Data Cache Parameter Measurements

Li Enyou, 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 cache performance. Unlike Saavedra and Smith, we characterise read accesses separately from write accesses; we determine whether a cache allocates on write; we can detect write-back and write-through policies; and we do not assume that the address mapping function is a bit-selection. 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 [7]. In 1996, Richard Sites opined that "over the coming decade memory subsystem design will be the only important design issue for microprocessors" [9]. In support of his opinion, Sites described the results of a database study using the TPC-C benchmark, in which both a 200 MHz Pentium Pro and a 400 MHz 21164 Alpha systems were measured at 4.2 to 4.5 CPU cycles per instruction retired. Most of the CPU cycles in these systems were spent waiting for memory. These CPUs thus ran 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 problems involving small datasets, the performance of the primary data cache is a crucial consideration. Our analyses 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.

Hand-tuning is necessary, at present, to achieve optimal performance on highly-parallel architectures. However hand-tuning is expensive, and the resulting executables are not very portable. For these reasons, most executables are prepared with little or no hand-tuning, by an optimising 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 optimisations 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 computer architects, because this cache forms an important "first layer" of the memory hierarchy in modern system design. In particular, this cache should provide extremely rapid access to the most recently used ("temporally local") data, as well as to nearby ("spatially local") data. Our microbenchmarks and analyses give insight into the performance and operation of such caches. For example, we show how cache associativity affects random-access time into variously-sized datasets.

In this paper, we show how to write a series of microbenchmarks that, in conjunction with our analyses, will estimate the capacity C, the associativity a, the blocksize b, and the time required for cache hits for both read and write accesses. Our microbenchmarks are thus similar to (and indeed are inspired by) those proposed by Saavedra and Smith[8]. 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. In our experiments, we find that read-miss time can be quite dissimilar to write-miss time, and we understand that the ratio of reads to writes in typical codes is 2:1 or more [11]. (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 analyses indicate that our microbenchmarks will deliver correct results for a wide variety of cache architectures. In particular, and unlike Saavedra and Smith[8], we do not assume that a particular bit-selection function is used to compute the set-index for a cache reference. 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" [10] or "EE-XOR" [3] set-indexing functions. Smith did not discover any advantage in "random" over bit-selection functions [10], however a more recent study of a particular set-indexing function has shown great advantage on some codes, and no significant disadvantage [3]. 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. One workstation has, as its CPU chip, a 180 MHz Pentium Pro; the other workstation has a 200 MHz Pentium with MMX. Our experimental results 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 suitable timing routine (*e.g.* `clock()`) is available. This routine will report wall-clock time accurately, to within its documented precision, on a system that is running no other computational tasks in foreground.

2. Any background tasks (*e.g.* those running on behalf of the operating system) will not introduce a systematic bias in our timing measurements, for example, by synchronising background task activity with our calls to the timing routine.

3. Memory is byte-addressible.

4. Memory "strides" are measured in bytes.

5. The length of a `double` is 8 bytes.

6. The length of an `int` is $w = 4$ bytes. (We will adjust this assumption when we port our code to other platforms.)

7. The compiler will not replace loops, such as those in Figure 1, with straight-line code such as `z += LOOPN * R`.

8. The compiler will place frequently-used scalar operands in data registers, 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 `x[]`.

9. Sufficient CPU resources are available such that the average loop time `T8` calculated by the kernel of Figure 1 will reflect the average time required for a stride-8 read to memory and not, for example, the average time for an instruction fetch or an indexing calculation.

10. The starting address of an array such as `x[]` in Figure 1 will be aligned in memory. That is, the $\lceil y \rceil$ least-significant bits of the address `&x[0]` will all be zero, where y is the base-2 logarithm of the size of `x[]` in bytes.

```
double *x = calloc(N, sizeof (double));
double stride8( double z, int R,
                int LOOPN, double *T8 ) {
  for( i=0; i<LOOPN; i++ ) {
    if( i==1 ) start_time = clock();
    for( j=0; j<R; j++ ) {
      z += x[j]; /* stride-8 read */
    }
  }
  *T8 = (double)(start_time - clock());
  for( i=0; i<LOOPN; i++ ) {
    if( i==1 ) start_time = clock();
    for( j=0; j<R; j++ ) {
      z += 1.0; /* timing baseline */
    }
  }
  *T8 -= (double) (start_time-clock());
  *T8 /= ( (double) CLOCKS_PER_SEC * R
          * (LOOPN - 1) );
  return z; /* ensure z is computed */
}
```

Figure 1. The timed kernel for a measurement of cache size, using stride-8 accesses.

