Automatic Measurement of Memory Hierarchy Parameters[*]

Kamen Yotov, Keshav Pingali, Paul Stodghill
{kyotov,pingali,stodghil}@cs.cornell.edu
Department of Computer Science,
Cornell University,
Ithaca, NY 14853.

ABSTRACT

On modern computers, the running time of many applications is dominated by the cost of memory operations. To optimize such applications for a given platform, it is necessary to have a detailed knowledge of the memory hierarchy parameters of that platform. In practice, this information is usually poorly documented if at all. Moreover, there is growing interest in self-tuning, autonomic software systems that can optimize themselves for different platforms, and these systems must determine memory hierarchy parameters automatically without human intervention.

One solution is to use micro-benchmarks to determine the parameters of the memory hierarchy. In this paper, we argue that existing micro-benchmarks are inadequate, and present novel micro-benchmarks for determining the parameters of all levels of the memory hierarchy, including registers, all caches levels and the translation look-aside buffer. We have implemented these micro-benchmarks into an integrated tool that can be ported with little effort to new platforms. We present experimental results that show that this tool successfully determines memory hierarchy parameters on many current platforms, and compare its accuracy with that of existing tools.

1. INTRODUCTION

This paper makes the following contributions.

- We describe novel algorithms for measuring memory hierarchy parameters, including cache parameters such as the capacity, associativity, block size and latency of caches at various levels of a memory hierarchy. Unlike existing micro-benchmarks that consider all levels of the memory hierarchy simultaneously, our micro-benchmarks consider one level of cache at a time, which permits us to measure cache parameters more accurately than existing approaches can. We also describe algorithms for measuring the parameters of the translation look-aside buffer (TLB), and the number of registers.
- We have implemented these algorithms in a tool called X-Ray [15], which is easily ported to new platforms.
- We describe experimental results obtained by running X-Ray on a number of high-performance platforms. These results show that X-Ray can measure more memory hierarchy parameters than existing tools can, and that the results it reports are usually more accurate.

On modern computers, the cost of memory accesses dominates the running time of most applications. To reduce the running time of a program, its memory access patterns can be optimized by transformations such as loop tiling and data reorganization [1]. The implementation of these transformations requires a detailed knowledge of the memory hierarchy of the platform on which the program will run. For example, algorithms for loop tiling use the capacity of the cache to select the tile size. Some of these algorithms use the cache block size and associativity as well to make a more accurate determination of tile size [14]. Similarly, efficient data reorganization requires knowing cache capacity and block size.

Traditionally, these kinds of optimizations were implemented either manually or in a compiler. In either case, the programmer or the compiler writer was assumed to have detailed specifications of the platform. In practice, parameters of the cache hierarchy are poorly documented, if at all, on most systems. On some machines, it may be possible to determine some of this information by reading special registers or records in the processor or operating system [3]. However, most processors and operating systems do not support such mechanisms or provide very limited support. Registers pose a different problem. The number of architected registers is specified in the instruction set, but what is relevant to program optimization is the number of registers that can be used for program variables, which may be different. For example, the SPARC instruction set has 32 architected floating-point registers, but register 0 is hardwired to 0, so the number of registers available to the register allocator is only 31. Therefore, it is useful to have micro-benchmarks to automatically determine memory hierarchy parameter values relevant to program optimization.

The need for such benchmarks is becoming all the more urgent given the trend towards *self-optimizing* software systems that can optimize their own performance without human intervention. Successful systems of this sort include ATLAS [13], which is a portable system that produces highly tuned linear algebra libraries, and FFTW [4] and Spiral [10], which are similar systems for generating digital signal processing libraries. When installed a new machine, these systems execute a set of micro-benchmarks to determine the hardware parameters of the machine, and then use these values to determine optimal values for various software parameters. Some of these systems, such as ATLAS, use global

[*] This work was supported by NSF grants ACI-9870687, EIA-9972853, ACI-0085969, ACI-0090217, ACI-0103723, and ACI-012140.

search to determine optimal values for software parameters, so they use the hardware parameter values only to guide the search process. Other systems use the hardware parameter values to directly estimate optimal values for the software parameters [14]. Once optimal values are determined for the software parameters, these systems generate C code, which is compiled using the native C compiler. This approach can provide portability without compromising performance. We note that accurate micro-benchmarks are key to the success of self-optimizing software as well.

In this paper, we present micro-benchmarks for measuring the parameters of the memory hierarchy of a platform, including all levels of cache, registers, and the TLB. Existing tools such as lmbench [7], Calibrator [6] and MOB [2] measure some of these memory hierarchy parameters, but our experiments show that none of them offer the same coverage of parameters or the accuracy of our micro-benchmarks. These tools implement variations of the micro-benchmark developed by Saavedra [11], which is reproduced in Hennessy and Patterson's architecture book [5] and is discussed in Section 2 of this paper. This benchmark, which is a C program, measures the time required to access a series of array elements with different strides. The timing results are fairly complex because the micro-benchmark considers all levels of the memory hierarchy simultaneously. Therefore, these results are usually interpreted manually to obtain the memory hierarchy parameters. Although tools like MOB, Calibrator and lmbench can determine some cache parameters automatically from these timing results, none of them is able to measure cache associativity for example. Moreover, optimizations performed by modern compilers when compiling the C code can confuse the timing measurements. Yet another problem is that hardware pre-fetching for fixed-stride accessed to memory on machines like the IBM Power architecture can compromise the timing measurements further.

The micro-benchmarks we present in this paper determine cache parameters for one level of cache at a time, rather than consider all levels simultaneously. In particular, when measuring the parameters of a cache level i, the micro-benchmarks ensure that higher cache levels L1, L2, ... are "transparent" to the measurements in the sense that the memory accesses are guaranteed to miss in those caches. Of course this is not a problem for the L1 cache, so our micro-benchmarks use fixed stride accesses there, as we discuss in Section 4. However, our algorithms are novel, and we take care to ensure that compiler optimizations and hardware pre-fetching do not compromise the timing measurements. In Section 5, we show that by using more complex memory access patterns (in particular, a sequence of sequences), we can measure the parameters of lower cache levels without interference from higher cache levels. In Sections 6 and 7, we show how some TLB parameters and the number of registers can be measured automatically. We present experimental results in Section 8 that show that we provide better accuracy and coverage of memory hierarchy parameter measurement than existing tools. Finally, in Section 9, we discuss future work.

2. PREVIOUS APPROACHES

The general approach to measuring memory hierarchy parameters is to repeatedly access the elements of a large array in memory using different strides, and measure the average time per access. The results are then interpreted to deduce different memory hierarchy parameters. The most widely known micro-benchmark for such measurements is the benchmark of Saavedra [11], a stylized version of which is presented in Figure 1. We make the following observations.

1. The benchmark performs series of experiments for pairs of the form $\langle$`csize`, `stride`$\rangle$, where the array size (`csize`) varies between `CACHE_MIN` and `CACHE_MAX` and the stride (`stride`) varies between 1 and `csize`. Both are restricted to powers of 2.
2. For each pair $\langle$`csize`, `stride`$\rangle$, the benchmark traverses the array `x` with the specified stride enough times (`SAMPLE`$\times$ `steps`) to ensure that the total time spend is at least `time` = 1 second.
3. The measurement for the same pair $\langle$`csize`, `stride`$\rangle$ is repeated using the exact same looping code structure, and the same number of times, but this time accessing a single scalar variable (`temp`) instead of elements of the array `x`.

The benchmark has problems at both the algorithmic and implementation level, as summarized below.

