power of two. So the total capacity C of the cache, measured in bytes, is given by $C = Sab$. Note: C and a are not necessarily powers of two.

3. The set-index mapping function $M()$ is not necessarily a bit-selection function. However all elements of any aligned array of size C can be cache-resident simultaneously, implying that the function $M()$ maps aligned address ranges of Sb bytes uniformly, with exactly one memory block of b bytes mapped to each of the S sets in the cache. So $M()$ may be expressed as the functional composition of an S-permutation $\pi_{x/Sb}$: $M(x) = \pi_{x/Sb}((x/b)\&(S-1))$. Note that the permutation $\pi_{x/Sb}$ may be the identity map, in which case the usual bit-selection function is employed for $M()$. Non-trivial $\pi_{x/Sb}$ have been proposed elsewhere [7].

4. A cache-miss on a read access will always cause the referenced block of data to be brought into the cache, displacing at most one "old" cache line according to either a LRU or a FIFO replacement algorithm. A cache-miss on a write access will cause the referenced block of data to be brought into the cache if and only if the primary cache has an "allocate on write" policy.

5. Though a cache hit on a read access will never issue a access to other deeper memory layers, a cache-hit on a write access to primary cache will cause a write access to a deeper layer in the memory hierarchy (*i.e.* to secondary cache, primary memory, or secondary memory) if and only if the primary cache has a "write-through" policy.

6. The time required by a cache miss on a read is at least 20% larger than the time required by a cache hit on a read. The time required for a cache miss on a write is at least 20% larger than the time required by a cache hit on a write, if the cache does not employ a "write-through" policy. If, instead, the cache employs a "write-through" policy, then the time required for a write access to primary cache is not significantly affected by whether the data is present in the primary cache.

7. The secondary cache can be disabled, if desired, to increase the accuracy of estimation of cache parameters other than the miss penalty. (Note: we have disabled secondary cache in our experiments on the Pentium and Pentium Pro workstations, described in this paper.)

In addition to the parameters and variables defined above, our microbenchmarks use the variables defined in item 1 below. We estimate the parameters defined in items 2 through 4.

1. A total of R distinct elements are referenced in an array holding a total of N elements, where each element is a 4-byte `long`. The value of N and R are different in each of our benchmark.

2. The average cache-read (resp. write) hit-access time for a particular access will be denoted t_r (resp. t_w), and is measured in nanoseconds. Note: our microbenchmark estimates of t_r and t_w are for cache bandwidth-bottlenecked computations such as the first loop of Figure 1. We write t'_r and t'_w for the somewhat larger, latency-influenced, estimates arising from the loop of Figure 5.

3. The average cache-read (resp. write) miss penalty is denoted by t_{pr} (resp. t_{pw}), measured in nanoseconds, for our bandwidth-bottlenecked loops. We write t'_{pr} and t'_{pw} for the somewhat larger, latency-influenced, estimates of miss penalty arising from the loop of Figure 5. Note: in our experiments on the Pentium and the Pentium Pro, we have disabled secondary cache, so our measured miss penalties are a reflection of access bandwidths and latencies to primary memory (DRAM), not of bandwidths or latencies to secondary cache.

4. The miss rate on reads (resp. writes) is M_r (resp. M_w), where $0 \leq M_r \leq 1$ and $0 \leq M_w \leq 1$. The average time for a read (resp. write) is $t_r + M_r t_{pr}$ (resp. $t_w + M_w t_{pw}$).

3. Measurement of cache parameters

In this section, we describe a series of microbenchmarks that will characterize cache performance parameters. Our first two microbenchmarks are similar to those described by Saavedra and Smith[9]. However our inner loops perform a memory read and not a memory read-write when we measure the cache capacity C and cache line size b; and our microbenchmark for measuring associtivity a uses a completely different methods.