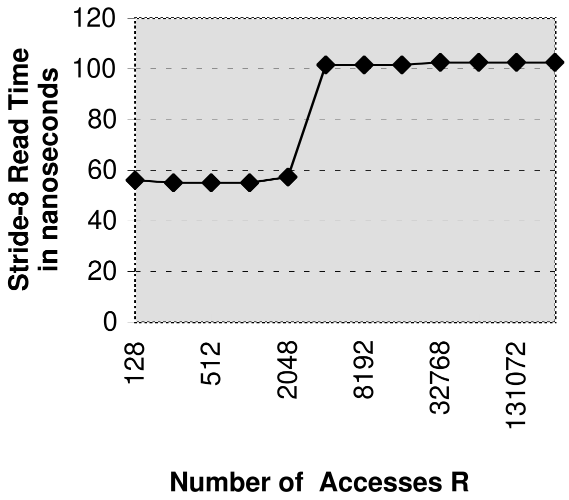

Figure 2. Read time for stride-8 accesses on PC1.

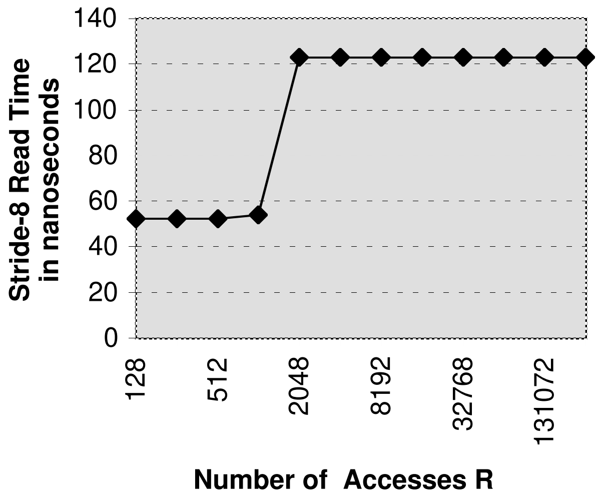

Figure 3. Read time for stride-8 accesses on PC4.

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

1. The block (line) size b of the cache is a power of two, with $b \geq 8$. We measure b in bytes.

2. The block-offset function $B()$ is a bit-selection function, sensitive only to the least significant bits of the address x. That is, we have $B(x) = x\&(b-1)$, where $\&$ is the bitwise "and" function. (This is the typical mechanism by which a cache optimises spatially-local accesses.)

3. The cache has associativity a, where a is (of course!) a positive integer.

4. There are L sets in the cache, with $L \geq 1$; each set consists of a blocks of b bytes. We assume that L is a power of two.

5. The total capacity C of the cache, measured in bytes, is given by $C = Lab$. Note that this implies $C \geq 8$. Also note that L divides C, however C is not necessarily a power of two.

6. 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 Lb bytes uniformly, with exactly one address block of b bytes mapped to each of the L sets in the cache. So $M()$ may be expressed as the functional composition of an L-permutation $\pi_{x/Lb}$ with the usual bit-selection function: $M(x) = \pi_{x/Lb}((x/b)\&(L-1))$. Note that the permutation $\pi_{x/Lb}$ may be the identity map, in which case the usual bit-selection function is employed for $M()$. Examples of non-trivial $\pi_{x/Lb}$ have been proposed elsewhere [3].

7. A cache-miss on a read access will cause the referenced block of data to be brought into the cache, displacing at most one "old" block of data according to either a LRU or a FIFO replacement algorithm.

8. A cache-miss on a write access will cause the referenced block of data to be brought into the cache only if the primary cache has an "allocate on write" policy.

9. A cache-hit on a write access to primary cache causes a write access to a deeper layer in the memory hierarchy (*i.e.* to secondary cache, primary memory, or secondary memory) only if the primary cache has a "write-through" policy.

```
int s = S / sizeof (double);
for( i=0; i<LOOPN; i++ ) {
  if( i==1 ) start_time = clock();
  for( j=0; j<R*s; j+=s ) {
    z += x[j]; /* stride-s read */
  }
}
*TS = (double)(start_time - clock());
*TS /= (double) (R * (LOOPN - 1));
```

Figure 4. The kernel for a measurement of cache block size, using variable-stride reads.

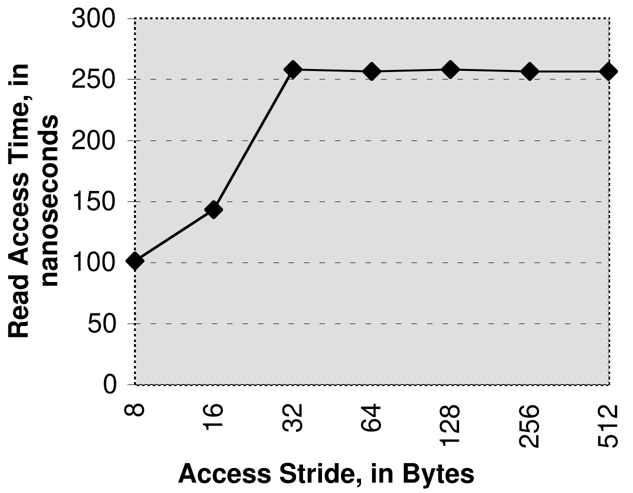

Figure 5. Read time for strided accesses on PC1.

