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

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 poorly documented if at all. Moreover, there is growing interest in self-tuning, autonomic software systems that can optimize themselves for different platforms; 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 parameters of all levels of the memory hierarchy, including registers, all data caches and the translation look-aside buffer. We have implemented these micro-benchmarks in a tool called X-Ray that can be ported easily to new platforms. We present experimental results that show that X-Ray successfully determines memory hierarchy parameters on current platforms, and compare its accuracy with that of existing tools.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Measurement techniques; D.1 [**Programming Techniques**]: Automatic Programming

General Terms

Algorithms, Experimentation, Measurement, Performance

Keywords

Micro-benchmarks, Hardware parameters, Memory hierarchy, Caches, Measurement, Optimization, Self-tuning, Autonomic systems

*This work was supported by an IBM Faculty Partnership Award, DARPA grant NBCH30390004, and by NSF grants ACI-0085969, ACI-0090217, ACI-0103723, ACI-0121401, and ACI-0406345.

SIGMETRICS'05, June 6–10, 2005, Banff, Alberta, Canada.

1. INTRODUCTION

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 [17].

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 a detailed specification of the platform. In practice, cache parameters 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 [4]. 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 by 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. Furthermore, some registers are usually reserved by compilers for the stack pointer, frame base register, etc. Therefore, it is useful to have micro-benchmarks to determine memory hierarchy parameter values relevant to program optimization.

The need for such benchmarks is 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 [16], which is a portable system that produces highly tuned linear algebra libraries, and FFTW [6] and Spiral [13], which are similar systems for generating digital signal processing libraries. When installed on a new machine, these systems execute a set of micro-benchmarks to determine the hardware parameters of the machine, and use these values to determine optimal values for various software parameters. Some of these systems, such as ATLAS, use global search, so they use the hardware parameter values only to guide the search process. Other systems use model-driven optimization to directly estimate optimal values for software parameters, given values of the hardware parameters. Such systems

obviously require very accurate estimates of hardware parameter values; in fact, the work reported in this paper was motivated by the inadequacies of existing micro-benchmarks for building a model-driven version of ATLAS [17]. Therefore, accurate micro-benchmarks are key to the success of self-optimizing software.

In this paper, we present micro-benchmarks for measuring the parameters of the memory hierarchy of a platform, including registers, all data cache levels, and the TLB. Existing tools such as lmbench [10] and Calibrator [9] measure some of these memory hierarchy parameters, but our experiments show that none of them offer the same parameter coverage or accuracy as our micro-benchmarks. These tools implement variations of the micro-benchmark developed by Saavedra [14], which is reproduced by Hennessy and Patterson [7] 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 Calibrator and lmbench can determine some cache parameters automatically from these timing results, none of them measures 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 on architectures like the IBM Power can compromise the timing measurements. Other tools use hardware performance counters to probe the memory hierarchy [3, 5], but these tools have portability problems.

The key difference between our approach and previous approaches is that our micro-benchmarks are designed so that when the parameters of cache $\mathcal{C}_i$ at level i are being measured, higher level caches $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{i-1}$ are "transparent" in the sense that memory accesses relevant to the measurements are guaranteed to miss in those caches. This isolation permits us to measure the associativity of caches directly, which existing micro-benchmarks cannot do. This in turn permits us to measure cache capacity accurately even when the associativity and therefore the capacity are not powers of 2 (for example the Itanium L3 cache has an associativity of 24); in contrast, most existing micro-benchmarks can only handle cache capacities that are powers of 2.

The rest of this paper is organized as follows. In Section 2, we discuss existing approaches and their drawbacks. In Section 3, we introduce the memory reference patterns used in our micro-benchmarks, and prove some important properties of these patterns. In Section 4, we present our micro-benchmarks for measuring L1 data cache parameters. In Section 5, we show how to measure the parameters of lower level caches without interference from higher level caches. In Sections 6 and 7, we show how to measure some TLB parameters and the number of registers. We present experimental results in Section 8 and discuss ongoing work in Section 9.

2. PREVIOUS APPROACHES

The most widely used micro-benchmark for these measurements is the benchmark of Saavedra [14], a stylized version of which is presented in Figure 1. This benchmark makes fixed-stride accesses to the elements of a large array in memory, and measures the average time per access. These timing results are then interpreted to determine cache parameters. We make the following observations.

1. The benchmark performs series of experiments for pairs $\langle$`csize`, `stride`$\rangle$, where the array size (`csize`) varies between `CACHE_MIN` and `CACHE_MAX` and `stride` varies between 1 and `csize`. Both are powers of 2.