3.1. Capacity

When an array is no larger than the cache, it will become cache-resident after the preread loop in Figure 1. Therefore, in the $t1$-timed loop in Figure 1, the miss rate M_r will be 0 if $4R \leq C$. This is "Regime 1" in the analysis of Saavedra and Smith [9].

For arrays that are too large to be cache-resident, there will be one cache miss for every $b/4$ inner loop iterations of the $t1$-timed loop. That is, the miss rate $M_r = 4/b$ if $4R \geq C + Sb$; this is "Regime 2a" in [9].

When $C < 4R < C + Sb$, the miss rate is $M_r = (4/b)\lceil(4R - C)/b\rceil/S$. This narrow interval is not named by Saavedra and Smith. We call it the "1–2a transitional regime."

A suitable microbenchmark for capacity is now easily described. Increase R in the loop of Figure 1 until the measured average stride-4 read time $T4(R)$ increases sharply from its regime-1 baseline of t_r, to reach its regime-2a plateau of $T4(R) = t_r + (4/b)t_{pr}$. In practice, we have to set `LOOPN` to a value sufficiently large for the resolution of the `clock()` routine.

The cache size is given by the largest R such that $T4(R) \approx t_r$ [9].

See Figure 2 for experimental results, averaged over five runs, on our two workstations. We can see that PC1's cache has a capacity C_1 (in bytes) that lies in the range $2^{14} < C_1 \leq 2^{15}$. Also, the capacity C_4 of our PC4 lies in the range $2^{13} < C_4 \leq 2^{14}$. These ranges could be narrowed, if desired, by testing $T4(R)$ for R is not powers of two.

Note that, in addition to estimating a capacity C, our stride-4 benchmark also estimates t_r and $(4/b)t_{pr}$.

3.2. Block size

If we vary the stride K of our array accesses, we can estimate the block size b in bytes [9]. A suitable inner loop

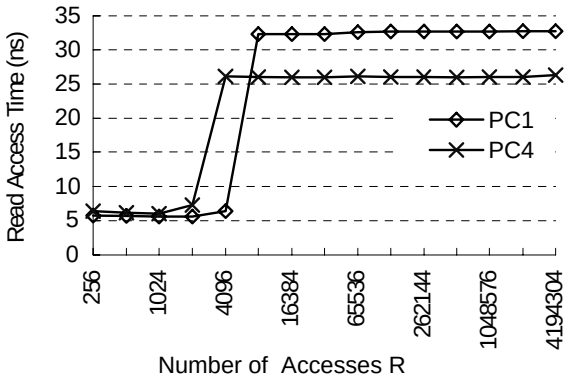

Figure 2. Read time for stride-4 accesses on PC1 and PC4.

```
volatile long vs1,*va;
double t2,TS;
int s=K/sizeof(long);
*va=calloc(N, sizeof(long));
start_time=clock();
for(l=0; l<LOOPN; l++)
  for(j=0; j<R;) {
    s1=va[j]; //16-unrolled
    s1=va[j+s];
    ... //13 similar lines
    s1=va[j+15*s]; j+=16*s;
  }
t2=(double)(clock()-start_time);
TS=t2/(R*LOOPN));
```

Figure 3. The kernel for a measurement of cache block size, using variable-stride reads.

is shown in Figure 3.

When $K \geq b$ and $4R \geq C + Sb$, the miss rate $M_r = 1$; this is Regime 2b of [9]. When $K < b$ and $4R \geq C + Sb$, the miss rate $M_r = K/b$; this is Regime 2a of [9]. We therefore set $R = 4C/4$ in this microbenchmark test, noting that $Sb \leq C$ for any cache. The reported average stride-K access time from the code of Figure 3 is thus $TS(K) = t_r + \min\{1, K/b\}t_{pr}$. The estimated blocksize b is numerically equal to the smallest value of K such that $TS(K) \approx TS(2K)$ [9]. Note that we have assumed that b is a power of two. So we only need to measure the $TS(K)$ with K a power of 2.