1. Algorithmic Level
   (a) The benchmark considers all levels of the memory hierarchy simultaneously, so each timing result is possibly influenced by several parameters from different cache levels. Therefore, the interpretation of the timing results is complex.
   (b) The benchmark does not interpret the timing results to produce actual memory hierarchy parameters itself, but rather produces a set of measurements that need to be interpreted manually.
   (c) The benchmark uses only array sizes restricted to powers of 2, which prevents it from measuring cache capacities that are not a power of 2. However, an increasing number of caches on modern architectures, such as the Level 3 cache on the Itanium 2, have capacities that are not a power of 2.
2. Implementation Level
   (a) The source code uses a very complex looping structure, which is the source of substantial loop overhead. An attempt is made to account for that overhead by measuring and subtracting the time of execution of a cloned version the same looping structure which does not perform any memory accesses. Unfortunately, there is no control over the back-end compiler, so different code may be produced for the two replicas, thus yielding inaccurate results.
   (b) All memory accesses are independent, which allows an aggressively optimizing compiler to schedule them in a way so that some overlap. This is mainly a problem for measuring access latency, but also cause inaccuracies in measuring other parameters.

```
#define SAMPLE (5)
#define CACHE_MIN (1024)
#define CACHE_MAX (16*1024*1024)

int x[CACHE_MAX];

int main ()
{
   int temp;
   for (int csize = CACHE_MIN; csize <= CACHE_MAX; csize *= 2)
      for (int stride = 1; stride <= csize / 2; stride *= 2)
      {
         double time = 0.0;
         int steps = 0;
         int tsteps = 0;
         int limit = csize - stride + 1;
         do
         {
            double time0 = get_time();
            for (int i = SAMPLE * stride; i != 0; --i)
               for (int index = 0; index < limit; index += stride)
                  x[index]++;
            steps++;
            time += get_time() - time0;
         } while (time < 1.0);
         do
         {
            double time0 = get_time();
            for (int i = SAMPLE * stride; i != 0; --i)
               for (int index = 0; index < limit; index += stride)
                  temp += index;
            tsteps++;
            time -= get_time() - time0;
         } while (tsteps < steps);
         printf("size:  %d, stride:  %d, time:  %d",
            csize * sizeof(int), stride * sizeof(int),
            (int)(time * 1E9 / (steps * SAMPLE * stride * ((limit - 1) / (stride + 1)));
      }
}
```

Figure 1: Standard memory hierarchy benchmark

(c) Both memory read and write are performed on the current array element, which introduces interference with write buffers and further prohibit the measurement of these operations in isolation.

(d) The addressing mode used to access array elements involves base address and offset and on many RISC architectures this operation requires an extra address computation instruction before performing the actual memory access instruction.

(e) The source code does not use the values of accessed array elements and more importantly the value of the `temp` variable for anything meaningful, so a smart optimizing compiler can prune portions of the code during dead code elimination.

(f) There is a constant stride between accesses of array elements and some modern architectures provide speculative hardware which is able to prefetch constant stride accesses to memory into the higher levels of the memory hierarhcy.

(g) It is implicitly assumed and very important for the benchmark that the array `x` is stored in a contiguous chunk of memory. In reality, it is only guaranteed to be contiguous in virtual memory and can be fragmented in physical memory. In many cases lower cache levels are physically addressed, which invalidates this important assumption.

The existing systems we examined all use this micro-benchmark in one form or another, although some of them attempt to address some of these problems in various ways. Our approach is very different at the algorithmic level, and it eliminates the implementation problems discussed above.

3. COMPACTNESS OF SEQUENCES

The micro-benchmarks discussed in this paper measure the associativity (A), block size (B), capacity (C), and hit latency (l) of caches. The first three parameters are sometimes referred to as the $\langle A, B, C\rangle$ of caches.

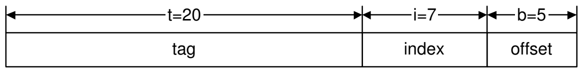


Figure 2: Memory address decomposition on P6

Figure 2 shows the typical structure of a memory address. We use the Intel P6 (Pentium Pro/II/III) architecture in the following explanations. On these machines, the L1 data cache is organized as $\langle A, B, C\rangle = \langle 4, 32, 16\text{KB}\rangle$. Therefore the cache contains $C \div B = 16384 \div 32 = 512$ individual blocks, divided into $512 \div A = 512 \div 4 = 128$ sets of 4 blocks each. The highest $t = 20$ bits constitute the block tag, $i = 7$ bits are needed to index one of the 128 sets, and $b = 5$ bits are needed to store the offset of a particular byte within the 32-byte block.

Definition 1. *For a cache with associativity A and capacity C, we define the* stride T *of that cache as* $T \equiv \frac{C}{A}$.

Note that $T = 2^{i+b}$, and thus $C = A \times 2^{i+b}$. Lemma 1 gives another characterization of T.

Lemma 1. *Consider a cache with stride T, and addresses m_0 and m aligned on a cache block boundary. The address m maps to the same cache set as m_0 iff $m = m_0 + k \times T$ for some integer k.*

Proof. Follows directly from the definition. $\square$

Unlike cache stride, associativity and capacity do not have to be a power of 2. For example, some versions of Intel Itanium 2 have a 24-way set associative L3 cache with capacity 6MB.

If W is a set of addresses, we define $\mathsf{project}_i(W)$ to be the subset of W containing only the addresses that map to cache set i, and $\mathsf{indices}(W)$ to be the set of cache indices of the elements of W.

Definition 2. *For a set of addresses W, and a index i,*

$$\mathit{project}_i(W) \equiv \{m \in W : \mathit{index}(m) = i\}$$

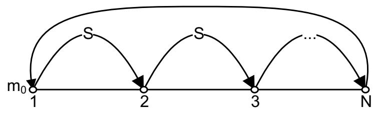
Definition 3. *For a set of addresses W,*

$$\mathit{indices}(W) \equiv \{i : \mathit{project}_i(W) \neq \emptyset\}$$

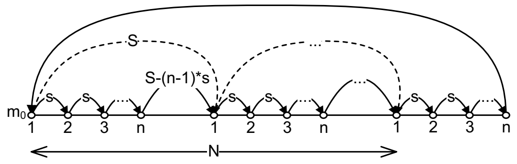
We assume that set-associative caches implement the least-recently-used (LRU) replacement policy. This assumption is reasonable because most modern processors implement variants of this policy. Moreover, our experimental results show that our micro-benchmarks can be accurate even when the policy is not LRU.

3.1 Sequences

Some of our micro-benchmarks access sequences of N addresses, where successive addresses are separated by a stride $S = 2^\sigma$ as shown in Figure 3(a). Such sequences are completely characterized by their starting address m_0, stride S and number of elements N and therefore we use the notation $\langle m_0, S, N\rangle$ to represent them. To measure parameters of multilevel memory hierarchies our micro-benchmarks use sequences of sequences, as shown in Figure 3(b). To represent them we use the notation $W = \langle\langle m_0, s, n\rangle, S, N\rangle$.



(a) $\langle m_0, S, N\rangle$



(b) $\langle\langle m_0, s, n\rangle, S, N\rangle$

Figure 3: Sequences of sequences

Definition 4.

(a) $\langle m_0, S, N\rangle \equiv [m_0, m_0 + S, \ldots, m_0 + (N-1)S]$

(b) $\langle\langle m_0, s, n\rangle, S, N\rangle \equiv \cup_{i\in[0,N-1]} \langle m_0 + i \times S, s, n\rangle$

In Definition 4(b), we call each subsequence $\langle m_0 + i \times S, s, n\rangle$ of $\langle\langle m_0, s, n\rangle, S, N\rangle$ an *inner* subsequence.

Notice that the sequence of addresses in Figure 3(b) can also be expressed as $\langle\langle m_0, S, N\rangle, s, n\rangle$. This property is expressed in Lemma 2.

Lemma 2. $\langle\langle m_0, s, n\rangle, S, N\rangle \equiv \langle\langle m_0, S, N\rangle, s, n\rangle$

3.2 Compactness

We determine cache parameters by measuring the average time per memory access when accessing the elements of certain sets of memory addresses.

When all addresses of an address sequence W can coexist together in a cache we say that W is *compact* with respect to that cache and the average access time is the cache hit latency l_{hit}. When the sequence is not compact and we repeatedly access its elements the cache will suffer some misses. If every single access is a cache miss, we say that W is *non-compact* and the average access time is the cache miss latency l_{miss}, which is typically much greater than l_{hit}. Finally, when some accesses are cache hits and some are cache misses, the average access time is between l_{hit} and l_{miss} and we say that W is *semi-compact.* Definition 5 presents this concepts formally.