2. For each $\langle$`csize`, `stride`$\rangle$, the benchmark traverses the array `x` with the specified stride `SAMPLE` $\times$ `steps` times to ensure that the total `time` spent is at least 1 sec.

3. The measurement for $\langle$`csize`, `stride`$\rangle$ is repeated the same number of times, replacing references to the array `x` with references to a single scalar variable `temp`.

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

1. Algorithmic Level

   (a) 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.

   (b) The benchmark considers all levels of the memory hierarchy simultaneously, so each timing result is possibly influenced by several parameters from different cache levels. Interpretation of the timing results is complex.

   (c) The benchmark uses only array sizes restricted to powers of 2, which prevents it from measuring cache capacities that are not powers of 2.

2. Implementation Level

   (a) The source code uses a very complex loop structure, which is the source of substantial loop overhead. An attempt is made to account for that overhead by measuring and subtracting the execution time of a cloned version the same loop structure that 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 versions, yielding inaccurate results.

   (b) All memory accesses are independent, which allows an aggressive optimizing compiler to schedule them in a way so that some overlap. This complicates the interpretation of timing results.

   (c) The addressing mode used to access array elements involves both a base address and an offset. On many RISC architectures, this operation requires an extra address computation instruction before the actual memory access instruction is performed.

   (d) The source code does not use the values of accessed array elements (and more importantly, the value of the `temp` variable) for producing output, so a smart optimizing compiler can eliminate portions of the code.

```
#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, tsteps = 0, 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

(e) There is a constant stride between successive accesses of array elements. The IBM Power 3, among others, performs hardware prefetching for such fixed stride accesses, which interferes which the timing measurements.

(f) A key assumption is that the array `x` is stored in a contiguous set of memory locations. In reality, it is only guaranteed to be contiguous in the logical address space of the processor, and it can be fragmented in the physical address space. In most processors, lower level caches are physically addressed, so the contiguity assumption for array elements is violated.

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 different at the algorithmic level, and it eliminates these problems.

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.

We use the Intel P6 (Pentium Pro/II/III) as an example. Figure 2 shows the structure of a memory address and Figure 3 shows the structure of the L1 data cache, which on these machines is organized as $\langle A, B, C\rangle = \langle 4, 32, 16\text{KB}\rangle$.

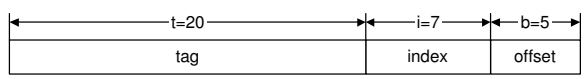


Figure 2: P6 memory address structure

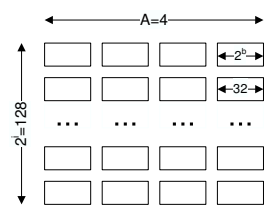


Figure 3: P6 L1 data cache structure

Therefore the cache contains $\frac{C}{B} = \frac{16384}{32} = 512$ individual blocks, divided into $\frac{512}{A} = \frac{512}{4} = 128$ 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 specify the offset of a particular byte within the 32-byte block. Note that $C = A \times 2^{i+b}$.

Definition 1. *For a cache with associativity A and capacity C, we define the* stride *of that cache as $T \equiv \frac{C}{A} = 2^{i+b}$.*

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. □

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,*

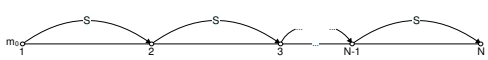
$$\mathit{indices}(W) \equiv \{i : \mathit{project}_i(W) \neq \emptyset\}$$

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 (e.g., the L1 data cache of IBM Power 3 uses FIFO replacement policy).

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 4(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 multi-level memory hierarchies, our micro-benchmarks use sequences of sequences, as shown in Figure 4(b). One way to think about these sequences of sequences is to imagine each of the N points in the sequence of Figure 4(a) being expanded into a sequence of n points (superpose Figures 4(a),4(b)). To represent them we use the notation $W = \langle\langle m_0, s, n\rangle, S, N\rangle$. Like S, the small stride s is always a power of 2.



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

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

Figure 4: 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 $\langle m_0 + i \times S, s, n\rangle$ an *inner* subsequence of $\langle\langle m_0, s, n\rangle, S, N\rangle$. Notice that the sequence of sequences $\langle\langle m_0, s, n\rangle, S, N\rangle$ can also be expressed as $\langle\langle m_0, S, N\rangle, s, n\rangle$. This property is expressed in Lemma 2.

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

PROOF. Omitted. □

We mention that a sequence of sequences can be viewed as the convolution of two sequences. Lemma 2 follows from the well-known fact that convolution is commutative.