Experimental results for $TS(K)$, averaged over five runs, on our two workstations are shown in Figure 4. Note that, in both cases, $TS(16) < TS(32) \approx TS(64)$. We conclude that both primary caches have 32-byte blocks ($b = 32$).

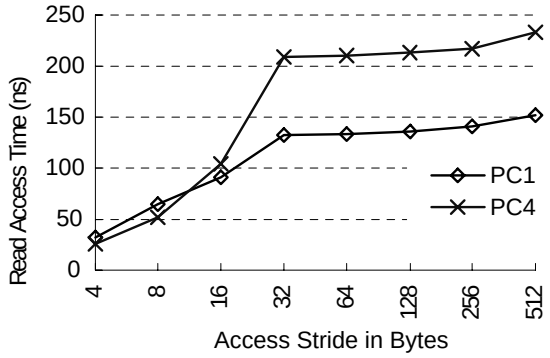

Figure 4. Read time for stride-K accesses on PC1 and PC4.

Because we know the `long` data type is 4 bytes, we can get the access times for read-hit and read-miss in the data cache from the results shown in Figure 2 now. For PC1, we get $t_r = 5.7$ ns and $t_{pr} = (32.2 - 5.7) \times (32/4) \approx 210$ ns. And for PC4, we get $t_r = 6.1$ ns and $t_{pr} = (26.0 - 6.1) \times (32/4) \approx 160$ ns.

We suspect there is some bias in our time measurements. We believe the read time for PC1 and PC4 should be exactly one clock (5.0 ns and 5.5 ns respectively). In future work, we hope to reduce this bias, or at least to explain it.

3.3. Associativity

Saavedra and Smith used a variable-stride benchmark, similar to our Figure 3, to estimate the associativity of a cache under the assumption that a bit-selection function is employed for set-indexing. This is equivalent to assuming that the function $\pi_{x/Sb}$ is the identity map, in our definition of the set-index function $M()$.

In our microbenchmark for estimating associativity, we make a measurement that is insensitive to $\pi_{x/Sb}$. In the code of Figure 5, we construct a (pseudo)random permutation [4] of the $4N/b$ array indices $\{0, b/4, 2b/4, ..., N - b/4\}$ of the 4-byte elements that would load at block offset zero in the primary cache. This construction requires a source `mrandom()` of pseudorandom variables that are uniformly and independently distributed in the range $\{0, 1, 2, ..., i-1\}$. Suitable C-language code for `mrandom()` is readily available [12].

In Figure 5, we use the first R elements of this permutation to define a length-R sequence of array elements to be accessed. Note that we access each element of `va[]` at most once, in each of `LOOPN` iterations of our timing loop. We call this mode of access "random without repetition" to distinguish it from the more usual benchmark in which elements are chosen for access "with repetition" from the index

```
va=calloc(N,sizeof(long));
/* Randomly Initialize the va[] */
for(j=0; j<N; j++)  va[j]=j;
for(i=4*N/b; i>1; i-=b/4 ) {
  j1=mrandom(i);
  /* j1 is u.i.d. in [0,i-1] */
  j=(j1\(b/4))*(b/4); temp=va[i-b/4];
  va[i-b/4]=va[j];      va[j]=temp;
}
for(j=0; j<R; j++)
  i=va[i]; /* preread loop */
start_time = clock();
for(l=0; l<LOOPN; l++) {
  i=va[0];
  for(j=0; j<R;) {
    i=va[i];
    i=va[i];
    ... //13 identical lines
    i=va[i]; j+=16;
  }
}
t3=(double)(clock()-start_time);
TR=t3/(R*LOOPN);
```

Figure 5. The kernel for a measurement of cache associativity, using random accesses without repetition.