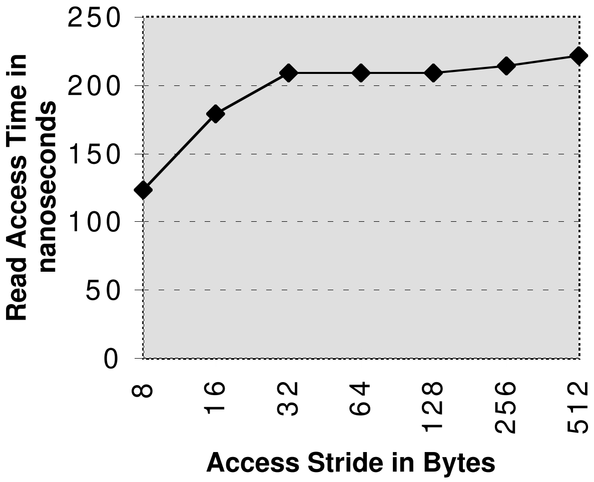

Figure 6. Read time for strided accesses on PC4.

10. The time required by a cache miss, on a read, for a loop such as the first one shown in Figure 1, is at least 20% larger than the time required by a cache hit on a read.

11. The time required for a cache access on a write, for a loop such as the last one shown in Figure 14, 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 does employ 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.

12. 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 PentiumPro processors, 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 5.

1. A total of R distinct elements are referenced in an array holding a total of of N elements, where each element is an 8-byte `double`. In all cases N will be a power of two.

2. The average cache-read (resp. write) hit-access time for a particular loop 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 8.

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 8. Note: in our experiments on the Pentium and the PentiumPro, 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 $0 \leq M_w \leq 1$.

5. 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-read parameters

In this section, we describe a series of microbenchmarks that will characterise cache performance parameters under read-only accesses. Our first two microbenchmarks are similar to the ones described by Saavedra and Smith[8], however our inner loops perform a memory read and not a memory read-write.

3.1. Capacity

When an array is no larger than the cache, it will become cache-resident after its first reading, under our assumptions. Therefore, in loop iterations $i > 0$ of the first loop nest in Figure 1, the miss rate M_r will be 0 if $8R \leq C$. This is "Regime 1" in the analysis of Saavedra and Smith [8].

For arrays that are too large to be cache-resident, there will be one cache miss for every $b/8$ inner loop iterations. That is, the miss rate $M_r = 8/b$ if $8R \geq C + Lb$; this is "Regime 2a" in [8].

When $C < 8R < C + Lb$, the miss rate is $M_r = (8/b)\lceil(8R - C)/b\rceil/L$. 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 until the measured average stride-8 readtime $T8(R)$ increases sharply from its regime-1 baseline of t_r, to reach its regime-2a plateau of $T8(R) = t_r + (8/b)t_{pr}$.

We set `LOOPN` to a value sufficiently large that $(\texttt{LOOPN})(8/b)t_{pr}$ is at least a few ticks of our `clock()` measurement. By assumption, $t_{pr} \geq (20\%)t_r$, so we can choose an appropriately-large `LOOPN` after making an initial estimate of t_r by measuring $T8(8)$.

The cache size is given by the largest R such that $T8(R) \approx t_r$ [8].

See Figures 2 and 3 for experimental results on our two workstations. From the data in these figures, 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 capacities other than powers of two.

Note that, in addition to estimating a capacity C, our stride-8 benchmark also estimates t_r and the quotient t_{pr}/b.

3.2. Block size

If we vary the stride S of our array accesses, we can estimate the block size b in bytes [8]. A suitable inner loop is shown in Figure 4.

When $S \geq b$ and $8R \geq C + Lb$, the miss rate $M_r = 1$; this is Regime 2b of [8]. When $S < b$ and $8R \geq C + Lb$, the miss rate $M_r = S/b$; this is Regime 2a of [8]. We therefore set $R = C/4$ in this microbenchmark test, noting that $Lb \leq C$ for any cache. The reported average stride-S access time from the code of Figure 4 is thus $TS(S) = t_r + \min\{1, S/b\}t_{pr}$. The estimated blocksize b is numerically equal to the smallest value of S such that $TS(S) \approx TS(2S)$ [8]. Note that we have assumed that all blocksizes are a power of two.

We can also estimate t_{pr} as being equal to $TS(b) - t_r$. Recall that t_r was estimated in our capacity microbenchmark, described earlier.

Experimental results from running $TS(S)$ on our two workstations are shown in Figures 5 and 6. Note that, in both cases, $TS(16) < TS(32) \approx TS(64)$. We conclude that both primary caches have 32-byte blocks ($b = 32$). We can also conclude that PC4 uses sub-block allocation, with a sub-block length of at most 16 bytes, because its stride-8 reads are relatively efficient in comparison to stride-16 reads. By contrast, PC1 takes only marginally more time per stride-16 read than in a stride-8 read, with the difference perhaps reflecting the greater number of cache-tag updates per cache-data memory read in the former case.

```
/* Knuth Vol. 2 (1981) p. 139 */
int pra[ 8*N/b ];
#define swapd(a,b) { \
  double t; t=(a); a=(b); b=t; \
}
for( i=0; i<8*N/b; i++ ) {
  pra[ i ] = i*b;
}
for( i=8*N/b; i>0; i-- ) {
  int j = mrandom( i );
  /* j is u.i.d. in [0,i-1] */
  swapd( pra[ i-1 ], pra[ j ] );
}
```

Figure 7. Constructing a random permutation in O(N/b) time.

```
for( i=0; i<LOOPN; i++ ) {
  if( i==1 ) start_time = clock();
  for( j=0; j<R; j++ ) {
    z += x[ pra[ j ] ];
  }
}
*TR += (double)(start_time - clock());
*TR /= (double)(2 * R * (LOOPN - 1));
```

Figure 8. The kernel for a measurement of cache associativity, using random accesses without repetition.

