# X-Ray : Automatic Measurement of Hardware Parameters*

Kamen Yotov, Keshav Pingali, Paul Stodghill,

Department of Computer Science,

Cornell University,

Ithaca, NY 14853.

October 6, 2004

**Abstract**

There is growing interest in autonomic, self-tuning software that can optimize itself on new platforms, without manual intervention. Optimization requires detailed knowledge of the target platform such as the latency and throughput of instructions, the numbers of registers, and the organization of the memory hierarchy. An autonomic optimization system needs to determine such platform-specific information on its own.

In this paper, we describe the design and implementation of X-Ray, which is a tool that automatically measures a large number of such platform-specific parameters. For some of these parameters, we also describe novel algorithms, which are more robust than existing ones. X-Ray is written in C for maximum portability, and it is based on accurate timing of a number of carefully designed micro-benchmarks. A novel feature of X-Ray is that it is easily extensible because it provides simple infrastructure and a code generator that can be used to produce the large number of micro-benchmarks needed for such measurements.

There are few existing tools that address this problem. Our experiments show that X-Ray produces more accurate and more complete results than any of them.

## 1 Introduction

There is growing interest in the design of self-optimizing computer systems that can improve their performance without human intervention. Some of these systems such as ATLAS [12] and FFTW [5] optimize themselves statically at installation time, while other systems such as those envisioned in IBM's autonomic systems initiative [1] optimize their performance continuously during execution.

In many self-optimizing systems, the process of optimization involves choosing optimal values for certain performance-critical parameters. For example, when ATLAS, a portable system for generating numerical linear algebra libraries, is installed on a machine, it performs a series of experiments to determine optimal values for software parameters like

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

loop tile sizes, and loop unroll factors, and then generates appropriate code with these parameter values. Although the optimal values of the parameters depend on hardware resources like cache capacity, number of registers etc. and therefore vary from machine to machine, ATLAS is able to adapt itself automatically in this way to each platform without human intervention.

In general, the space of possible values for optimization parameters is too large to be explored exhaustively, so it is necessary to limit the search for optimal parameter values. For example, to find the optimal cache tile size, it is sufficient to limit the search to tile sizes that are smaller than the capacity of the cache; larger tiles will not fit in the cache and cannot be optimal. Similarly the search space for register tiles can be bounded by the number of registers in the processor. In general therefore, bounding the search space of optimization parameters requires knowledge of hardware parameters such as the capacity of the cache, and the number of registers. Furthermore, for some programs, analytical models can be used to compute optimal values for code optimization parameters; not surprisingly, these models may require more detailed and accurate information about the hardware, such as the line size and associativity of various levels of the memory hierarchy [13].

For software to be self-tuning and self-optimizing, it is necessary that these hardware parameters be determined *automatically* without human intervention. On some machines, it may be possible to determine some of this information by reading special hardware registers or records in the operating system [3]. However, most processors and operating systems do not support such mechanisms or provide very limited support; for example, we do not know of any platform on which the latency and throughput of instructions can be determined in this manner.

An alternative approach is to use carefully crafted micro-benchmarks to measure hardware parameters automatically. Perhaps the most well-known micro-benchmark is the Hennessy-Patterson benchmark [6] that determines memory hierarchy parameters by measuring the time required to access a series of array elements with different strides. However, the timing results from this benchmark must be analyzed manually to determine memory hierarchy parameters, so it is not suitable for incorporation into an autonomic system. Furthermore, the benchmark makes fixed stride accesses to memory, so hardware pre-fetching on machines like the Power 3 can corrupt the timing results. Finally, this benchmark does not determine the latency and throughput of instructions, or whether instructions like fused multiply-add exist. Some of these problems were addressed in more recent tools like Calibrator [7], lmbench [8], and MOB [2], but our experience with these tools is somewhat mixed, as we describe in Section 5.

This paper makes the following contributions.