Definition 5. *For a cache with associativity A,*

$$\begin{aligned}\mathit{compact}(W) &\equiv \forall i \in \mathit{indices}(W) : |\mathit{project}_i(W)| \leq A\\ \mathit{non\text{-}compact}(W) &\equiv \forall i \in \mathit{indices}(W) : |\mathit{project}_i(W)| > A\\ \mathit{semi\text{-}compact}(W) &\equiv \neg\mathit{compact}(W) \wedge \neg\mathit{non\text{-}compact}(W)\end{aligned}$$

The definition says that, for any cache index from the set of indices for W, a compact sequence will have at most A elements with this index, while a non-compact sequence will have at least $A+1$ elements with this index. A sequence is semi-compact if there is an index with at most A elements, as well as an index with at least $A+1$ elements.

Lemma 3. *Compact sequences have the following properties.*

(a) For a cache with capacity C and block size B, and an address m_0, aligned on a cache block boundary, the half-open interval $[m_0, m_0 + C)$ is a compact.

(b) A subset of a compact sequence is compact.

(c) If $\mathit{indices}(W_1) \cap \mathit{indices}(W_2) = \emptyset$, and W_1 and W_2 are compact then $W_1 \cup W_2$ is compact.

(d) If $\mathit{indices}(W_1) \cap \mathit{indices}(W_2) = \emptyset$, and $W_1 \cup W_2$ is non-compact, then W_1 and W_2 are non-compact.

(e) If W_1 and W_2 are non-compact then $W_1 \cup W_2$ is non-compact.

Proof. (a) The interval $[m_0, m_0 + C)$ is equivalent to the sequence $W = \langle m_0, 1, C\rangle = \left\langle\langle m_0, 1, B\rangle, B, \frac{C}{B}\right\rangle$. Because m_0 is aligned on B, the cache lines used by W are the same as the cache lines used by $\widehat{W} = \left\langle m_0, B, \frac{C}{B}\right\rangle$, in which only one address is mapped to a single cache line. Furthermore $\widehat{W}$ can be expressed as $\left\langle\left\langle m_0, B, \frac{T}{B}\right\rangle, T, \frac{C}{T}\right\rangle$. From Lemma 1, all inner subsequences $\widehat{w_i} = \left\langle m_0 + i \times T, B, \frac{T}{B}\right\rangle$ map exactly one element to each cache set. Therefore $\widehat{W}$ maps exactly $A = \frac{C}{T}$ elements to each cache set, and by Definition 5 it is compact. Because W uses the exact same cache lines, it is also compact.

Results (b)-(e) follow directly from Definition 5. ☐

4. L1 DATA CACHE

To measure the parameters of the L1 data cache our micro-benchmarks measure the average time per element to access the elements of certain compact and non-compact fixed stride sequences.

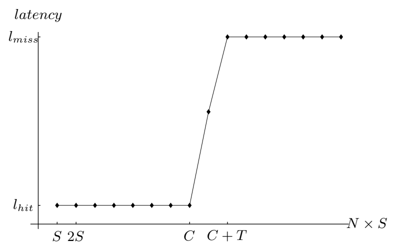


Figure 4: Example of (semi-/non-)compact

Figure 4 gives some intuition about the compactness properties of a sequence $W = \langle m_0, S, N\rangle$ where $S \leq T$. When $N \times S \leq C$ the sequence is compact as it maps at most A addresses to each cache set. When $N \times S \geq C + T$ the sequence is non compact, as it maps at least $A+1$ addresses to each cache set. When $C < N \times S < C + T$, the sequence maps A addresses to some of the cache sets and $A + 1$ address to the rest of the cache sets. For $S \geq T$ there are no semi-compact sequences, and for $S < T$, W is semi-compact for $\frac{T}{S}-1$ different values of N. For example, for $S = \frac{T}{2}$ there is only one $N = \frac{C+S}{S}$ for which the W is semi-compact.

Theorem 1 describes the necessary and sufficient conditions for compactness and non-compactness of a sequence of this type for a given cache. Informally, this theorem says that as the stride S gets bigger, the maximum length of a compact sequence with that stride decreases until it bottoms out at A, while the minimum length of a non-compact sequence with that stride decreases until it bottoms out at $A + 1$.

Theorem 1. *Consider a cache with parameters $\langle A, B, C\rangle$ and a sequence $W = \langle m_0, S, N\rangle$.*

(a) $\textsf{compact}(W) \Leftrightarrow N \leq N_c = A\left\lceil \frac{T}{S}\right\rceil$

(b) $\textsf{non-compact}(W) \Leftrightarrow N \geq N_{nc} = (A+1)\left\lceil \frac{T}{S}\right\rceil$

Proof. There are two cases to consider.

- $S \geq T$. In this case $N_c = A$ and $N_{nc} = A + 1$.

  Since both S and T are powers of 2, S must be an integer multiple of T. From Lemma 1 it follows that all N addresses in the sequence map to the same cache set.

  Therefore the sequence is compact iff for $N \leq A = N_c$ and non-compact iff $N \geq A + 1 = N_{nc}$.

- $S < T$. Since S and T are both powers of 2, $\frac{T}{S}$ is an integer, and $N_c = A \times \frac{T}{S}$ and $N_{nc} = (A + 1) \times \frac{T}{S}$.

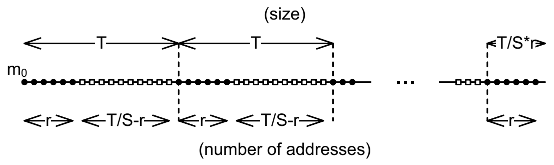


Figure 5: Decomposition of $W = \langle m_0, S, N\rangle$

Let $N = p \times \frac{T}{S} + r$, where $0 \leq r < p$. We can divide W into $p + 1$ parts, in which the first p have $\frac{T}{S}$ elements, and the last one has r elements. Furthermore, we represent W as the union of two sequences of sequences: one with $p + 1$ subsequences of length r and one with p subsequences of length $\frac{T}{S} - r$. This is presented pictorially in Figure 5.

$$\begin{aligned} W &= \langle\langle m_0, S, r\rangle, T, p+1\rangle \\ &\quad \cup \left\langle\left\langle m_0 + r \times S, S, \frac{T}{S} - r\right\rangle, T, p\right\rangle \\ &= \langle\langle m_0, T, p+1\rangle, S, r\rangle \\ &\quad \cup \left\langle\langle m_0 + r \times S, T, p\rangle, S, \frac{T}{S} - r\right\rangle \end{aligned}$$

From Lemma 1, the elements of each inner subsequence map to the same cache set, whereas elements from different subsequences map to different cache sets. Therefore r cache sets will have $p + 1$ different addresses mapped to each of them and $\frac{T}{S} - r$ cache sets will have p addresses mapped to each of them.

– $N < N_c$. In this case $p < A$, i.e. $p + 1 \leq A$. Therefore $\frac{T}{S}$ cache sets have at most A different addresses mapped to each of them, so W is compact.

– $N = N_c$. In this case $p = A$ and $r = 0$. Therefore $\frac{T}{S}$ cache sets have exactly A different addresses mapped to each of them, so W is compact.

– $N_c < N < N_{nc}$. In this case $p = A$ and $0 < r < \frac{T}{S}$. Therefore r cache sets have exactly $A + 1$ different addresses mapped to each of them and $\frac{T}{S} - r$ different cache sets have exactly A different addresses mapped to each of them, so W is neither compact nor non-compact (it is semi-compact).

– $N \geq N_{nc}$. In this case $p \geq A + 1$. Therefore $\frac{T}{S}$ cache sets have at least $A + 1$ different addresses mapped to each of them, so W is non-compact.

The required result follows directly from this.

□

4.1 Algorithms for Measuring Parameters

In this section we use the function $\textsf{is\_compact}(W)$ to determine empirically if W is compact. Our implementation of this function repeatedly accesses each address in W, computes the average time per access l, and declares the sequence to be compact if l is close to the hit latency of the cache l_{hit}, which is measured as described in Section 4.1.1.

Although this procedure seems simple in principle, the timing measurements require some special care to avoid the problems discussed in Section 2 and we discuss how we address these in Section 4.2.

4.1.1 Cache Latency

We determine l_{hit} by measuring the average time to per access of the sequence $\langle m_0, 1, 1\rangle$, which is compact since it contains a single element.