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, block size B, and an address m_0 aligned on a cache block boundary, the half-open interval $[m_0, m_0 + C)$ is 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 $W' = \left\langle m_0, B, \frac{C}{B}\right\rangle$, in which only one address is mapped to a single cache line. Furthermore 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 $w'_i = \left\langle m_0 + i \times T, B, \frac{T}{B}\right\rangle$ map exactly one element to each cache set. Therefore 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

Figure 5 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 (from Lemma 3(a,b)). 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 is semi-compact as it 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.

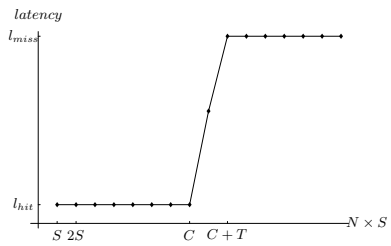


Figure 5: Example of (semi-/non-)compact

Theorem 1 describes the necessary and sufficient conditions for compactness and non-compactness of a sequence with respect to a given cache. Intuitively, the number of cache sets occupied by elements of a large enough sequence of stride $S \geq B$ is $\lceil \frac{T}{S} \rceil$. The number of elements mapped to two cache sets differ by at most one. Therefore, the sequence is compact if $N \leq A \lceil \frac{T}{S} \rceil$, and non-compact if $N \geq (A+1) \lceil \frac{T}{S} \rceil$. The proof given below establishes the iff condition, and also covers the case when $S < B$.

THEOREM 1. *Consider a cache with parameters $\langle A, B, C \rangle$ and a sequence $W = \langle m_0, S, N \rangle$, where m_0 is aligned on a cache block boundary.*

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

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

PROOF.

- $S \geq T$. 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 $N \geq A + 1 = N_{nc}$.

- $B \leq S < T$, and let $k = \frac{T}{S}$ and $N = p \times k + r$, where $0 \leq r < k$. Therefore:

$$\begin{aligned} W &= \langle m_0, S, p \times k + r \rangle \\ &= \langle m_0, S, p \times k \rangle \cup \langle m_0 + S \times p \times k, S, r \rangle \\ &= \langle \langle m_0, T, p \rangle, S, k \rangle \cup \langle m_0 + p \times T, S, r \rangle \end{aligned}$$

  From Lemma 1 follows that:

  1. Each inner subsequence of $\langle \langle m_0, T, p \rangle, S, k \rangle$ maps exactly p elements to a single cache set;
  2. The k inner subsequences of $\langle \langle m_0, T, p \rangle, S, k \rangle$ map to k different cache sets;
  3. The r elements of the sequence $\langle m_0 + p \times T, S, r \rangle$ map to the same cache sets as the first r of the k inner subsequences of $\langle \langle m_0, T, p \rangle, S, k \rangle$.

  We now look at the following subcases:

  - For $N = N_c = k \times A$, $p = A$ and $r = 0$. We have A different elements of W mapped to each of k different cache sets, and therefore W is compact.
  - For $N < N_c$, W is compact by Lemma 3(b).
  - For $N = N_{nc} = k \times (A+1)$, $p = A+1$ and $r = 0$. We have exactly $(A+1)$ different elements of W mapped to each of k different cache sets, and therefore W is non-compact.
  - For $N > N_{nc}$ we have at least $(A+1)$ different elements of W mapped to each of k different cache sets, and therefore W is non-compact.
  - For $N_c < N < N_{nc}$, $p = A$ and $0 < r < k$. We have exactly $(A+1)$ different elements of W mapped to r of the k cache sets and exactly A different elements of W mapped to $k - r$ of the k cache sets. Therefore W is semi-compact.

  The required result follows directly from this.

- $S < B$, and let $k = \frac{B}{S}$. In this case, groups of k consecutive elements of W map to the same cache line, and therefore:

$$\mathsf{compact}\,(W) \Leftrightarrow \mathsf{compact}\,(\langle m_0, S, N' \rangle) \Leftrightarrow$$
$$\Leftrightarrow \mathsf{compact}\left(W' = \left\langle m_0, B, \frac{N'}{k} \right\rangle\right).$$

  N' is the smallest multiple of k, such that $N' \geq N$. Since we already proved the theorem for W',

$$\mathsf{compact}\,(W) \Leftrightarrow \mathsf{compact}\,(W') \Leftrightarrow$$
$$\Leftrightarrow \frac{N'}{k} \leq A\frac{T}{B} \Leftrightarrow N' \leq k \times A\frac{T}{B} \Leftrightarrow N \leq N_c = A \left\lceil \frac{T}{S} \right\rceil.$$

  By analogy, if N'' is the largest multiple of k, such that $N'' \leq N$,

$$\mathsf{non\text{-}compact}\,(W) \Leftrightarrow \mathsf{non\text{-}compact}\,(\langle m_0, S, N'' \rangle) \Leftrightarrow$$
$$\Leftrightarrow \mathsf{non\text{-}compact}\left(W'' = \left\langle m_0, B, \frac{N''}{k} \right\rangle\right) \Leftrightarrow$$
$$\Leftrightarrow \frac{N''}{k} \geq (A+1)\frac{T}{B} \Leftrightarrow N'' \geq k \times (A+1)\frac{T}{B} \Leftrightarrow$$
$$\Leftrightarrow N \geq N_{nc} = (A+1) \left\lceil \frac{T}{S} \right\rceil.$$

□

4.1 Algorithms for Measuring Parameters

In this section we use the function $\mathsf{is\_compact}\,(W)$ to determine empirically if W is compact. Its implementation is discussed in Section 4.2.

4.1.1 Cache Latency

We determine l_{hit} by measuring the average time 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 a cache, for which pseudo-code is presented 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 double N until the sequence is not compact. Let N_{old} be 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, $A = N - 1$ and $C = \frac{S}{2} \times A$.