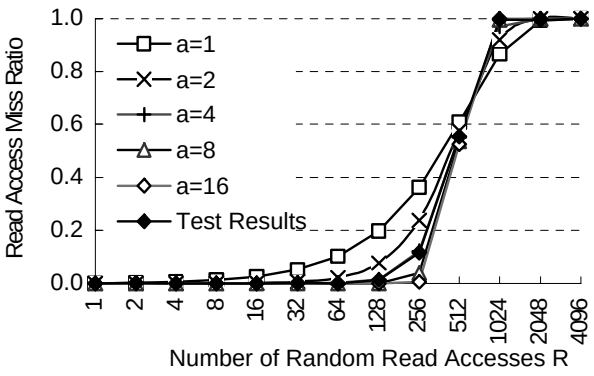

Figure 6. Read miss ratio for PC1.

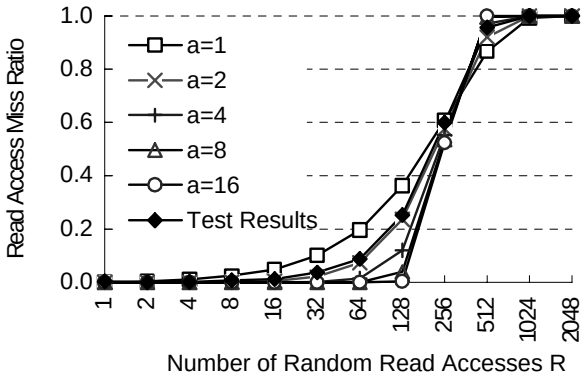

Figure 7. Read miss ratio for PC4.

space of an array in a uniform, independent fashion. We believe a benchmark based on random access with repetition would not yield as accurate an estimate of associativity, because the miss rate M_r in the analogue of our Theorem 2, below, would not be zero in the small-R case.

We are aware of no published analysis of caches being accessed randomly without repetition. However, the combinatorics are not very difficult, as indicated by the following theorems (presented here without proof).

Theorem 1 *If an urn contains Q marbles of which exactly Q/S are black, and we choose R marbles blindly, then the probability $P_{k,Q}$ of our choosing exactly k black marbles is*

$$P_{k,Q} = \binom{Q/S}{k}\binom{Q(1-1/S)}{R-k}\Big/\binom{Q}{R} \qquad (1)$$

Corollary 1 *If R b-byte elements are chosen randomly without repetition for reading from an aligned array of Q b-byte elements, where Q is a power of 2 greater than the number of sets S in the primary cache, then the probability of exactly k of these elements competing for cache lines that belong to a cache set is $P_{k,Q}$.*

Theorem 2 *If k distinct data elements $x_1, x_2, ..., x_k$ are read-accessed in sequence by a cache of associativity a, if this same sequence of read-accesses is repeated for a total of* `LOOPN` *times, and if any two of the k elements belong to different memory blocks and they are mapped to the same cache set, then the read-miss ratio M_r for the last* `LOOPN-1` *read sequences is*

$$M_r = \begin{cases} 0 & \text{if } k \le a \\ 1 & \text{otherwise} \end{cases} \qquad (2)$$

Corollary 2 *If R b-byte elements are chosen randomly without repetition for reading from an aligned array of Q b-byte elements, where $Q = 8C/b$, then the cache miss rate of* `t3` *-timed loop in the Figure 5 is*

$$E[M_r(R;a)] = \frac{S\sum_{k>a} kP_{k,Q}}{R} \qquad (3)$$

Proof. We have supposed that the cache mapping function uniformly maps the data blocks into different cache sets if the data array `va[]` is aligned to the cache size. So each cache set has the same probability $P_{k,Q}$ of being hit by k data accesses from the R random-selected data blocks

in the inner loop. Considering the former corollary, we can get the result in Equation 3.