3.3. Associativity

Saavedra and Smith used a variable-stride benchmark, similar to our Figure 4, to estimate the associativity of a

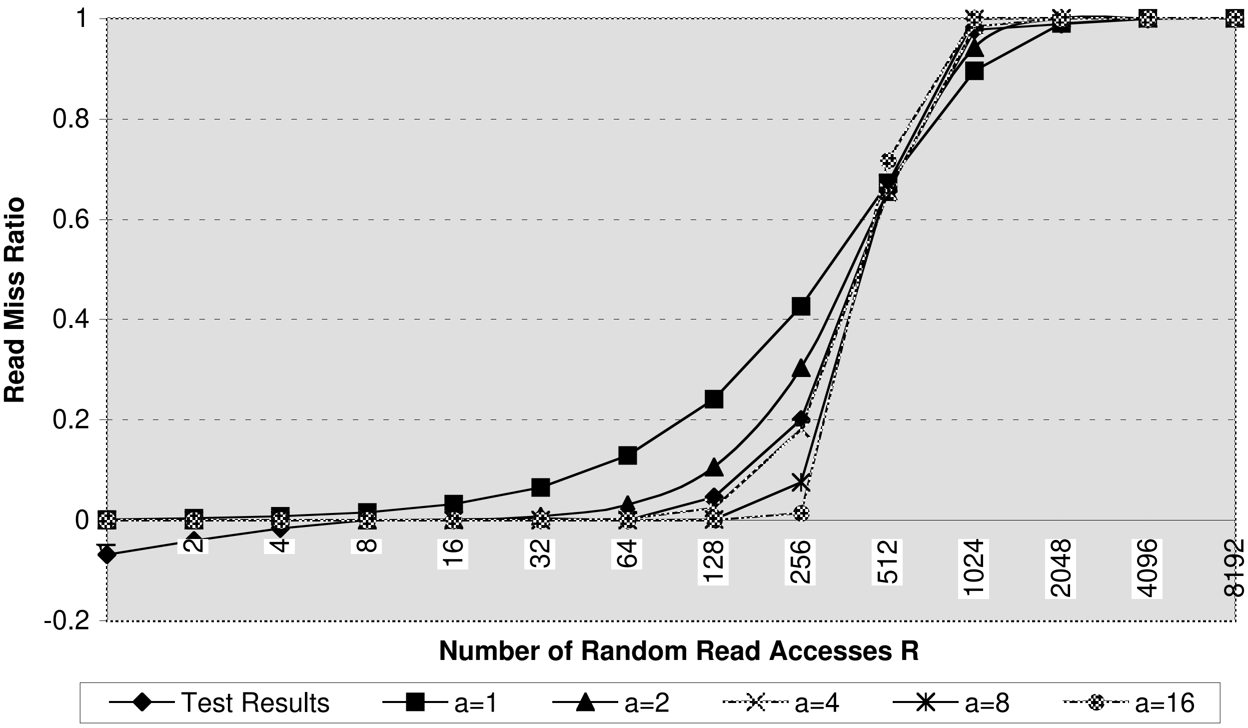

Figure 9. Read miss ratio for PC1.

cache under the assumption that a bit-selection function is employed for set-indexing. This is equivalent to assuming that the function $\pi_{x/Lb}$ is the identity map, in our notation for the set-index function $M()$.

In our microbenchmark for estimating associativity, we make a measurement that is insensitive to $\pi_{x/Lb}$. In the code of Figure 7, we construct a (pseudo)random permutation of the $8N/b$ array indices $\{0, b/8, 2b/8, ..., N - b/8\}$ of the 8-byte elements that would load at block offset zero in the primary cache. This construction requires a source `mrandom()` of pseudorandom variates 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 8, 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 `x[]` 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 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-k case.