4.1.2 Capacity and Associativity

Theorem 1 suggests a method for determining the capacity C and the associativity A of the cache. First, we find A by determining the asymptotic limit of the length of a compact sequence as the stride is increased. The smallest value of the stride for which this limit is reached is T, the stride of the cache; once we know A and T, we can find C.

```
$S \leftarrow 1$;
$N \leftarrow 1$;
while (is_compact ($\langle m_0, S, N \rangle$))
    $N \leftarrow 2 \times N$
$N_{old} \leftarrow N$;
$N \leftarrow 0$;
while ($N \neq N_{old}$)
    $S \leftarrow 2 \times S$;
    $N_{old} \leftarrow N$;
    $N \leftarrow \min N_{min} \in [1, N_{old}] : \neg$is_compact ($\langle m_0, S, N_{min} \rangle$));
$A \leftarrow N - 1$;
$C \leftarrow \frac{S}{2} \times A$;
```

Figure 6: Measuring C and A of L1 Data Cache

Pseudo-code for measuring C and A of the L1 data cache is shown in Figure 6. The algorithm can be described as follows. Start with the sequence $\langle m_0, S, N \rangle = \langle m_0, 1, 1 \rangle$, which is compact, and keep doubling N until the sequence is not compact. Let N_{old} is the first N for which this happens. Now start doubling the stride S, and for each S compute the smallest N, for which $\langle m_0, S, N \rangle$ is not compact. This value of N can be found by using binary search in the interval $[1, N_{old}]$. If $N \neq N_{old}$, Let $N_{old} = N$ and recompute N for the next S. Repeat this step until $N = N_{old}$. At this point, declare $A = N - 1$ and the $C = \frac{S}{2} \times A$.

The largest stride S used in this algorithm is $2T$. We will exploit this fact when we consider multi-level cache hierarchies.

Note that the number of addresses accessed by the algorithm in this micro-benchmark is on the order of the associativity of the cache, which is superior to previous approaches because non-compactness produces a very pronounced performance drop, which is much easier to detect automatically.

4.1.3 Block Size

For given cache parameters C, A and T, $\langle m_0, T, 2A \rangle$ is non-compact since all $2A$ addresses map to the same cache set. This sequence can also be expressed as $\langle \langle m_0, T, A \rangle, C, 2 \rangle$. If we offset the second half of the sequence by a constant δ, as shown in Figure 7, we get the sequence $\langle \langle m_0, T, A \rangle, C + \delta, 2 \rangle$.

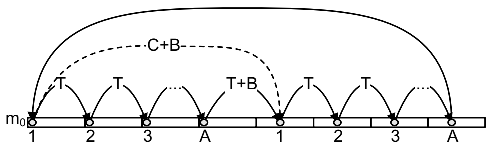


Figure 7: Modified address sequence for measuring B

The addresses in each of the inner subsequences $\langle m_0, T, A \rangle$ and $\langle m_0 + C + \delta, T, A \rangle$ map to a single cache set. When $0 \leq \delta < B$ this cache set is the same for both subsequences. When $\delta \geq B$ they map to two different cache sets. Therefore the smallest value of δ for which the full sequence $\langle \langle m_0, T, A \rangle, C + \delta, 2 \rangle$ is compact is $\delta = B$. Figure 8 shows pseudo-code for the algorithm.

```
$\delta \leftarrow 1$
while (!is_compact ($\langle \langle m_0, T, A \rangle, C + \delta, 2 \rangle$))
    $\delta \leftarrow 2 \times \delta$;
return $\delta$;
```

Figure 8: Algorithm for measuring B

4.2 Implementation of is_compact

The algorithms in Section 4.1 call the function is_compact (W) to determine whether sequence W is compact. We now describe how this function is implemented to avoid the problems discussed in Section 2.

The array of elements is declared of type pointer (`void *`) instead of integer (`int`) as in the Saavedra benchmark. The array is initialized in such a way that each element contains the address of the element which should be accessed immediately after it. A local variable p is initialized with the address of the element which should be accessed first. This initialization is performed off-line, *before* the actual timing.

A simplified version of the timing routine is presented in Figure 9. The variable `R` is chosen so that the loop executes for at least a predetermined amount of time t. Larger values of `R` are likely to produce more accurate timing results at the expense of additional running time. In our implementation, we use $t = 1$ second.

In addition, in the actual implementation, the `while` loop is unrolled several times to avoid loop overhead.

```
$startTime \leftarrow$ get_time();
while (--R)
    $p \leftarrow$ *(void **)$p$;
$timePerAccess \leftarrow$ (get_time() $-$ $startTime$) $\div$ R;
printf("", p);
```

Figure 9: Improved timing of memory accesses

It is easy to see that the only operation performed in the loop body is $p \leftarrow$ `*(void **)`p, which reads the memory address stored at address p and updates p with it, effectively following the pointer chain preprogrammed inside the array.

The following points address the implementation problems of the Saavedra benchmark discussed in Section 2.

(a) The code in Figure 9 uses the simplest possible looping structure, and loop overhead can be reduced as much as needed, by sufficient unrolling. In our implementation we unroll 256 times.

(b) Each of the memory accesses depends on the previous one to produce the actual address to access, so aggressive compilers cannot take advantage of instruction-level parallelism and overlap them.

(c) Each memory access constitutes precisely one memory read instruction, so the actual timing corresponds exactly to the average latency per access.

(d) All modern architectures today support indirect addressing mode, so each operation should be translated to a single machine instruction (e.g. "`lea eax, [eax]`" on x86 ISA).

(e) The final value of the variable p is used by the `printf` statement, so the compiler is not able to optimize the memory accesses away by dead code elimination.

(f) For a correct implementation of is_compact (W), it is important that we repeatedly access all elements of the sequence, but the actual order in which we access them is irrelevant. To prevent hardware constant stride prefetchers, like those on the IBM Power architecture, from interfering with our timings, we initialize the array elements by chaining the pointers so that we visit the elements in a pseudo-random order.

Suppose the address sequence is $m_0, m_1, \ldots, m_{n-1}$. One way to reorder this sequence is to choose a number p, such that p and n are mutually prime. Then, after element m_i, visit element $m_{(i+p) \text{ modulo } n}$ instead of element $a_{(i+1) \text{ modulo } n}$. As p and n are mutually prime, the recurrence $i \leftarrow (i+p)$ modulo n is guaranteed to generate all the integers between 0 and $n-1$ before repeating itself.

(g) All modern processors have virtually indexed L1 data caches and therefore physical continuity is not an issue. Lower levels of the memory hierarchy are usually physically indexed, so physical continuity is important for lower levels of the memory hierarchy, as we discuss in Section 5.4.

5. LOWER LEVELS OF THE MEMORY HIERARCHY

We denote the cache at level i as $\mathcal{C}_i$, its $\langle A, B, C\rangle$ parameters as $\langle A_i, B_i, C_i\rangle$, its stride as T_i and its hit latency as l_i. We extend the notation from the previous section, so that $\mathsf{compact}_i(W)$ denotes that $\mathsf{compact}(W)$ with respect to $\mathcal{C}_i$. We extend non-compact and semi-compact in the same way.

Measuring parameters of lower levels of the memory hierarchy is considerably more difficult than measuring the parameters of the L1 data cache. One reason why the algorithms described in Section 4 cannot be used directly is that $\mathcal{C}_i$ is accessed only if $\mathcal{C}_{i-1}$ suffers a miss. Therefore compactness with respect to $\mathcal{C}_i$ of a sequence of addresses can be accurately determined empirically only if this sequence is non-compact with respect to $\mathcal{C}_1, \mathcal{C}_2, \ldots \mathcal{C}_{i-1}$.

Our solution to this problem is to transform any sequence W into a new sequence W^*, with the following properties.