We estimate the associativity of a cache by making the series of measurements TR(1), TR(2), TR(4), ..., TR($4N/b$). We estimate a miss ratio $M_r(R)$ from a measurement TR(R) as follows: $M_r(R) \approx (\text{TR}(R) - t'_r)/t'_{pr}$, where we take $t'_r = \min_R\{TR(R)\}$ and $t'_{pr} = \max_R\{TR(R)\} - t'_r$. We then find the a that gives a best-fit of our analytic form for $E[M_r(R;a)]$ (which is a function of a and R) to our series of estimates $M_r(R)$. This best-fit a is our estimate for the associativity of the cache.

Note: the average read-hit access time t'_r for this microbenchmark may be somewhat larger than the t_r estimated in our first microbenchmark.

Also, the read-miss penalty t'_{pr} for our associativity-testing microbenchmark is substantially larger than the t_{pr} estimated in our first microbenchmark. Because we have disabled the secondary cache, a random read-miss in this microbenchmark usually incurs the latency of a full (RAS/CAS) DRAM cycle, whereas the sequential read-miss of our first microbenchmark may proceed at the full bandwidth of a fast (CAS-only) DRAM cycle.

Our experimental results for associativity measurement, averaged on ten runs, on our two workstations are shown in Figures 6 and 7, for $N = 2^{15}$ and $N = 2^{14}$ respectively. We have a good fit to $a = 4$ for PC1 and to $a = 2$ for PC4, in agreement with the manufacturer's description [3].

4. Measurement of cache-write policies

We have written three microbenchmarks to determine cache-write policies and to estimate write times. Our first write microbenchmark is called the "pre-read kernel". It is shown in Figure 8. Based on our assumption that any cache system must allocate on read miss, We designed it to reveal whether a primary cache uses a write-through or write-back policy; and for write-back cache, we can also determine whether it allocate on write miss or not. Its name comes from the first part of the second loop, which reads `va[0..R-1]`. This read loop will establish cache residency of the last $\min\{C/4, R\}$ elements of `va[]`. Note: background tasks may slowly "pollute" our cache, displacing elements of `va[]`, so we add a read loop before several write loop in the second timed loop; and the first loop is designed to get the extra time used by the read accesses in the second loop.

If the cache uses a write-through policy, then the average write time reported by the pre-read microbenchmark is TW2(R) $\approx t_w$, that is, essentially invariant with R.

If the cache uses write-back policy, the average write time reported by the pre-read microbenchmark is noticeably dependent on R and whether the cache allocates on a write

```
register long s1,s2;
volatile long *va;
va=calloc(N, sizeof(long));
start_time=clock();
for(l=0; l<LOOPN; l++)
  for(j=0;j<R;) {
    s1=va[j]; //16-unrolled
    s1=va[j+1];
    ... //13 similar lines
    s1=va[j+15]; j+=16;
  }
t4=(double)(clock()-start_time);
start_time=clock();
for(l=0; l<LOOPN; l++) {
  for(j=0;j<R;) {
    s1=va[j]; //16-unrolled
    s1=va[j+1];
    ... //13 similar lines
    s1=va[j+15]; j+=16;
  }
  for(el=0; el<16; el++)
    for(j=0;j<R;) {
      va[j]=s2; //16-unrolled
      va[j+1]=s2;
      ... //13 similar lines
      va[j+15]=s2; j+=16;
    }
}
t5=(double)(clock()-start_time);
TW2=(t5-t4)/(R*LOOPN*16);
```

Figure 8. The "pre-read kernel" for determining cache writeback policy.

miss. Our reasoning is as follows: if $4R \leq C$, the array is cache-resident and there are no misses no matter the cache uses write-allocate or not.