We are aware of no published analyses 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/L are black, and we choose R marbles blindly, then the probability $P_{k,Q}$ of our choosing exactly k black marbles is*

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

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 L in the primary cache, then the probability of exactly k of these elements competing for a single line in the primary cache 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 all k of these elements are competing for block-offset zero in a single cache line, then the read-miss ratio M_r for the last* `LOOPN - 1` *read sequences is*

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

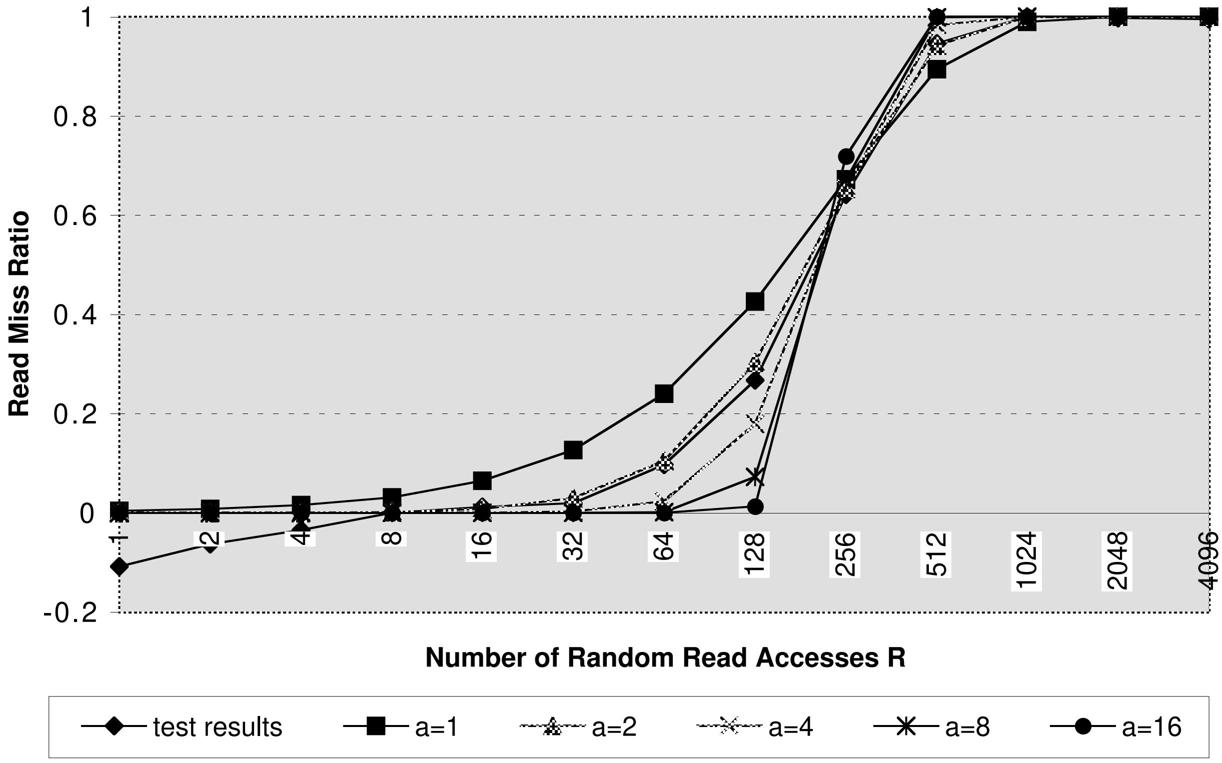

Figure 10. Read miss ratio for PC4.

Corollary 2 *If $Rw \geq C + Lb$, the expected read-miss ratio $E[M_r]$ of the timed portion of the program of Figure 8 is*

$$E[M_r] = \frac{Rw/b + L\sum_k kP_{k,8N/b}}{2R} \tag{3}$$

Proof. We must consider the R sequential accesses to the array `pra[]` in addition to the R random accesses to the array `x[]`. The total size of the accessed region of `pra[]` is Rw bytes. Because $Rw \geq C + Lb = (a+1)Lb$, the accesses to `pra[]` will touch all L sets in the cache at least $(a+1)$ times. The cache's LRU or FIFO replacement policy will cause one read miss fo each access to the first element in each block of w/b elements of `pra[]`, for a total of Rw/b read misses on the sequential accesses. To this term in the numerator we add the expected number of misses on any cache set that is referenced by $k \geq 1$ of our random accesses. Every one of these accesses is a miss, because the cache is effectively flushed of `x[]` elements on each iteration, by the sequential accesses to `pra[]`. There are L sets to consider, each contributes $kP_{k,8N/b}$ misses. The miss ratio is the total number of misses divided by the total number of references $2R$.

Corollary 3 *If $Rw \leq Lb$, then*

$$E[M_r] = \Big((Rw/b)\sum_{k \geq a}(k+1)P_{k,8N/b} + (L - Rw/b)\sum_{k>a} kP_{k,8N/b}\Big)/2R \tag{4}$$

Proof. The referenced portion of the array `pra[]` occupies only one block in only Rw/b sets in the cache. Elements of the array `pra[]` will thus only be displaced from the cache in lines that are hit by $k \geq a$ random accesses. In such lines, there are $k+1$ misses per iteration: one for each random access, plus one to reload the `pra[]` block. In the remaining $L - Rw/b$ lines of the cache that do not hold elements of `pra[]`, there are k misses per iteration when $k > a$.