1. $\mathsf{compact}_i(W^*) \Leftrightarrow \mathsf{compact}_i(W)$
2. $\mathsf{non\text{-}compact}_j(W^*)$, for all $j \in [1, i-1]$

Intuitively, W is of the form presented in Figure 3(a). We want to transform it to W^*, which is a sequence of sequences of the form presented in Figure 3(b), so that the extra memory accesses exhaust the associativity at cache levels above $\mathcal{C}_i$. Such a transformation may be necessary because on some architectures, lower level caches are less associative than higher level caches. For example some versions of the IBM Power 3 have 8MB, 8-way set associative $\mathcal{C}_2$ and 64KB, 128-way set associative $\mathcal{C}_1$. Therefore the final iteration of the algorithm in Figure 6 should be examining the sequence $W = \langle m_0, 2\text{MB}, 9\rangle$ and declaring it non-compact. Without transforming W this will not happen, because although the sequence is non-compact with respect to $\mathcal{C}_2$, it is compact with respect to $\mathcal{C}_1$. As we discuss later, the corresponding W^* we use for such W is $W^* = \langle\langle m_0, 512, 15\rangle, 2\text{MB}, 9\rangle$, which is non-compact with respect to $\mathcal{C}_1$. Another way to view this sequence is $W^* = \langle\langle m_0, 2\text{MB}, 9\rangle, 512, 15\rangle$, i.e. 15 copies of the original sequence W shifted by a factor of 512. Each of these copies behaves identically to the original W with respect to $\mathcal{C}_2$, but together they force non-compactness with respect to $\mathcal{C}_1$.

To generalize Theorem 1 to sequences of sequences we first prove Lemma 4.

LEMMA 4. *Consider a cache with parameters $\langle A, B, C\rangle$ and stride T. If $W_1 = \langle m_0, S, N\rangle$ and $W_2 = \langle m_0 + \delta, S, N\rangle$, where m_0 and $m_0 + \delta$ is aligned on a cache block boundary, and $0 < \delta < \min(S, T)$, then $\mathsf{indices}(W_1)$ and $\mathsf{indices}(W_2)$ are disjoint.*

PROOF. We will consider two cases.

First, let $S \geq T$. From Lemma 1 all elements of W_1 map to the same cache set i_1; similarly all elements of W_2 map to the same cache set i_2. Because $\delta < T$, m_0 and $m_0 + \delta$ map to different cache sets, so $i_1 \neq i_2$.

Second, let $S < T$. Let $N = p \times \frac{T}{S} + r$, where $0 \leq r < \frac{T}{S}$. Then:

$$W_1 = \langle m_0, S, N\rangle \subset \left\langle m_0, S, (p+1) \times \frac{T}{S}\right\rangle = \widehat{W_1}$$

$$W_2 = \langle m_0 + \delta, S, N\rangle \subset \left\langle m_0 + \delta, S, (p+1) \times \frac{T}{S}\right\rangle = \widehat{W_2}$$

Now we split W_1 and W_2, which both have $(p+1) \times \frac{T}{S}$ elements, into two sequences of $p+1$ subsequences with $\frac{T}{S}$ elements each.

$$\widehat{W_1} = \left\langle \widehat{w_1} = \left\langle m_0, S, \frac{T}{S}\right\rangle, T, p+1\right\rangle$$

$$\widehat{W_2} = \left\langle \widehat{w_2} = \left\langle m_0 + \delta, S, \frac{T}{S}\right\rangle, T, p+1\right\rangle$$

From Lemma 1, $\mathsf{indices}\left(\widehat{W_1}\right) = \mathsf{indices}(\widehat{w_1})$ and $\mathsf{indices}\left(\widehat{W_2}\right) = \mathsf{indices}(\widehat{w_2})$. The addresses of the last elements of $\widehat{w_1}$ and $\widehat{w_2}$ are $m_0 + T - S$ and $m_0 + \delta + T - S$ respectively. Therefore , all addresses in $\widehat{w_1}$ and $\widehat{w_2}$ are contained in the half-open interval $[m_0, m_0 + T)$. Any two addresses $m_1 \in \widehat{w_1}$ and $m_2 \in \widehat{w_2}$ are aligned on a cache block boundary and therefore from Lemma 1 they map to different cache sets. Therefore $\mathsf{indices}(\widehat{w_1})$ and $\mathsf{indices}(\widehat{w_2})$ are disjoint, so $\mathsf{indices}\left(\widehat{W_1}\right)$ and $\mathsf{indices}\left(\widehat{W_2}\right)$ are disjoint, which implies that $\mathsf{indices}(W_1)$ and $\mathsf{indices}(W_2)$ are disjoint. $\square$

THEOREM 2. *Consider a cache with parameters $\langle A, B, C\rangle$ and stride T, and a sequence of sequences $W^* = \langle\langle m_0, s, n\rangle, S, N\rangle$, where $(n-1) \times s < \min(T, S)$ and $B \leq s$.*

(a) $\mathsf{compact}(W^*) \Leftrightarrow N \leq N_c = A\left\lceil\frac{T}{S}\right\rceil$

(b) $\mathsf{non\text{-}compact}(W^*) \Leftrightarrow N \geq N_{nc} = (A+1)\left\lceil\frac{T}{S}\right\rceil$

PROOF. From Lemma 2 and Definition 4,

$$W^* = \langle\langle m_0, S, N\rangle, s, n\rangle = \cup_{i\in[0,n-1]} \langle m_0 + i \times s, S, N\rangle .$$

From Theorem 1 each of the sequences $w_i = \langle m_0 + i \times s, S, N\rangle$ for $i \in [0, n-1]$ is compact for $N \leq N_c$, non-compact for $N \geq N_{nc}$, and semi-compact otherwise.

From Lemma 4, $\mathsf{indices}(w_i)$ are pairwise disjoint sets for all $i \in [0, n-1]$. The required result follows from Lemma 3. $\square$

Note that Theorem 1 is a special case of Theorem 2 for $n = 1$. In this case the constraint $(n-1) \times s < T$ is trivially true and the sequence $\langle m_0, s, n\rangle$ has a single element (m_0).

5.1 Two Cache Levels

Consider two cache levels, $\mathcal{C}_1 = \langle A_1, B_1, C_1\rangle$ and $\mathcal{C}_2 = \langle A_2, B_2, C_2\rangle$.

To apply the algorithms in Section 4.1 to measure parameters for $\mathcal{C}_2$, we replace each sequence W in those algorithms with a sequence of sequences W^*, such that $\mathsf{compact}_2(W^*) \Leftrightarrow \mathsf{compact}_2(W)$ and $\mathsf{non\text{-}compact}_1(W^*)$.

Ideally we would have a general construction that could construct such a W^* from any $W = \langle m_0, S, N\rangle$. Since we do not have such a general construction, we will present an approach, which works for the particular sequences used by the algorithms in Section 4.1. In particular we will restrict ourselves to sequences for which $S \le 2T$ (because $2T$ is the largest stride used by these algorithms). Furthermore, it is invariably the case that $C_2 \ge 2C_1$, so if $(N-1) \times S \le 2C_1$ the sequence W can be assumed compact without performing an empirical measurement. Therefore we can restrict ourselves to sequences for which $(N-1) \times S > 2C_1$.

With these restrictions, we choose

$$W^* = \langle m_0, S, N\rangle = \langle\langle m_0, s, n\rangle, S, N\rangle,$$

where:

$$\begin{aligned} s &= T_1 \\ n &= \left\lceil \frac{A_1+1}{N} \right\rceil . \end{aligned}$$

LEMMA 5. *If $S \le 2T_2$ and $(N-1) \times S > 2C_1$ then*

(a) $\mathsf{compact}_2(W^*) \Leftrightarrow \mathsf{compact}_2(W)$ and

(b) $\mathsf{non\text{-}compact}_1(W^*)$.

PROOF.

(a) First we show that Inequality (1) holds.

$$\left(\left\lceil \frac{A_1+1}{N} \right\rceil - 1\right) \times T_1 < \frac{S}{2} \qquad (1)$$

The opposite is impossible, because then:

$$\begin{aligned} \frac{S}{2} &\le \left(\left\lceil \frac{A_1+1}{N} \right\rceil - 1\right) \times T_1 \le \left(\frac{A_1+N}{N} - 1\right) \times T_1 \\ &\le \frac{A_1}{N} \times T_1 < \frac{S}{2C_1} \times A_1 \times T_1 = \frac{S}{2} \Rightarrow \frac{S}{2} < \frac{S}{2}. \end{aligned}$$

From (1) and $S \le 2T_2$ we conclude that $\left(\left\lceil \frac{A_1+1}{N} \right\rceil - 1\right) \times T_1 < \min(S, T_2)$. Therefore we can apply Theorem 2 to W^*, and so $\mathsf{compact}_2(W^*) \Leftrightarrow N \le N_c$, where $N_c = A_2 \times \left\lceil \frac{T_2}{S} \right\rceil$. On the other hand, from Theorem 1, $\mathsf{compact}_2(W) \Leftrightarrow N \le N_c$. Therefore $\mathsf{compact}_2(W^*) \Leftrightarrow \mathsf{compact}_2(W)$.