- We describe a robust, easily extensible framework for implementing micro-benchmarks to measure hardware parameters.
- We design a number of novel algorithms for measuring memory hierarchy parameters and other architectural features, and show how they can be implemented in this framework.
- We describe a tool called X-Ray, which is an implementation of these ideas, and evaluate its capabilities on seven modern architectures. We show that it produces more accurate and complete results than existing tools.

For portability, X-Ray is implemented completely in C [4]. One of the interesting challenges of this approach is to ensure that the C compiler does not perform any high-level optimizations on our benchmarks that might pollute the timing results.

The rest of this paper is organized as follows. In Section 2, we give an overview of the X-Ray system and the techniques that it uses for generating and timing micro-benchmark programs in C. Section 3 discusses the micro-benchmark algorithms that we have developed for determining features of the CPUs. Section 4 discusses micro-benchmarks for measuring features of the memory hierarchy. We present experimental results in Section 5 to show the effectiveness of our approach. In Section 6, we discuss future work and our conclusions.

# 2 Overview of X-Ray

Figure 1 shows the general framework for X-Ray micro-benchmarks. A micro-benchmark relies on one or more timing measurements of (*nano-benchmarks*) performed by a high-resolution *Timer*. Nano-benchmarks are not written by hand, rather they consist of C code emitted by a *Code Generator*, based on a parameterized *Configuration* and values for its corresponding parameters. Configurations are discussed in detail in Section 2.4.

The choice of which nano-benchmarks to run is made dynamically, by an *Online Analysis Engine*, which can base its decision on *External Parameters*, which are results from other micro-benchmarks, and most importantly on the results of the previous nano-benchmarks it chose to run.

To make this clear, let us consider an example – measuring the initiation interval of double-precision floating-point addition. X-Ray has a general Initiation Interval micro-benchmark, which can be briefly described as follows.

- The "External Parameters" are $\langle \text{ADD}, \text{F64} \rangle$. That is, we want to measure addition (ADD), and that we want to consider the double-precision floating-point data type (F64).
- The only "Other Micro-benchmark Results" is the CPU Frequency ($F_{CPU}$).
- The Online Analysis Engine uses the existing "throughput" parameterized nano-benchmark configuration which measures the average time $t$ to perform a sequence of independent operations. Its parameters are the operation (ADD) the data type (F64) and the number of independent operations $n$. It executes this nano-benchmark for increasing successive values of $n$, introducing more and more instruction level parallelism in the sequence. It stops when the average time per operation $\frac{t}{n}$ does not decrease any more, i.e. the CPU has been saturated and no more parallelism can be exploited. At this point, it generates a result of $F_{CPU} \times \frac{t}{n}$, which is the rate at which the CPU can issue independent operations of this type (in cycles).

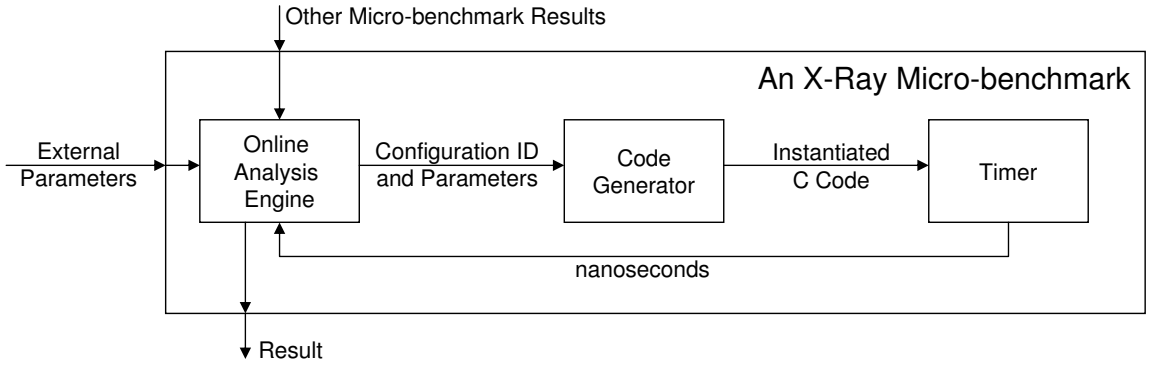


Figure 1: The X-Ray Framework for Micro-benchmarks

To implement a new micro-benchmark, one needs to first decide what measurements are needed and implement the corresponding nano-benchmarks, unless they are already developed. This is a matter of a few lines of code, which describe the parameterized configuration. After that, the script for the Online Analysis Engine needs to be implemented, which describes which nano-benchmarks to run, in what order, and how to produce a final result from the external parameters and the results of the nano- and other micro-benchmarks.