```
S ← 1;
N ← 1;
while (is_compact(⟨m0, S, N⟩))
    N ← 2 × N;
repeat
    S ← 2 × S;
    N_old ← N;
    N ← min N' ∈ [1, N_old] : ¬is_compact(⟨m0, S, N'⟩);
until (N = N_old);
A ← N − 1;
C ← S/2 × A;
```

Figure 6: Measuring C and A of L1 Data Cache

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

Note that towards the end of the execution of the algorithm, the number of distinct addresses accessed is on the order of the associativity of the cache. Non-compactness of these sequences results in a very pronounced increase in average access time, enabling our approach to produce more accurate results than those obtained using other approaches.

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 $\langle\langle m_0, T, A\rangle, C + \delta, 2\rangle$.

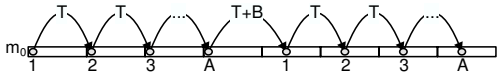


Figure 7: Measuring B of L1 data cache

The addresses in each of the two inner subsequences map to a single cache set. This cache set is the same for both inner subsequences when $0 \leq \delta < B$, and is different when $\delta \geq B$. 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.

```
δ ← 1
while (¬is_compact(⟨⟨m0, T, A⟩, C + δ, 2⟩))
    δ ← 2 × δ;
return δ;
```

Figure 8: Algorithm for measuring B

4.2 Implementation of is_compact

The algorithms in Section 4.1 call is_compact (W) to determine experimentally whether sequence 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 enough" to the cache hit latency l_{hit}, which is measured as described in Section 4.1.1. Because of noise and timer inaccuracies, l may not be exactly equal to l_{hit} even if the set is compact. X-Ray currently assumes that the miss latency of each cache level is at least twice the hit latency at that level; therefore it declares that a sequence is not compact if the average time per access is at least twice greater than the hit latency. Intuitively, this means that the time per access has to double before the difference is considered "significant". We are currently evaluating more statistically sound strategies.

We now address the practical problems discussed in Section 2. In our implementation, 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 offline, *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. Additionally, in the implementation, the `while` loop is unrolled several times to avoid loop overhead.

```
startTime ← get_time();
while (--R)
    p ← *(void **)p;
timePerAccess ← (get_time() − startTime) ÷ 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.

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., "`lwz 11,0(11)`" on the PowerPC 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; however, the actual order in which we access them is irrelevant. To prevent hardware constant stride prefetchers such as those on the IBM Power architecture from interfering with our timing, we initialize the array elements by chaining the pointers so that we visit the elements in a pseudo-random order.

Consider the address sequence $m_0, m_1, \ldots, m_{n-1}$. One way to reorder this sequence is to choose a number p, such that p and n are mutually prime. After visiting element m_i, we 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 $i \leftarrow (i+p) \text{ modulo } n$ is guaranteed to generate all the integers between 0 and $n-1$ before repeating itself.

Note that this technique does not address problems posed by Markov predictors [8] and Load Value predictors [2]. However, to the best of our knowledge, these mechanisms have not yet been implemented in a commercial processor.

(g) All modern processors have virtually indexed L1 data caches and therefore physical contiguity of the array is not an issue. Lower levels of the memory hierarchy are usually physically indexed, so physical contiguity is important for lower levels of the memory hierarchy, as we discuss in Section 5.3.