If $4R \geq C$ and write-allocate, the miss ratio M_{w2} on this benchmark is:

$$M_{w2} = (4/b)\min\{1, \lceil(4R - C)/b\rceil/S\} \quad (4)$$

If $4R \geq C$ and not write-allocate, then $C/4$ of the R referenced elements are cache-resident, so the miss ratio is:

$$M_{w2} = 1 - C/4R \quad (5)$$

Equation 6 summarizes our analysis of the performance of a write-back cache on our pre-read benchmark.

$$\text{TW2}(R) \approx \begin{cases} t_w & \text{if } (4R \leq C) \wedge \text{WB} \\ t_w + M_{w2}t_{pw} & \text{if } (4R > C) \wedge \text{WB} \end{cases} \quad (6)$$

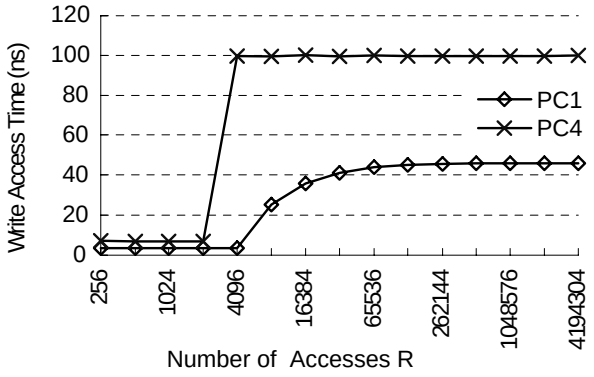

Figure 9. Average write time for PC1 and PC4, on our preread microbenchmark.

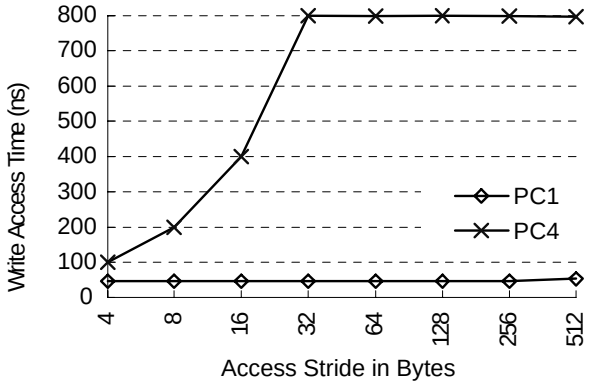

Figure 10. Average stride-K write time for PC1 and PC4.

So, for a write-through cache, we expect to see a write access time that does not change when we increase R. For a write-back and write-allocate cache, we expect a steep-rise from $R = C/4$ to $R = (C + Sb)/4$, and the write access time should be constant for $R \geq (C + Sb)/4$; for a write-back and not write-allocate cache, we expect a gradual rise for $R \geq C/4$.

Our experimental results for PC1 and PC4, averaged on five runs, on our pre-read benchmark are shown in Figure 9. The gradual rise, for PC1 after TW2(4096), indicates that it uses a write-back policy and does not employ write-allocate, and its capacity is 16 kilobytes. And the sharp rise, for PC4 between TW2(2048) and TW2(4096), indicates that it has a write-back, write-allocate primary cache with a capacity of 8 kilobytes.

In addition, we can got the access times for write-hit and write-miss from Figure 9. For PC1, $t_w = 3.5$ ns and $t_{pw} = 45.6 - 3.5 \approx 42$ ns, because the data cache of PC1 does not use the write-allocate policy. And for PC4, $t_w = 6.8$ ns and $t_{pw} = (99.6 - 6.8) \times (32/8) \approx 740$ ns, because its data cache uses a write-allocate policy.

We have written another microbenchmark for write accesses, to evaluate the dependence of write-access time on stride. We do not pre-read in this microbenchmark. Its kernel code is not shown here, however it is very similar to our strided-read microbenchmark in Figure 3. Under the same reason as for strided-read benchmark, we set $R = 4C/4$ in this microbenchmark. The reported average stride-K write access time is thus $TS(K) = t_r + \min\{1, K/b\}t_{pr}$ if the cache uses write-allocate and write-back policy; otherwise $TS(K)$ should not change with the variation of K.