## 2.1 Operating system calls

The C language standard [4] does not specify all of the functionality that is required to implement our set of micro-benchmarks. Certain things, such as the API for high performance timing and thread manipulation, vary from operating system to operating system. Even targeting a standard OS API, such as POSIX, is not sufficient, since certain important elements are missing there also. Therefore, we decided to build a system that is architecture independent but not OS-independent. We feel that this is a reasonable tradeoff because using OS-specific APIs allows us to produce much better results, and because there are far fewer different operating systems to support than architectures.

Currently X-Ray uses OS-specific calls for the following.

- Initializing and reading high-resolution timers, discussed in Section 2.2
- Boosting the priority of the current thread, used for accurate timing if available
- Setting the affinity of a thread to a particular virtual processor, discussed in Section 3.6
- Obtaining the number of available virtual processors, discussed in Section 3.6
- Allocation and deallocation of super-pages, discussed in Section 4.5.2

## 2.2 Measuring performance accurately

Even with access to an OS-specific high-resolution timer, it is hard to accurately time operations that take only a few CPU cycles to execute.

Suppose we want to measure the time taken to execute a statement $S$. If this time is small compared to the granularity of the timer, we must measure the time required to execute this statement some number of times $R_S$ (dependent on $S$), and divide that time by $R_S$. If $R_S$ is too small, the time for execution cannot be measured accurately, whereas if $R_S$ is too big, the experiment will take longer than it needs to.

```
R = 1;
while(true)
{
   t = measure_S(R);
   if (t > t_min)
      return t / R;
   R *= 2
}
```

Figure 2: Pseudo-code for time measurement

```
   t_s = now();
   i = R;
loop:  S;
   if (--i)
      goto loop;
   t_e = now();
   return t_e − t_s;
```

(a)

```
   t_s = now();
   i = R / U;
loop:
   S;
   S;
   ...repeat U
times...
   S;
   if (--i)
      goto loop;
   t_e = now();
   return t_e − t_s;
```

(b)

```
initialize;
volatile int v = 0;
switch (v)
{
case 0:
   i = R/U;
   t_s = now();
loop:
case 1:  S;
case 2:  S;
...
case U:  S;
   if (--i)
      goto loop;
   t_e = now();
   if (!v)
      return t_e − t_s;
}
use;
```

(c)

```
initialize;
volatile int v = 0;
switch (v)
{
case 0:
   i = R/U;
   t_s = now();
loop:
case 1:  S_1;
case 2:  S_2;
...
case i:  S_i;
...
case n:  S_n;
case n+1:  S_1;
...
case W:  S_n;
   if (--i)
      goto loop;
   t_e = now();
   if (!v)
      return t_e − t_s;
}
use;
```

(d)

Figure 3: Approach for timing nano-benchmarks

To determine $R_S$, we start by setting $R_S$ to 1, and exponentially grow $R_S$ until the experiment runs for at least $t_{min} = 0.5s$ seconds. Pseudo-code is shown in Figure 2.

It is straight-forward to show that the total execution time for this code is bounded from above by $4 * t_{min}$. In this code, `measure`$_S$`(`$R_S$`)` is a procedure that executes $R_S$ repetitions of statement $S$, whose implementation we discuss next.