5. LOWER LEVELS OF CACHE

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 $\text{compact}_i(W)$ denotes that $\text{compact}(W)$ 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 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 a sequence W into a new sequence W^* with the following properties.

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

Intuitively, W is of the form presented in Figure 4(a). We want to transform it to W^*, which is a sequence of sequences of the form presented in Figure 4(b), so that the extra memory accesses exhaust the associativity of caches 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 examine the sequence $W = \langle m_0, 2\text{MB}, 9\rangle$ and declare 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 this 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 it is easy to verify that together, they force non-compactness with respect to $\mathcal{C}_1$.

The constructions in this section assume that (i) each cache level is at least twice bigger than the level immediately above it, and (ii) the stride of any cache level is at least as large as the block size of any other cache level. These can be formally specified as $C_i \geq 2C_{i-1}$ and $T_i \geq B_j$. To the best of our knowledge, these assumptions are satisfied by all current machines. Furthermore, the measurements assume that $l_i \geq 2l_{i-1}$ to detect changes in compactness of sequences as described in Section 4.2.

The constructions rely on a generalization of Theorem 1 to sequences of sequences, which is presented in Theorem 2. To prove it, we need the following lemma.

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

PROOF. If not, there must be elements $e_1 = m_0 + p \times S$ and $e_2 = m_0 + \delta + q \times S$ in sequences W_1 and W_2 respectively such that $\text{index}(e_1) = \text{index}(e_2)$.

Since m_0 and $m_0 + \delta$ are aligned on a cache block boundary, both m_0 and δ must be non-negative multiples of the block size B. Since $B \leq \delta < S$ and both B and S are powers of 2, S must be a multiple of B. Therefore, e_1 and e_2 must be multiples of B. If they map to the same cache index, these two addresses must differ by some multiple of the cache stride T. Therefore, $m_0 + p \times S = m_0 + \delta + q \times S + k \times T$ for some integers p, q, and k. This can be simplified to $\delta = (p - q) \times S - k \times T$. Since both S and T are powers of 2, it follows that δ is a multiple of $\min(S, T)$. This means that either $\delta = 0$ or $\delta \geq \min(S, T)$. Either way, this contradicts the assumption that $0 < \delta < \min(S, T)$. Therefore, $\text{indices}(W_1)$ and $\text{indices}(W_2)$ are disjoint. □

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

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

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 inner subsequences $W_i = \langle m_0 + i \times s, S, N\rangle$ 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, $\text{indices}(W_i)$ are pairwise disjoint sets for all $i \in [0, n-1]$. The required result follows from Lemma 3. □

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 $\text{compact}_2(W^*) \Leftrightarrow \text{compact}_2(W)$ and $\text{non-compact}_1(W^*)$.

To construct W^*, we restrict ourselves to sequences for which $S \leq 2T$ (because $2T$ is the largest stride used by these algorithms). Furthermore, because we assume $C_2 \geq 2C_1$, we can assume $\mathsf{compact}_2(W)$ if $(N-1) \times S < 2C_1$. Therefore we can restrict ourselves to sequences for which $(N-1) \times S \geq 2C_1$.

LEMMA 5. *Let $W = \langle m_0, S, N \rangle$ be a sequence in which $S \leq 2T_2$ and $(N-1) \times S \geq 2C_1$, where T_2 is the stride of L2 cache, and C_1 is the capacity of the L1 cache. Let $W^* = \langle \langle m_0, s, n \rangle, S, N \rangle$, where*

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

The following properties hold:

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

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

PROOF. First we show that

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

The opposite is impossible, because

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

Therefore each inner subsequence of W^* is properly contained between successive elements of the sequence W.

(a) From $(n-1) \times s < \frac{S}{2}$ and $S \leq 2T_2$ we conclude that $(n-1) \times s < \min(S, T_2)$.

From Theorem 2, $\mathsf{compact}_2(W^*) \Leftrightarrow N \leq A_2 \left\lceil \frac{T_2}{S} \right\rceil$. From Theorem 1, $\mathsf{compact}_2(W) \Leftrightarrow N \leq A_2 \left\lceil \frac{T_2}{S} \right\rceil$. Therefore $\mathsf{compact}_2(W^*) \Leftrightarrow \mathsf{compact}_2(W)$.