Corollary 4 *If $Lb \leq Rw \leq C$ then*

$$E[M_r] \approx \frac{L\sum_{k \geq a - Rw/bL}(Rw/bL + k)P_{k,8N/b}}{2R} \tag{5}$$

Proof. The elements of `pra[]` occupy approximately $Rw/bL \leq a$ blocks of each of the L sets in the cache. Cache sets that are accessed by $k < a - Rw/bL$ of our random references will thus contribute no misses. Cache sets that are accessed more frequently will contribute approximately $Rw/bL + k$ misses.

Corollary 5 *If $C \leq Rw \leq C + Lb$, then $E[M_r]$ is the linear interpolation, in R, between Equations 3 and 5.*

```
double y* = calloc( C/8, 8);
for( i=0; i<C/8; i++ ) {
  z += y[i]; /* y[] fills cache */
}
for( i=0; i<R; i++ ) {
  x[i] = 0; /* write-allocate? */
}
start_time = clock();
for( i=0; i<LOOPN; i++ ) {
  for( j=0; j<R; j++ ) {
    x[j] = z; /* write-miss? */
  }
}
*TW1 += (double)(start_time -clock());
*TW1 /= (double)(R * (LOOPN -1));
```

Figure 11. The "pre-write kernel" for determining whether cache blocks are allocated on a write-miss.

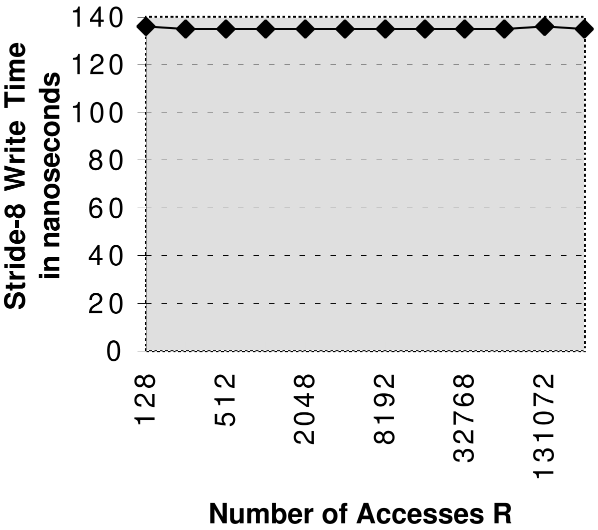

Figure 12. Average write time for PC1, on our prewrite microbenchmark.

We estimate the associativity of a cache by making the series of measurements TR(1), TR(2), TR(4), ..., TR($8N/b$). 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 $t'_r = \text{TR}(8)$ and $t'_{pr} = \text{TR}(8N/b) - t'_r$. We then find the a that gives a best-fit of our analytic form for $E[M_r]$ (which is a function in 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. Other operations such as loop index updates will be overlapped (in typical optimised code) with the read accesses of our first microbenchmark, and multiple array-read accesses may be in progress simultaneously. In the current microbenchmark, however, the `j`-th random-read to `x[]` cannot proceed until the `j`-th sequential-read to `pra[]` has delivered a result. We believe that a typical optimising compiler will *not* unroll and reschedule the loop to avoid a latency bottleneck on this dependency in our associativity-testing microbenchmark, so we would not be surprised to find that $t'_r > t_r$ in our experimentation.

Also, the read-miss penalty t'_{pr} for our associativity-testing microbenchmark is substantially larger than the t_{pr} estimated in our first microbenchmark, for the following reason. Recall that 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 testing on our two workstations are shown in Figures 9 and 10, for $N = 2^{15}$. We have a good fit to $a = 4$ for PC1 and to $a = 2$ for PC4, in agreement with the manufacturer's description [4].

4. Measurement of cache-write policies

We have written three microbenchmarks to determine cache-write policies and to estimate write times. Our first cache-write microbenchmark, shown in Figure 11, is designed to determine whether a write-allocation policy is employed. The third of the three loops in this microbenchmark is a timed series of stride-8 writes into an array that may, or may not, be cache-resident. The previous two loops will establish cache residency for the array `x[]` if and only if it is small enough to fit in the cache, and if the cache has a write-allocation policy.

We call this microbenchmark the "pre-write kernel," because of the write-loop on `x[]` that precedes the timed series of array writes.

The time per write TW1 reported by the pre-write kernel of Figure 11 will be $\text{TW1}(R) \approx t_w + t_{wp}$, that is a constant for all R, if the cache does not allocate on write. If, instead, the cache *does* allocate on write, then $\text{TW1}(R)$ will be dependent on R, rising noticeably when the referenced portion of the array is not cache resident. By our assumptions about cache behaviour, $\text{TW1}(R) \approx t_w$ when $8R \leq C$. When $8R \geq C + Lb$, then $\text{TW1}(R) \approx t_w + (8/b)t_{wp}$: every array block-reference will be a miss, as we have assumed FIFO or LRU replacement of cache lines.