## 2.3 Preventing unwanted compiler optimizations

A simplistic implementation of `measure`$_S$ is shown in Figure 3(a). This code will have considerable loop overhead, which might be greater than the time spent in executing $S$. To address this problem, we can unroll the loop $U$ times to reduce the loop overhead as shown in Figure 3(b). Another problem is that optimizations by the C compiler may ruin the experiment. For example, consider the case when we want to measure the time for a single addition operation. In our framework, we would measure the time taken to execute the assignment statement $p_0$=$p_0$+$p_1$, where $p_0$ and $p_1$ are variables. Many C compilers will replace the $U$ instances of $p_0$=$p_0$+$p_1$ in the loop body to the single statement $p_0$=$p_0$+$U$*$p_1$, which prevents the time for executing statement $S$ from being measured! One option is to disable optimizations in the compiler, but this is likely to disable register allocation and instruction scheduling. In our example, $p_0$ and $p_1$ would be assigned to memory locations and the operation $p_0$=$p_0$+$p_1$ is likely to involve two loads and one store in addition to the addition operation.

What we need to do is to generate C programs which the compiler can optimize aggressively without disrupting

the sequence of operations whose execution time we want to measure. We solve this problem using two features of C, the `volatile` specifier and the `switch` construct. The semantics of C require that a `volatile` variable cannot be optimized by allocating it to a register, and that all reads and writes to it must go to memory. Therefore, we can rewrite our `measure`$_S$`(`$R$`)` code as shown in Figure 3(d). The roles of `initialize` and `use` are explained below. Because $v$ is a `volatile` variable, the compiler cannot assume anything about its value. Since every instance of $S$ has a `case` label, the compiler is unlikely to combine the instances in any way.

If `initialize` assigns the value of a `volatile` variable to each of the variables that appear in $S$, then the compiler will not be able to optimize the initialization away. Furthermore, it will not be able to assume anything about the initial values of these variables. Finally, there must be a "use" for each of the variables that appear in $S$ after the computation; otherwise the compiler might be able to delete the entire computation as dead code. This can be achieved by assigning the value of each variable to another `volatile` variable.

### 2.4 Configurations

In X-Ray, nano-benchmark code of the form shown in Figure 3(d) is generated on the fly by the Code Generator in Figure 1. Its input is a stylized specification of the statement that must be timed, which we call a *configuration*. For example, for determining the time to add two doubles, we use the configuration $\langle p_1 = p_1 + p_2, double, 2\rangle$. In this tuple, the first argument is the statement whose execution time is to be measured, the second argument is the type of the variables, and the third argument is the number of variables. Given this configuration, the code generator produces code of the form shown in Figure 3(d).

As we will see in Section 3, there are cases where we wish to measure the performance of a sequence of different statements $S_1, S_2, \ldots, S_n$. To prevent the compiler from optimizing this sequence, the code generator will give each $S_i$ a different case label, generating code of the form shown in Figure 3(d). In this figure, the number of case labels $W$ is the smallest multiple of $n$, not smaller than $U$. The configuration for this sequence we write as $\langle S_1|S_2|\ldots|S_n, T, r\rangle$.

## 3 CPU Micro-benchmarks

We now describe how X-Ray measures a number of key CPU parameters. We use the following standard terminology in this section.

### 3.1 CPU Frequency

CPU Frequency ($F_{CPU}$) is an important hardware parameter because other parameters are measured relative to it (i.e. in clock cycles). This micro-benchmark needs only the timing $t$ of the configuration $\langle p_1 = p_1 + p_2, \texttt{int}, 2\rangle$ and returns $\frac{1000}{t}$, as t is measured in nanoseconds, the result is measured in MHz. The implicit assumption is that dependent integer adds can be executed at the rate of one per CPU cycle.

## 3.2 Instruction Latency

Instruction Latency is the number of cycles after an instruction is issued, when its result becomes available to subsequent instructions. This micro-benchmark needs the timing $l$ of the configuration $\langle p_1 = \mathrm{op}(p_1, p_2), \mathrm{type}, 2\rangle$ and the already computed CPU frequency $F_{CPU}$. Then it produces the result $F_{CPU} \times l$. Here "op" is the C operation that naturally maps to the needed instruction and "type" is the type of the operands.