(b)
- $S > T_1$. Since $s = T_1$, all $n \times N$ elements of W^* map to the same L1 cache set. W^* is non-compact because $n \times N = \left\lceil \frac{A_1+1}{N} \right\rceil \times N \geq (A_1+1)$.
- $S \leq T_1$. Because of the proper nesting, $n = 1$, i.e., $W = W^*$. $N \geq \frac{2C_1}{S} + 1 > \frac{2A_1 \times T_1}{S} \geq (A_1+1)\frac{T_1}{S} = (A_1+1)\left\lceil \frac{T_1}{S} \right\rceil$, and by Theorem 1 $\mathsf{non\text{-}compact}_1(W^*)$.

□

5.2 Multiple Cache Levels

To generalize the approach from Section 5.1 to multiple cache levels $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_l$ we replace $W = \langle m_0, S, N \rangle$ with $W^* = \langle \langle m_0, s, n \rangle, S, N \rangle$, where W^* is constructed by considering caches $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_{l-1}$. Specifically, we choose

$$\begin{aligned} s &= \min_{i \in I} T_i \\ n &= \max_{i \in I} \left(\left\lceil \frac{A_i+1}{N} \right\rceil \times \frac{T_i}{s} \right), \end{aligned}$$

and $I = \{i \in [1, l) : T_i < S\}$.

LEMMA 6. *If $S \leq 2T_l$ and $(N-1) \times S > 2C_i$ for all $i \in [1, l)$ then*

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

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

PROOF. $\left(\left\lceil \frac{A_i+1}{N} \right\rceil - 1 \right) \times T_i < \frac{S}{2}$ for all $i \in [1, l)$ (by analogy with Inequality (1)). For $i \in I$ we have $T_i \leq \frac{S}{2}$ and therefore:

$$\left\lceil \frac{A_i+1}{N} \right\rceil - 1 < \frac{S}{2T_i} \Rightarrow \left\lceil \frac{A_i+1}{N} \right\rceil T_i \leq \frac{S}{2}.$$

Therefore:

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

Therefore each inner subsequence of W^* is properly contained between successive elements of the sequence W.

(a) From $(n-1) \times s < \frac{S}{2}$ and $S \leq 2T_i$ we conclude that $(n-1) \times s < \min(S, T_i)$.

From Theorem 2, $\mathsf{compact}_l(W^*) \Leftrightarrow N \leq A_l \times \left\lceil \frac{T_l}{S} \right\rceil$. From Theorem 1, $\mathsf{compact}_l(W) \Leftrightarrow N \leq A_l \times \left\lceil \frac{T_l}{S} \right\rceil$. Therefore $\mathsf{compact}_l(W^*) \Leftrightarrow \mathsf{compact}_l(W)$.

(b)
- $i \in I$. Consider

$$\begin{aligned} W' &= \left\langle \left\langle m_0, s, \left\lceil \frac{A_i+1}{N} \right\rceil \frac{T_i}{s} \right\rangle, S, N \right\rangle \\ &= \left\langle \left\langle \left\langle m_0, s, \frac{T_i}{s} \right\rangle, T_i, \left\lceil \frac{A_i+1}{N} \right\rceil \right\rangle, S, N \right\rangle \\ &= \left\langle \left\langle \left\langle m_0, T_i, \left\lceil \frac{A_i+1}{N} \right\rceil \right\rangle, S, N \right\rangle, s, \frac{T_i}{s} \right\rangle. \end{aligned}$$

By Lemma 5(b), applied to cache levels i and l, all inner subsequences of W' are non-compact with respect to $\mathcal{C}_i$. Therefore, by Lemma 3(e), $\mathsf{non\text{-}compact}_i(W')$.
$W^* = \left\langle \left\langle m_0, s, \max_{i \in I} \left(\left\lceil \frac{A_i+1}{N} \right\rceil \frac{T_i}{s} \right) \right\rangle, S, N \right\rangle \supseteq W'$, and with respect to $\mathcal{C}_i$, $\mathsf{indices}(W^*) = \mathsf{indices}(W')$. Therefore $\mathsf{non\text{-}compact}_i(W^*)$.
- $i \notin I$. $W^* = \langle \langle m_0, s, n \rangle, S, N \rangle = \langle \langle m_0, S, N \rangle, s, n \rangle$. A proof similar to that of Lemma 5(b) shows that all inner subsequences of the latter are non-compact with respect to $\mathcal{C}_i$. By Lemma 3(e), $\mathsf{non\text{-}compact}_i(W^*)$ holds.