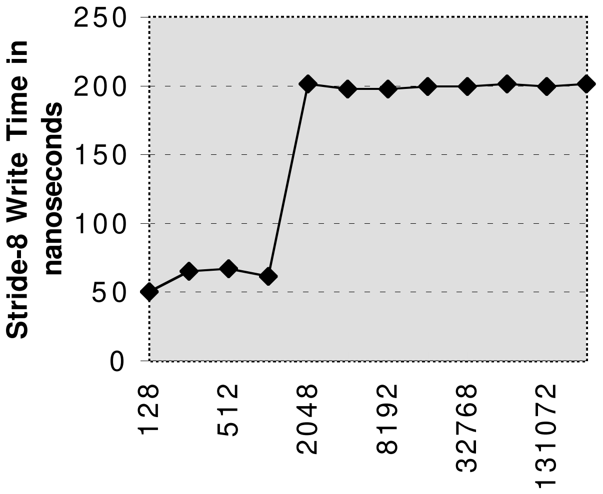

Figure 13. Average write time for PC4, on our prewrite microbenchmark.

```
for( i=0; i<R; i++ ) {
  z += x[i]; /* read-allocate x[] */
}
start_time = clock();
for( i=0; i<LOOPN; i++ ) {
  for( j=0; j<R; j++ ) {
    x[j] = z; /* write-miss? */
  }
}
*TW2 += (double)(start_time -clock());
*TW2 /= (double)(R * LOOPN);
```

Figure 14. The "pre-read kernel" for determining cache writeback policy.

Our experimental results for TW1(R) on two workstations are shown in Figures 12 and 13. We conclude that PC1 does not allocate on write, as its average write time is essentially constant. The average write time for PC4, on the other hand, changes markedly between $R = 1024$ and $R = 2048$, indicative of a write-allocate cache with a capacity of 8 kilobytes.

The "pre-read kernel" of our second microbenchmark for writes is shown in Figure 14. This microbenchmark is designed to reveal whether a primary cache uses a write-through policy. Our name for this kernel is taken from the first of its two loops, which reads `x[0..R-1]`. This read will establish cache residency for the last min$\{C/8, R\}$ elements read from `x[]`.

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.

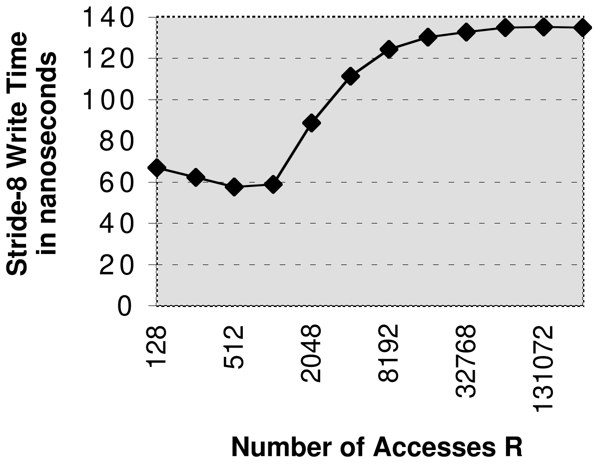

Figure 15. Average write time for PC1, on our preread microbenchmark.

If the cache does not use a write-through policy, it is normally called a write-back cache. For write-back caches, the average write time reported by the pre-read microbenchmark is noticeably dependent on R. Our reasoning is as follows: if $8R \le C$, the array is cache-resident and there are no misses. If $8R \ge C$, in the case of a write-allocate cache, the miss ratio M_{w2} on this benchmark is

$$M_{w2} = (8/b) \min\{1, \lceil (8R - C)/b \rceil / L\} \qquad (6)$$

If $8R \ge C$, and if writes don't allocate, then at most $C/8$ of the R referenced elements are cache-resident, so the miss ratio $M_{w2} \ge 1 - C/8R$. (Note: background tasks may slowly "pollute" our cache, displacing elements of `x[]`, so we know only a lower bound on the miss ratio in this case.)

With M_{w2} as defined in the previous paragraph, Equation 7 summarises our analysis of the performance of a write-back cache on our pre-read benchmark.

$$\mathrm{TW2}(R) \approx \begin{cases} t_w & \text{if } (8R \le C) \wedge \neg\mathrm{WT} \\ t_w + M_{w2} t_{wp} & \text{if } (8R > C) \wedge \neg\mathrm{WT} \end{cases} \qquad (7)$$

Our experimental results for PC1 and PC4 on our pre-read benchmark are shown in Figures 15 and 16. We had already determined that PC1 does not employ write allocation, so we expect to see a fairly gradual rise in TW2(R) for $R > 1024$. This rise indicates that PC1 does *not* employ write-through. The cache policies of write-allocate, and not write-through (commonly termed "write-back"), are clearly present in PC4, indicated by the step-function rise in TW2(R) near $R = 1024$.