## 3.3 Instruction Initiation Interval

Instruction Initiation Interval is the rate in cycles at which the CPU can issue independent instructions of that type. This is a more formal description of the example we used in the introduction. Required are the timings $t_i$ of $\langle \{p_1 = \mathrm{op}(p_1, p_{i+1}); p_2 = \mathrm{op}(p_2, p_{i+1}); \ldots p_i = \mathrm{op}(p_i, p_{i+1}); \}, \mathrm{type}, i+1\rangle$ and $F_{CPU}$. The Online Analysis Engine generates successively $t_0, t_1, \ldots, t_i$, effectively introducing more ILP in the measured compound statement. When $\frac{t_i}{i}$ stops decreasing, it generates the result $F_{CPU} \times \frac{t_i}{i}$.

## 3.4 Instruction Existence

In certain cases it is not obvious how a certain C statement is translated. One very common operation for numerical applications is $p_1 = p_1 + p_2 * p_3$. On some platforms, this statement is compiled into a single fused multiply-add instruction (FMA), while on some it is compiled into a separate multiply and add instructions. If an FMA instruction does not exist, the compiler will need an extra register to store the intermediate value and schedule two instructions instead of one. This has an impact on how such sequences of statements need to be scheduled. For example, the ATLAS framework produces different code for the compiler depending upon the existence of an FMA instruction.

With an FMA instruction the CPU can execute an add instruction "for free" together with a multiply instruction. Therefore we determine the existence of FMA by comparing the initiation interval of a simple multiply and of a fused multiply-add.

On some machines, determining the existence of an FMA by measuring latency, as opposed to initiation interval, is not correct. For example on the SGI R12000 architecture, the latency of an FMA is 4 cycles, while the latencies of separate multiply and add are 2 cycles each. Nevertheless, using the FMA is beneficial because (i) it is fully pipelined and a new one can be issued each cycle and (ii) there is no need for a temporary register to hold the intermediate result.

## 3.5 Number of Registers

To determine the number of registers available to be allocated to variables of certain type, this micro-benchmarks needs the timings $t_i$ of $\langle \{p_1 = p_1 + p_i; p_2 = p_2 + p_1; \ldots p_i = p_i + p_{i-1}; \}, \mathrm{type}, i\rangle$. If all $p_k$ are allocated in registers, the time per operation $\frac{t_i}{i}$ is much smaller than when some are allocated in memory. The goal is to determine the maximum $i$, for which $\frac{t_i}{i}$ remains small. The Online Analysis Engine doubles $i$ until it observes the drop in performance for some $i = i_{max}$. After that it performs a binary search for $i$ in the interval $\left[\frac{i_{max}}{2}, i_{max}\right)$. It reports the value of $i$ as the number of registers allocatable to variables of this type.

### 3.6 SMP and SMT parallelism

To measure the number of processors in a SMP architecture, X-Ray once again uses the "throughput" nano-benchmark. Its configuration is instantiated with $\langle \text{ADD}, \text{I32}, n \rangle$, where $n$ is the value at which the CPU is saturated as measured by the Initiation Interval micro-benchmark.

The number $p$ of concurrent instances of this configuration, which exhibits no slowdown compared to running a single instance characterizes the number of CPUs in a SMP. Reading the number of CPUs with an OS call returns the number $v$ of virtual SMT processors. The SMT per CPU of the system is computed as $\frac{v}{p}$. To find which two virtual processors share the same physical processor, X-Ray executes instances of the aforementioned configuration concurrently on both. If there is no slowdown, the two virtual processors do not share a physical processor.

## 4 Memory Hierarchy Micro-benchmarks

Most computers have complex, multi-level memory hierarchies, which often include three levels of cache, main memory, and disk. Translation look-aside buffers (TLB's) are used to speed up the translation of virtual addresses to physical addresses. Optimizing the cache behavior of programs requires knowledge of each level's parameters.

In this section, we present algorithms for automatically measuring some memory hierarchy parameters that are important in practice. These algorithms are more powerful than existing ones in the literature. For example, our algorithm for measuring the capacity of a cache is the first one that is designed to accurately work for caches whose capacity is not a power of 2, and for caches that support exclusion, such as the caches on AMD machines. In addition, our benchmarks are not affected by hardware prefetching, unlike existing benchmarks.