□

5.3 Algorithms for Measuring Parameters

We use the algorithms in Section 4.1 to measure latency, capacity and associativity at lower cache level, by substituting $\mathsf{is\_compact}(W)$ with $\mathsf{is\_compact}_i(W^*)$, which 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 $\mathcal{C}_i$). If $(N-1) \times S < 2C_{i-1}$, the implementation does not perform any measurements but simply assumes that the sequence is compact.

One major complication when measuring parameters of lower level caches is that on modern platforms $\mathcal{C}_1$ is typically virtually indexed, but lower levels are always physically indexed. This is a problem because contiguity in virtual memory is not a sufficient condition for contiguity 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 level caches 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 super-page.

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, we provide OS-specific memory allocation and deallocation routines, which are then used by the cache micro-benchmarks 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 [12]. 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.

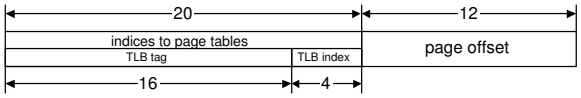


Figure 10: Memory address decomposition on P6

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, [12] 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.

3. *Ensuring TLB access*: As in the case of lower level caches, 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 the TLB stores memory protection information which is needed to complete the particular memory operation.

4. *Physical contiguity*: As with lower level caches, 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.

Because of these complications, 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 below.

For a sequences $W = \langle m_0, S, N \rangle$, let $N = p \times \lceil \frac{T_1}{S} \rceil + r$, where $0 \leq r < \lceil \frac{T_1}{S} \rceil$. 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. $\frac{C_1}{B_1} \geq 2\frac{C_{TLB}}{B_{TLB}}$, which is true for all modern platforms today. Under this assumption, it is easy to see that $\mathsf{compact}_1(W^*)$.

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]. We will conduct experiments on other platforms in the future.

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.).

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

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., in ATLAS [16]).

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.

$$
\begin{aligned}
&r_0 \leftarrow \mathsf{add}\,(r_0, r_n)\,;\\
&r_1 \leftarrow \mathsf{add}\,(r_1, r_0)\,;\\
&r_2 \leftarrow \mathsf{add}\,(r_2, r_1)\,;\\
&\ldots\\
&r_n \leftarrow \mathsf{add}\,(r_n, r_{n-1})\,;
\end{aligned}
$$

Figure 11: Sequence with n variables

The form of the sequences we use is shown 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 therefore 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 $[n_{max}/2, n_{max})$.

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.

Neither lmbench nor Calibrator 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.

Architecture	available / actual			
	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

Table 1 shows that on some platforms, the number of registers measured by X-Ray is different than the number of architected registers. In each case, we verified that the difference arose because some registers are reserved by the architecture or the compiler for some special use. In particular, the number of available integer registers is smaller than the actual number on all platforms because integer registers may be used to hold the values of the program counter, stack pointer, frame base register etc. 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.

Measurements on the Itanium 2 illustrate a different point. This processor has 128 floating-point registers but two of these registers are hardwired to 0.0 and 1.0. In spite of this, X-Ray concluded that the Itanium has 128 available registers, because the average access time did not increase significantly until three or more variables were spilled. Reducing the significance threshold used by X-Ray may permit a more accurate measurement but this increases sensitivity to noise. In any case, the difference may be irrelevant to software because the results suggest that the software can assume that there are 128 available floating-point registers without significant loss of performance from register spills.

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 [18]. 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 from running the memory-hierarchy portion of X-Ray on 7 platforms with the results from running *Calibrator* v0.9e [9] and *lmbench* v3.0a3 [10, 11, 15].

Because all the tools, including X-Ray, measure hardware parameters empirically, the results sometimes vary from one execution to the next. These variations are negligible for X-Ray, but quite noticeable sometimes 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. When a parameter is not successfully measured by a tool, 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;
- **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. 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 5.3. 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. The Itanium was the only machine in our study that has an L3 cache. Table 3 shows the results of these measurements.

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 is also cached in all lower levels. AMD machines on the other hand use exclusion, so data never resides in both the L1 and L2 caches simultaneously. This 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 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.