Experimental results for our two workstations, averaged on five runs, on our strided-write microbenchmark are shown in Figure 10. It providing the write access time for different strides of PC4. The cache parameters implied by the graph, such as cache block size of PC4, non-write-allocate of PC1 etc., are consistent with that we got from our other microbenchmarks.

Table 1. Summary of experimental results with secondary cache disabled.

Machine	PC1	PC4
CPU	Pentium MMX	Pentium Pro
Clock	200 MHz	180 MHz
Capacity C	16 KB	8 KB
Block size b	32 B	32 B
Associativity a	4	2
Read:		
hit time t_r	5.7 ns	6.1 ns
miss pen. t_{pr}	210 ns	160 ns
Write:		
hit time t_w	3.5 ns	6.8 ns
miss pen. t_{pw}	42 ns	740 ns
Allocate on write?	no	yes
Write-through?	no	no

5. Summary and Future Work

We have described a series of microbenchmarks that measure the most important performance parameters of a primary cache. Our microbenchmarks are similar in spirit to those of Saavedra and Smith [9]. However our microbenchmarks characterize read and write times separately, they determine cache write policies, and they will work on caches with set-index functions other than the bit-selection function that is commonplace today.

Our experimental results, obtained by running our microbenchmarks on two workstations, are summarized in Table 1. Our measured capacity, blocksize and associativity

are in agreement with the manufacturer's description [3], and our access times seem plausible for systems in which the secondary cache is disabled.

In future work, we would like to analyze the performance of skewed caches [2] under our microbenchmarks. Such caches employ distinct S-permutations $\pi_{x/Sb,i}$ to define the set-index function for each bank i, where $1 \leq i \leq a$.

The results in this paper are obtained by disabling the secondary cache, enabling the compiler's optimization function for maximum speed to maximize the measurement accuracy. One reason our microbenchmarks are successful is that the instruction access and executions of these kernel loops are not bottlenecks on modern CPUs. In the future, we will examine the accuracy of our microbenchmarks with the secondary cache enabled. We are also interested in developing latency-bound microbenchmarks.

We intend to release our microbenchmarks for public distribution soon. Finally, we hope to find time, and funding, for porting our microbenchmarks to Unix or Linux.

References

[1] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[2] F. Bodin and A. Seznec. Skewed associativity improves program performance and enhances predictability. *IEEE Trans. on Computers*, 46(5):530–544, May 1997.

[3] Intel Corporation, `http://developer.intel.com/design/product.htm`

[4] D. E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley Publishing Company, 3rd edition, 1998.

[5] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Apr. 1991.

[6] A. R. Lebeck and D. A. Wood. Cache profiling and the SPEC benchmarks: A case study. *IEEE Computer*, pages 15–26, October 1994.

[7] E. Li. *Study on the Storage Schemes in High-Performance Computer Systems.* PhD thesis, Institute of Computer Technology, Chinese Academy of Sciences, 1997.

[8] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design: The Hardware / Software Interface*. Morgan Kaufmann, second edition, 1998.

[9] R. H. Saavedra and A. J. Smith. Measuring cache and TLB performance and their effect on benchmark runtimes. *IEEE Trans. Computers*, 44(10):1223–1235, October 1995.

[10] R. Sites. It's the memory, stupid. *Microprocessor Report*, 10(10):19, August 1996.

[11] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Trans. on Software Engineering*, SE-4(2):121–1130, March 1978.

[12] C. Thomborson. Tools for randomized experimentation. In M. E. Tarter and M. D. Lock, editors, *Statistical Applications of Expanding Computer Capabilities: Proceedings of the 25th Symposium on the Interface*, Computing Science and Statistics (Volume 25), pages 412–416. Interface Foundation of North America, 1993. Code available at `http://www.cs.auckland.ac.nz/~cthombor/Mrandom`

[13] S. J. Von Worley and A. J. Smith. Microbenchmarking and performance prediction for parallel computers. Technical Report UCB/CSD-95-873, Computer Science Division (EECS), University of California at Berkeley, May 1995.