### 4.1 Cache Parameters

We will focus on the *associativity*, *block size*, *capacity*, and *hit time* [6] of caches. Our experiments assume that the replacement policy is least-recently-used (LRU), since almost all caches implement variants of this policy, and our experiments show that even when they do not, the results are still accurate.

We will use Intel P6 (Pentium Pro/II/III) as our running example. On these machines, the L1 data cache has a capacity $C$ of 16KB, an associativity $A$ of 4, and a block size $B$ of 32 bytes. Therefore the cache contains $16384/32 = 512$ individual blocks, divided into $512/4 = 128$ sets of 4 blocks each. Figure 4 shows the structure of a memory address on this architecture. The highest 20 bits constitute the block *tag*, 7 bits are needed to *index* one of the 128 sets, and 5 bits are needed to store the *offset* of a particular byte within the 32-byte block. Through the rest of the paper, we will refer to the width of the fields by $t$, $i$ and $b$ respectively.

The following relation obviously holds: $C = A \times 2^{i+b}$. The associativity (and therefore the cache capacity) does not have to be a power of 2; for example, the L3 cache on the Itanium has an associativity of 24.

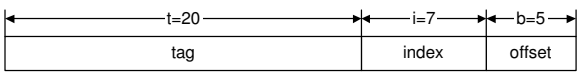


Figure 4: Memory Address decomposition on P6

## 4.2 Compact Sets

The routines described in this section determine cache parameters by measuring the amount of time it takes to repeatedly access certain sets of memory locations. If all these locations can reside in the cache simultaneously, the total access time is small compared to the time it takes to access them if they cannot all be in the cache at the same time.

**Definition 1.** *For a given cache, we say that a set of memory addresses $S$ is* compact *(with respect to some specific cache level, written* `compact`$(S)$*) if the data in these memory addresses can reside in the cache simultaneously. Otherwise, we say that that set of addresses is* non-compact.

For example, the set containing a singe memory address is always compact, whereas a set containing more than $C$ different addresses is not compact for a cache of capacity $C$. For any address $m$, the addresses in the half-open interval $[m, m + C)$ form a compact set. Compact sets have an obvious containment property: any subset of a compact set is itself compact. Equivalently, any superset of a non-compact set is non-compact.

The micro-benchmarks discussed in this section access sequences of $n$ memory locations using a fixed-stride $S = 2^\sigma$, shown pictorially in Figure 5. We will use $\langle S, n\rangle$ to refer to such a sequence. Theorem 1 describes necessary and sufficient conditions for the compactness of such a sequence for a given cache. Informally, this theorem says that as the stride gets larger (as $\sigma$ increases), the maximum length of a compact sequence with that stride gets smaller until it bottoms out at $A$. The proof of this theorem requires the following lemma.

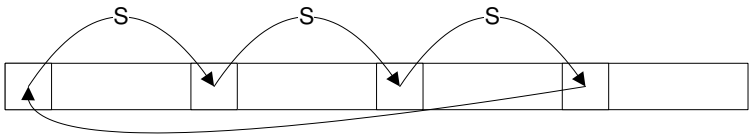


Figure 5: Fixed stride memory accesses ($n = 4$)

**Lemma 1.** *A sequence $R = \langle S(= 2^\sigma), n\rangle$ starting at address $m_0$ is non-compact iff $R$ contains a subsequence $Q = m_0, m_0 + k\frac{C}{A}, m_0 + 2k\frac{C}{A}, ..., m_0 + Ak\frac{C}{A}$ for some integer $k \geq 1$.*

*Proof.* ($\Leftarrow$): Since $\frac{C}{A} = 2^{i+b}$, the addresses in subsequence $Q$ differ only in the tag field, and not in the index or offset fields. Therefore, all these addresses map to the same cache set. Since there are $(A + 1)$ elements in the subsequence, this subsequence is non-compact, so $R$ is non-compact.

($\Rightarrow$): There are two cases.