	Architecture	**Actual**	**X-Ray**	**Calibrator**	**lmbench**
L1 Data Cache Capacity (KB)	Pentium 4	8	8	8	8
	Itanium 2	16	16	16	abort
	Athlon MP	64	64	64	empty
	Opteron 240	64	64	64	abort
	UltraSPARC IIIi	64	64	64	64
	R12000	32	32	32	32
	Power 3	64	64.5	64	64
L1 Data Cache Block Size (bytes)	Pentium 4	64	64	32	64
	Itanium 2	64	64	64	abort
	Athlon MP	64	64	64	empty
	Opteron 240	64	64	32	abort
	UltraSPARC IIIi	32	32	32	32
	R12000	16	16	64	32
	Power 3	128	128	128	128
L1 Data Cache Associativity (count)	Pentium 4	4	4	n/a	n/a
	Itanium 2	4	4	n/a	n/a
	Athlon MP	2	2	n/a	n/a
	Opteron 240	2	2	n/a	n/a
	UltraSPARC IIIi	4	4	n/a	n/a
	R12000	2	2	n/a	n/a
	Power 3	128	129	n/a	n/a
L1 Data Cache Hit Latency (cycles)	Pentium 4	2	4.32	2.02	2.06
	Itanium 2	2	1.99	2	abort
	Athlon MP	3	3.02	3.17	empty
	Opteron 240	3	3	3.02	abort
	UltraSPARC IIIi	2	2	1.99	2
	R12000	2	2.02	2.07	2.01
	Power 3	2	2.01	2	2.01
L2 Data Cache Capacity (KB)	Pentium 4	512	512	384	512
	Itanium 2	256	256	256	abort
	Athlon MP	512	576	384	512
	Opteron 240	1024	1088	768	abort
	UltraSPARC IIIi	512	os	1024	1024
	R12000	512	os	2048	2048
	Power 3	512	os	6144	6144
L2 Data Cache Block Size (bytes)	Pentium 4	128	128	128	128
	Itanium 2	128	128	128	128
	Athlon MP	64	64	64	64
	Opteron 240	64	64	64	64
	UltraSPARC IIIi	64	os	64	64
	R12000	128	os	128	128
	Power 3	128	os	128	128
L2 Data Cache Associativity (count)	Pentium 4	8	8	n/a	n/a
	Itanium 2	8	8	n/a	n/a
	Athlon MP	16	18	n/a	n/a
	Opteron 240	16	17	n/a	n/a
	UltraSPARC IIIi	?	os	n/a	n/a
	R12000	?	os	n/a	n/a
	Power 3	?	os	n/a	n/a
L2 Data Cache Hit Latency (cycles)	Pentium 4	?	41.52	17.75	20.36
	Itanium 2	?	5.98	4.16	abort
	Athlon MP	?	36	18.25	2.69
	Opteron 240	?	22.8	13.13	abort
	UltraSPARC IIIi	?	12.89	12.41	15.15
	R12000	?	13.69	11.94	13.86
	Power 3	?	18.13	8.52	17.19
Main Memory Latency (cycles)	Pentium 4	?	761.92	372.42	368.31
	Itanium 2	?	297.65	281.45	abort
	Athlon MP	?	471.35	400.57	197.58
	Opteron 240	?	136.21	126.81	abort
	UltraSPARC IIIi	?	os	164	172.81
	R12000	?	os	110.92	122.06
	Power 3	?	os	136.22	160.84

Table 2: Summary of experimental results

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

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.

Experimental results for measuring TLB parameters and number of registers were discussed earlier 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 some TLB parameters and the number of registers. The experimental results show that our approach automatically measures more cache parameters with greater precision than existing approaches do. 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 accomplish this, our micro-benchmarks use 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 [18]. X-Ray can also measure hardware parameters such as the CPU frequency, instruction latency, throughput, and existence, 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 measuring parameters of victim and instruction caches, improving the quality of measuring TLB parameters, measuring other cache parameters such as bandwidth, parallelism, write mode, and sharedness, and implementing OS support for Solaris, AIX, etc. We are also investigating more statistically sound approaches for determining when jumps occur in the timing measurements.

X-Ray is freely available at `http://iss.cs.cornell.edu/Software/X-Ray.aspx`.

10. REFERENCES

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

[2] Martin Burtscher and Benjamin G. Zorn. Hybrid load-value predictors. *IEEE Trans. Comput.*, 51(7):759–774, 2002.

[3] C.L. Coleman and J.W. Davidson. Automatic memory hierarchy characterization. *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 103–110, 2001.

[4] 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.

[5] Jack Dongarra, Shirley Moore, Phil Mucci, Keith Seymour, and Haihang You. Accurate cache and TLB characterization using hardware counters. In *Proceedings of the International Conference on Computational Science (ICCS) 2004, Krakow, Poland*, 2004.

[6] 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".

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

[8] Doug Joseph and Dirk Grunwald. Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133, 1999.

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

[10] Larry McVoy and Carl Staelin. lmbench. `http://www.bitmover.com/lmbench/`.

[11] 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.

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

[13] 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".

[14] 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.

[15] 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.

[16] 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.

[17] 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".

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