We have written a third microbenchmark for write accesses, to evaluate the dependence of write-access time on

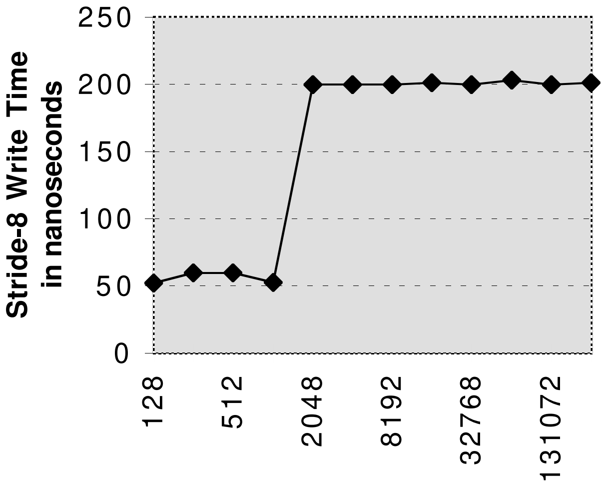

Figure 16. Average write time for PC4, on our preread microbenchmark.

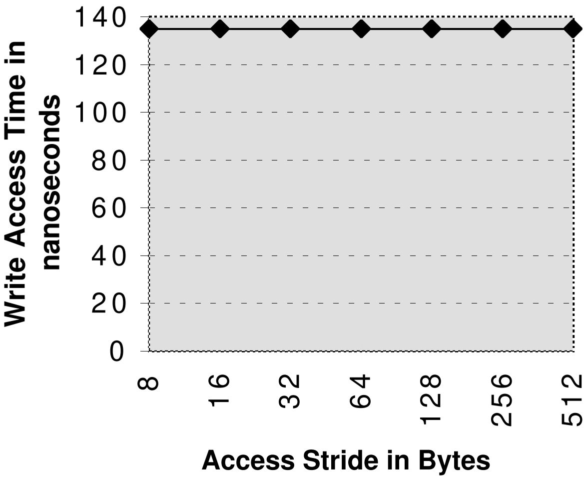

Figure 17. Average stride-S write time for PC1.

stride. We do not pre-read or pre-write in this microbenchmark. Its kernel code is not shown here, however it is very similar to our strided-read microbenchmark Figure 4. We can estimate t_{wp} from the results of this third microbenchmark, given an estimate of t_w from the small-R results from our pre-read microbenchmark.

Experimental results for our two workstations, on our strided-write microbenchmark are in Figures 17 and 18. We note that PC4's stride-8 writes are not particularly efficient in comparison to its stride-16 writes, indicating that an entire cache block (and not just a sub-block) is allocated on a cache write.

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 [8]. However our microbenchmarks characterise 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 summarised in Table 1. Our measured capacity, blocksize and associativity are in agreement with the manufacturer's description [4], and our access times seem plausible for systems in which the secondary cache is disabled.

In future work, we would like to analyse the performance of skewed caches [2] under our microbenchmarks. Such caches employ distinct L-permutations $\pi_{x/LB,i}$ to define the set-index function for each bank i, where $1 \leq i \leq a$. We believe that skewed caches will not establish complete residency for aligned subarrays of size $R = C/8$ after a single read. We also believe that skewed caches will, after many iterations of our random sequence without repetition, behave essentially as though they have very high associativity (approaching 2^a).

We believe it possible, at least in principle, to use a single kernel `i = p[i]`, with variable permutations `p[]`, in all our microbenchmarks that measure read accesses. Such a kernel would be bottlenecked on memory latency, as one iteration cannot start until the prior iteration's read is complete. We are also interested in similar, latency-bound, microbenchmarks for writes.

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

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

[5] M. S. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. In

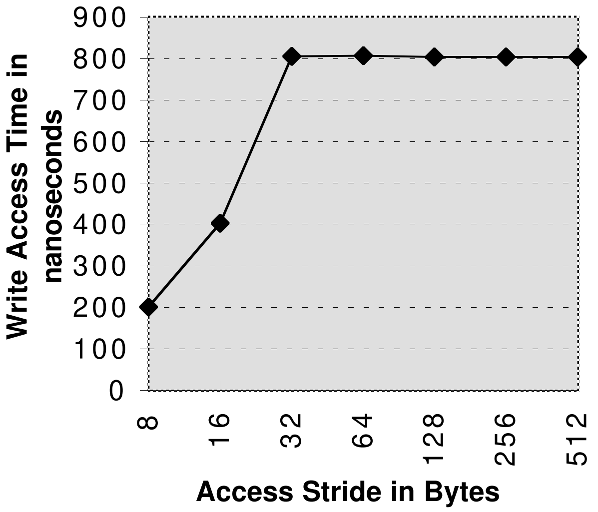

Figure 18. Average stride-S write time for PC4.

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

[8] 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.

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

[10] 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.

[11] A. J. Smith. Cache memories. *Computer Surveys*, 14(3):473–530, September 1982.

[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.

Name	PC1	PC4
CPU	Pentium w/ MMX	PentiumPro
Speed	200 MHz	180 MHz
Capacity C	16 KB	8 KB
Line size b	32 B	32 B
Associativity a	4	2
Read:		
hit time t_r	55 ns	52 ns
miss pen. t_{rp}	202 ns	159 ns
Write:		
hit time t_w	57 ns	52 ns
miss pen. t_{wp}	78 ns	751 ns
Allocate on write?	no	yes
Write-through?	no	no

Table 1. Summary of experimental results for primary caches of Pentium and PentiumPro systems, with secondary cache disabled.