(b) From $(N-1) \times S > 2C_1$ we obtain:

$$\begin{aligned} N &> \frac{2C_1 + S}{S} \ge \frac{A_1 \times T_1 + T_1}{S} = (A_1+1) \times \frac{T_1}{S} \\ &\ge (A_1+1) \times \left\lceil \frac{T_1}{S} \right\rceil \Rightarrow N > (A_1+1) \times \left\lceil \frac{T_1}{S} \right\rceil \end{aligned}$$

From Theorem 2, it follows that $\mathsf{non\text{-}compact}_1(W^*)$

□

5.2 Multiple Cache Levels

To generalize the approach from Section 5.1 to multiple cache levels $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_k$ we replace W with $W^* = \langle\langle m_0, s, n\rangle, S, N\rangle$, where

$$\begin{aligned} s &= \min_{i<k} T_i \\ n &= \max_{i<k} \left\lceil \frac{A_i+1}{N} \right\rceil \times \frac{T_i}{s} \end{aligned}$$

LEMMA 6. *If $S \le 2T_k$ and $(N-1) \times S > 2C_i$ for all $i \in [1, k-1]$ then*

(a) $\mathsf{compact}_k(W^*) \Leftrightarrow \mathsf{compact}_k(W)$ and

(b) $\mathsf{non\text{-}compact}_i(W^*)$ for all $i \in [1, k-1]$.

PROOF.

(a) By analogy with Inequality (1), Inequality (2) holds.

$$\max_{i \in [1,k-1]} \left(\left\lceil \frac{A_i+1}{N} \right\rceil - 1\right) \times T_i < \frac{S}{2} \qquad (2)$$

Therefore:

$$\begin{aligned} (n-1) \times s &= \left(\max_{i<k} \left\lceil \frac{A_i+1}{N} \right\rceil \times \frac{T_i}{s} - 1\right) \times s \\ &= \max_{i<k} \left\lceil \frac{A_i+1}{N} \right\rceil \times T_i - s \\ &< \frac{S}{2} - s \\ &< \min(S, T_k). \end{aligned}$$

From Theorem 2, applied to W^*, it follows that if $N_c = A_2 \times \left\lceil \frac{T_k}{S} \right\rceil$, then $\mathsf{compact}_k(W^*) \Leftrightarrow N \le N_c$. From Theorem 1, $\mathsf{compact}_k(W) \Leftrightarrow N \le N_c$ for the same N_c. Therefore $\mathsf{compact}_k(W^*) \Leftrightarrow \mathsf{compact}_k(W)$.

(b) $(N-1) \times S > 2C_i$ for all $i \in [1, k-1]$ and by analogy with the proof of Theorem 2(b), $\mathsf{non\text{-}compact}_i(W^*)$ holds for all $i \in [1, k-1]$.

□

5.3 Algorithms for Measuring Parameters

We use the function $\mathsf{is\_compact}_i(W)$ to determine empirically if $\mathsf{compact}_i(W)$ holds. Our implementation of this function repeatedly accesses each address in W, computes the average time per access l, and declares the sequence to be compact if l is close to l_i (the hit latency of $\mathcal{C}_i$).

Given the transformation from W to W^* as discussed in Section 5.2, we can use the algorithms in Section 4.1 to measure latency, capacity and associativity at any cache level.

5.4 Implementation of is_compact

There is one important complication when measuring parameters of lower cache levels. On modern platforms $\mathcal{C}_1$ is typically virtually indexed, but lower levels are always physically indexed. This is a problem because continuity in virtual memory is not a sufficient condition for continuity in physical memory, and thus a fixed stride sequence of addresses in the virtual address space may not map to a fixed stride sequence in physical address space.

To measure parameters of lower cache levels it is therefore necessary to allocate physically contiguous memory. There are two ways to acquire such memory in a modern operating system: (i) request physically contiguous pages from the kernel, or (ii) request virtual memory backed by a superpage.

The first approach is generally possible only in kernel mode, and there are strict limits on the amount of allocatable memory. It is mainly used for direct memory access (DMA) devices. Another, somewhat smaller problem is that such memory regions typically consist of many pages and TLB misses might introduce inter-level interference noise in our cache measurements.

The second approach is more promising, but currently there is no portable way to request super-pages from all operating systems. To address this problem, in our implementation we provide OS-specific memory allocation and deallocation routines, which are then used by the cache microbenchmarks to allocate memory supported by super-pages. We have implemented this approach for Linux, and we will implement it for other operating systems in the near future.

There has been some work on transparently supporting variable size pages in the OS [9]. When such support becomes generally available, our OS-specific solution will not be required.

6. MEASURING TLB PARAMATERS

The general structure of a virtual memory address is shown in Figure 10 (the field widths are Intel P6 specific). The low-order bits contain the page offset, while the hi-order bits are used for indexing page tables during the translation to a physical address. Because the translation from virtual to physical address is too expensive to perform on every memory access, a TLB is used to cache and reuse the results.

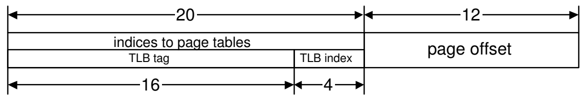


Figure 10: Memory address decomposition on P6

A TLB has a certain number of *entries* E each of which can cache the address translation for a single virtual memory page of size P. Even though TLB does not store the actual data but only its physical address and a few flags, it uses the upper portion of the virtual address in a way a normal cache does (for encoding index and tag), and so we can consider it a normal cache $\mathcal{C}_{TLB} = \langle A, B, C\rangle = \langle A_{TLB}, P, E \times P\rangle$. Ideally we would like to use our cache parameter measurement algorithms discussed in Section 4.1, but some complications arise as outlined below.

1. *Variable page size*: measuring parameters for caches with variable block size is not possible with our current algorithms. On current operating systems, the default is to use only a single page size, and therefore there is no immediate danger of measurement failure. Furthermore, [9] suggests that when transparent support for multiple page sizes becomes available, TLB misses will be automatically minimized and will have negligible impact on performance. At that point measuring the TLB parameters would not be necessary.

2. *Replacement policy*: typically a TLB has high associativity and LRU is impractical to implement because of speed issues. In practice processors use much simpler replacement policies like round-robin or random. Some even perform a software interrupt on a TLB miss and leave to the operating system to do the replacement. Surprisingly these inconsistencies do not prevent us from producing accurate measurement results.

3. *Ensuring TLB access*: As in the case of lower cache levels, we need to make sure that the TLB is accessed when memory references are issued by the processor. In modern platforms this is ensured by the fact that L1 data caches are usually physically tagged, but even more importantly by the fact that TLB caches memory protection information which is needed to complete the particular memory operation.

4. *Physical Continuity*: As with lower cache levels, we need physically contiguous memory to perform TLB measurements. Unfortunately, using super-pages is not an alternative for obvious reasons, and so a kernel module is required.

For a sequences $W = \langle m_0, S, N\rangle$, let $N = p \times \left\lceil \frac{T_1}{S} \right\rceil + r$, where $0 \le r < \left\lceil \frac{T_1}{S} \right\rceil$. To measure TLB parameters using the algorithms described in Section 4.1, we transform W into (T_1 and B_1 are the stride and the block size of $\mathcal{C}_1$ respectively):

$$W^* = \left\langle \left\langle m_0, S, \left\lceil \frac{T_1}{S} \right\rceil \right\rangle, T_1 + B_1, p \right\rangle \cup \langle m_0 + (p-1) \times (T_1 + B_1), S, r\rangle$$

We assume that the $\mathcal{C}_1$ has at least twice as many blocks as there are entries in the TLB, i.e. $\frac{C_1}{B_1} \ge 2\frac{C_{TLB}}{B_{TLB}}$, which is true for all modern platforms today. Under this assumption, it is easy to see that $\mathsf{compact}_1(W^*)$.

Because we do not have a portable solution to (4) above, our experience with measuring TLB parameters is limited. None of the other tools produced any correct results on any of the tested platforms. Therefore, we describe our limited experimental results in this section.

Using the algorithms in Section 4.1 with the modified sequences W^*, we were able to accurately measure the TLB parameters of a Pentium III as 64 page entries, 4-way set associative, and page size of 4KB. We also measured the TLB parameters of a Pentium 4 as 65 page entries, fully-associative, with a page size of 4KB. On the Pentium 4 our measurement is close to the correct one (measured associativity 65 vs. actual associativity of 64)[1]. In the final paper, we will present TLB results for other platforms.

7. MEASURING AVAILABLE REGISTERS

Registers are often considered a level-0 cache $\mathcal{C}_0$, as they are at the top of the memory hierarchy. If a machine has N registers of type T, we can characterize $\mathcal{C}_0 = \langle A, B, C\rangle = \langle N, \mathsf{sizeof}(T), N \times \mathsf{sizeof}(T)\rangle$. $\mathcal{C}_0$ can exhibit spacial locality only in the case of vector registers (MMX, SSE, etc.). Furthermore, it is fully associative and the replacement policy is software controlled.

The only way to directly exercise this control is to program in assembly language. Portable software, on the other hand, is usually written in a high-level language like C and the native compiler is responsible for register allocation, register spills and fills. Nevertheless, when the ultimate goal is high-performance, programmers need to make assumptions about the number of registers available for register allocation and apply optimizing transformations like array scalarization and loop unrolling appropriately (e.g. ATLAS [13]).

Our approach to measuring the number of registers of particular type `T` is to generate special code sequences that access n different variables, measure the time per operation for several n, and infer the number of registers from the results.

[1]This problem may be similar to the one we discuss about the L1 data cache of Power 3 in Section 8.1

```
$r_0 \leftarrow \mathsf{add}\,(r_0, r_n)$;
$r_1 \leftarrow \mathsf{add}\,(r_1, r_0)$;
$r_2 \leftarrow \mathsf{add}\,(r_2, r_1)$;
...
$r_n \leftarrow \mathsf{add}\,(r_n, r_{n-1})$;
```

Figure 11: Sequence with n variables

The particular kind of sequences we are using is presented in Figure 11. Note that if the compiler is able to allocate all n variables into registers, each add operation will be translated to a single ALU instruction. On the other hand, if at least one variable is not allocated to a register, additional memory access instructions will be emitted in addition to the ALU instruction to fetch the data from the memory hierarchy. Since each operation in the sequence depends on the previous one, the incurred additional latency cannot be hidden and the average time per operation is much higher.

Measuring the number of available registers reduces to finding the longest code sequence whose average access time is the same as that of the sequence of length 1. In our implementation we start with $n = 1$ and keep doubling it until an increase in access time is observed, say for $n = n_{max}$. Then we use binary search to find the the n we need in the interval $\left[\frac{n_{max}}{2}, n_{max}\right)$.

Note that this method measures the *effective* number of available registers, which is the value that is relevant for program optimization. This value can often be smaller than the number of actual registers on the given architecture for the following reasons.

- Some registers may be reserved for the Stack Pointer, Frame Pointer, Return Address, etc.
- Some registers may be hardwired with specific values, most often the floating point values 0.0 and 1.0.
- Compilers may use some registers in a special way, and they might not be available to the general register allocator, e.g. accumulators, register windows, etc.
- Compilers might not use all available registers for different reasons, e.g. targeting an older version of the ISA.

By appropriately defining the operation add, this method is able to measure all types of registers, including integer, floating point, and vector registers (e.g. MMX, SSE, 3DNow!, Altivec) through compiler intrinsics.

None of lmbench, Calibrator, and MOB try to measure the number of available registers. The ATLAS framework attempts to provide a rough estimate for the number of floating point registers, but they can afford to be conservative, as opposed to precise, because they only use the estimate to bound their search space. Table 1 summarizes our measurement results.

As expected the number of available integer registers is always less than the actual number of registers because some registers are reserved for use either by the hardware or by the compiler. The measured number of floating point registers is equal to the actual number in all cases except on the UltraSPARC IIIi machine, where one of the registers is hardwired to 0.0. The measured number of vector registers is always equal to the actual number. We do not provide results for 3DNow! and SSE2 registers, because they are equivalent to MMX and SSE register respectively.

	available / actual			
Architecture	**int**	**double**	**MMX**	**SSE**
Pentium 4	5 / 8	8 / 8	8 / 8	8 / 8
Itanium 2	123 / 128	128 / 128	n/a	n/a
Athlon MP	5 / 8	8 / 8	8 / 8	8 / 8
Opteron 240	14 / 16	16 / 16	8 / 8	16 / 16
UltraSPARC IIIi	24 / 32	31 / 32	n/a	n/a
R12000	22 / 32	32 / 32	n/a	n/a
Power 3	28 / 32	32 / 32	n/a	n/a

Table 1: Experimental results for registers

8. EXPERIMENTAL RESULTS

The implementation of the memory micro-benchmarks described in this paper is part of an open micro-benchmark tool called X-Ray [15]. To report cache latency in CPU cycles we use a micro-benchmark for measuring CPU frequency, which is part of X-Ray. In this section we compare the results of running the memory-hierarchy portion of X-Ray on 7 platforms with the results of running the following three tools.

- **Calibrator v0.9e** [6] is a memory system benchmark aimed at measuring capacity, block size, and latency at each level of the memory hierarchy and TLB parameters, such as number of entries, page size, and latency.
- **lmbench v3.0a3** [8, 12, 7] is a suite of benchmarks for measuring operating systems parameters such as thread-creation time and context-switch time. Version 3, contains micro-benchmarks for measuring latency and parallelism of different operations, capacity, block size, and latency of each level of the memory hierarchy, and the number of TLB entries.
- **MOB v0.1.1** [2] is an ambitious project to create a benchmark suite capable of measuring a large number of properties of the memory hierarchy, including capacity, block size, associativity, sharedness, replacement policy, write mode, and latency of each level, as well as the corresponding TLB parameters.

Because all the tools, including X-Ray, measure hardware parameters empirically, the results sometimes vary from one execution to the next. These variations are negligibly small with X-Ray, but sometimes quite noticeable with the other tools. The results we present for the other tools are the best ones we obtained in several trial runs.

Table 2 shows the memory hierarchy parameters, along with the results from measuring them with the different tools. Whenever a parameter was not successfully computed, we use the following special entries to specify the reason:

- **n/a** – the tool does not claim to be able to measure this hardware parameter;
- **empty** – the benchmark completed but did not produce a value for this parameter;
- **abort** – an abnormal termination of some kind occurred prevented the benchmark from completion;
- **build** – the benchmark did not build successfully;

- **os** – OS-specific support is required for X-Ray to complete this measurement and we have not implemented such support yet.

8.1 L1 Data Cache

As Table 2 shows, X-Ray successfully found the correct values for all L1 cache parameters on all the platforms other than the Power 3, where it decided that the cache was 129-way set associative although it is actually 128-way set-associative. For reasons we do not understand, there was no performance loss in the micro-benchmark when moving from 128 to 129 steps, but there was a performance loss in moving from 129 to 130. This anomaly also affected the determination of the cache capacity slightly. The performance of the other tools varies, and the details are presented in Table 2.

8.2 Lower Level Caches

Lower level caches are physically addressed on all modern machines so we found it necessary to use super-pages to obtain consistent measurements of lower level cache parameters, as discussed in Section 5.4. Support for super-pages is very OS-specific, so we targeted the Linux system as a proof of concept. Table 2 shows that X-Ray was able to measure lower level cache parameters correctly on all the Linux machines in our study (Pentium 4, Itanium 2, Athlon MP, and Opteron 240). We are currently working on the implementation for Solaris, IRIX and AIX, which will allow us to test X-Ray on the rest of the machines as well. These results will be reported in the final paper.

The numbers for the AMD machines (Athlon and Opteron) are interesting because they expose the fact that the L1 and L2 caches on these machines implement cache *exclusion*. Most platforms support cache *inclusion*, which means that information cached at a particular level of the memory hierarchy should also be cached in all lower levels. This is necessary to support cache-coherency protocols in SMP systems. AMD machines on the other hand use exclusion, so data never resides in both the L1 and L2 caches simultaneously. While this requires the L1 cache to snoop on the bus to resolve coherency issues, it effectively increases the useful capacity of L2 by the capacity of the L1.

X-Ray classified the 512KB, 16-way associative L2 cache of the AthlonMP as an 18-way set-associative cache with a capacity of 576KB (exactly $C_1 + C_2$). Similarly on the Opteron 240, the 1MB L2 was classified as a 17-way set associative cache with an effective capacity 1088KB (exactly $C_1 + C_2$). If the actual capacity of the L_2 cache is needed, it can be obtained by subtracting the capacity of the L_1 cache, although the combined capacity is what is actually relevant for an autonomic code that wants to perform an optimization like cache tiling.

The performance of the other tools varied. Calibrator produced somewhat pessimistic results for cache capacity on some of the Linux machines; we believe this effect too arises from non-contiguous physical memory since this reduces the effective cache capacity. lmbench terminates abnormally on some platforms, but produces accurate results when it terminates cleanly. MOB produced accurate results only for the capacity of the L2 cache of Itanium 2. In all other cases, it either aborted, produced a wrong result or did not produce a result at all.

	Architecture	**Actual**	**X-Ray**	**Calibrator**	**lmbench**	**MOB**
L1 *C* (KB)	Pentium 4	8	8	8	8	8
	Itanium 2	16	16	16	abort	4
	Athlon MP	64	64	64	empty	abort
	Opteron 240	64	64	64	abort	empty
	UltraSPARC IIIi	64	64	64	64	abort
	R12000	32	32	32	32	build
	Power 3	64	64.5	64	64	empty
L1 *B* (bytes)	Pentium 4	64	64	32	64	abort
	Itanium 2	64	64	64	abort	104
	Athlon MP	64	64	64	empty	abort
	Opteron 240	64	64	32	abort	empty
	UltraSPARC IIIi	32	32	32	32	abort
	R12000	16	16	64	32	build
	Power 3	128	128	128	128	empty
L1 *A* (count)	Pentium 4	4	4	n/a	n/a	empty
	Itanium 2	4	4	n/a	n/a	empty
	Athlon MP	2	2	n/a	n/a	empty
	Opteron 240	2	2	n/a	n/a	empty
	UltraSPARC IIIi	4	4	n/a	n/a	empty
	R12000	2	2	n/a	n/a	empty
	Power 3	128	129	n/a	n/a	empty
L1 *l* (cycles)	Pentium 4	2	2	2	2	abort
	Itanium 2	2	2	2	abort	5
	Athlon MP	3	3	3	empty	abort
	Opteron 240	3	3	3	abort	empty
	UltraSPARC IIIi	2	2	2	2	abort
	R12000	2	2	2	2	build
	Power 3	2	2	2	2	empty
L2 *C* (KB)	Pentium 4	512	512	384	512	abort
	Itanium 2	256	256	256	abort	256
	Athlon MP	512	576	384	512	abort
	Opteron 240	1024	1088	768	abort	empty
	UltraSPARC IIIi	512	os	1024	1024	abort
	R12000	512	os	2048	2048	build
	Power 3	512	os	6144	6144	0
L2 *B* (bytes)	Pentium 4	128	128	128	128	abort
	Itanium 2	128	128	128	128	empty
	Athlon MP	64	64	64	64	abort
	Opteron 240	64	64	64	64	empty
	UltraSPARC IIIi	64	os	64	64	abort
	R12000	128	os	128	128	build
	Power 3	128	os	128	128	empty
L2 *A* (count)	Pentium 4	8	8	n/a	n/a	empty
	Itanium 2	8	8	n/a	n/a	empty
	Athlon MP	16	18	n/a	n/a	empty
	Opteron 240	16	17	n/a	n/a	empty
	UltraSPARC IIIi	?	os	n/a	n/a	empty
	R12000	?	os	n/a	n/a	empty
	Power 3	?	os	n/a	n/a	empty
L2 *l* (cycles)	Pentium 4	?	21	18	20	abort
	Itanium 2	?	6	4	abort	6
	Athlon MP	?	36	18	3	abort
	Opteron 240	?	23	13	abort	empty
	UltraSPARC IIIi	?	13	12	15	abort
	R12000	?	14	12	14	build
	Power 3	?	18	9	17	1
Memory *l* (cycles)	Pentium 4	?	381	372	368	abort
	Itanium 2	?	298	281	abort	empty
	Athlon MP	?	471	401	198	abort
	Opteron 240	?	136	127	abort	empty
	UltraSPARC IIIi	?	os	164	173	abort
	R12000	?	os	111	122	build
	Power 3	?	os	136	161	empty

Table 2: Summary of experimental results

	Actual	X-Ray	Calibrator	lmbench	MOB
C (KB)	6144	6144	6144	abort	4096
B (bytes)	128	128	128	abort	empty
A (count)	24	24	n/a	n/a	n/a
l (cycles)	?	19	14	abort	6

Table 3: Summary of Itanium 2 $\mathcal{C}_3$ parameters

The cache access latency figures produced by all the tools for lower level caches should be taken with a grain of salt since the actual access time can fluctuate substantially depending on what other memory bus transactions are occurring at the same time.

We discussed our experimental results for measuring TLB parameters and number of registers in Secions 6 and Section 7 respectively.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we described novel algorithms for measuring the associativity, block size, and capacity of all levels of the memory hierarchy, as well as TLB parameters and number of registers. The experimental results show that our approach automatically measures more parameters with greater precision than existing approaches. This is because our micro-benchmarks measure the parameters of one level of the memory hierarchy at a time, unlike existing tools that consider all levels simultaneously. To do this, our micro-benchmarks measure access time for more complex sequences of addresses than existing tools do.

The memory hierarchy benchmarks described here are implemented as part of an open framework for development of micro-benchmarks called X-Ray [15]. X-Ray can also measure the following hardware parameters:

- CPU frequency,
- instruction latency and throughput,
- instruction existence (e.g. fused multiply-add),
- SMP and SMT availability, and
- the number and type of functional units in the CPU.

We are actively designing and developing new micro-benchmarks and we are currently working on:

- implementing OS support for Solaris, AIX, etc.,
- improving quality of TLB measurements,
- measuring instruction cache parameters,
- cache bandwidth, parallelism, write mode, and sharedness (unified or dedicated).

X-Ray is freely available and a URL for downloading it will be in the final paper.

10. REFERENCES

[1] R. Allan and K. Kennedy. *Optimizing Compilers for Modern Architectures.* Morgan Kaufmann Publishers, 2002.

[2] Josep M. Blanquer and Robert C. Chalmers. MOB: Memory Organization Benchmark. `http://www.nmsl.cs.ucsb.edu/mob`.

[3] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou. Experiences and lessons learned with a portable interface to hardware performance counters. In *PADTAD Workshop, IPDPS 2003*, April 2003.

[4] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[5] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 1990.

[6] Stefan Manegold. The calibrator: a cache-memory and TLB calibration tool. `http://homepages.cwi.nl/~manegold/Calibrator/calibrator.shtml`.

[7] Larry McVoy and Carl Staelin. MOB: Memory Organization Benchmark. `http://www.bitmover.com/lmbench/`.

[8] Larry McVoy and Carl Staelin. lmbench: Portable tools for performance analysis. In *USENIX 1996 Annual Technical Conference, January 22–26, 1996. San Diego, CA*, pages 279–294, Berkeley, CA, USA, January 1996.

[9] Juan Navarro, Sitararn Iyer, Peter Druschel, and Alan Cox. Practical, transparent operating system support for superpages. *SIGOPS Oper. Syst. Rev.*, 36(SI):89–104, 2002.

[10] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[11] Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect of benchmark run. Technical Report CSD-93-767, February 1993.

[12] Carl Staelin and Larry McVoy. mhz: Anatomy of a micro-benchmark. In *USENIX 1998 Annual Technical Conference, January 15–18, 1998. New Orleans, Louisiana*, pages 155–166, Berkeley, CA, USA, June 1998.

[13] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001. Also available as University of Tennessee LAPACK Working Note #147, UT-CS-00-448, 2000 (`www.netlib.org/lapack/lawns/lawn147.ps`).

[14] Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[15] Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-Ray: Automatic measurement of hardware parameters. Technical Report TR2004-1966, October 2004.