- $S \geq \frac{C}{A}$: Since $S = 2^\sigma \geq \frac{C}{A} = 2^{i+b}$, $S$ is an integer multiple of $\frac{C}{A}$. Since $R$ is non-compact, it must have at least $(A + 1)$ elements. Therefore, the subsequence $Q$ can be chosen to be the sequence containing the first $A + 1$ elements of $R$, with $k = \frac{2^\sigma}{2^{i+b}}$.
- $S < \frac{C}{A}$: Since $S = 2^\sigma < \frac{C}{A} = 2^{i+b}$, $\frac{C}{A}$ is an integer multiple of $S$.

  Let $Q = m_0, m_0 + \frac{C}{A}, m_0 + 2\frac{C}{A}, ..., m_0 + C$. We have to show that $Q$ is a subsequence of $R$. This follows from the fact that (i) the step of $Q$ is an integer multiple of $S$, and (ii) the set of addresses in the interval $[m_0, m_0 + C)$ is compact, so the last element of the non-compact sequence $R$ must be greater than or equal to $m_0 + C$.

□

Theorem 1 follows immediately from Lemma 1.

**Theorem 1.** *Consider a cache $\langle A, B, C\rangle$. For a given stride $S = 2^\sigma$, let $n_c = \max(\frac{C}{S}, A)$. A sequence $R_c = \langle S, n_c\rangle$ is compact while a sequence $R_{nc} = \langle S, n_c + 1\rangle$ is not.*

## 4.3 Cache Parameter Measurement Framework

The usual approach to measuring cache parameters is to access elements of an array $A$ with different strides $S$ and then reason about the time per single access [6]. Pseudo-code for this approach is shown in Figure 6.

```
while (--R) // repeat R times
   for (int i = 0; i < N; i = i + 1)
      access(A[i * S]);
```

Figure 6: Naïve strided access

In practice, this approach is less than satisfactory for a number of reasons. The control overhead of the loop can introduce significant noise into timing measurements. Furthermore, the expression `A[i * S]` requires a relatively complex addressing mode containing scaling and displacement which might not be available on the testing platform in a single instruction. Finally, the different iterations of the `for` loop are independent so processors that exploit instruction level parallelism can execute some of them in parallel, complicating the measurement of serial execution time.

For these reasons, we use an approach similar to the one first introduced by the Calibrator project [7]. The idea is to declare that the array $A$ contains pointers (`void *`) as elements, and then initialize the array so that each element points to the address of the next element to be accessed. Figure 11 shows the pseudo-code. The variable $p$ is initialized to the element of the array that should be accessed first; the body of the loop traverses the chain of pointers starting at this point.

```
while (--R) // repeat R times
   p = *(void **)p;
```

Figure 7: Improved sequential stride access

Most CPUs have an indirect addressing mode, so the dereference can be translated into a single instruction. To improve timing accuracy, the loop can be unrolled an arbitrary number of times. In our implementation, we accomplish this by invoking the nano-benchmark code generator, discussed in Section 2, with the configuration $\langle p = $ `*(void **)`$p$, `void *`$, 1\rangle$ where $p$ is the only register needed of type "`void *`". The execution time of this configuration will depend on how $p$ and the corresponding array of pointers $A$ are initialized. We will always initialize $p$ to the first element of $A$. We discuss different ways of initializing $A$ in the following sections.

## 4.4 L1 Data Cache

Theorem 1 suggests a method for determining the capacity and associativity of the cache. First, we find $A$ by determining the asymptotic limit of the size of a compact set as the stride is increased. The smallest value of the

stride for which this limit is reached is $\frac{C}{A}$; once we know $A$, we can find $C$.

Our algorithm for measuring the capacity and associativity of the L1 data cache is presented in Figure 8. We use the test `compact`$(\langle S, n\rangle)$ which determines if the set $\langle S, n\rangle$ is compact by comparing its time per access with the time per access of a set like $\langle 1, 1\rangle$ that is known to be compact. These times are measured using the configuration discussed in Section 4.3.

```
S ← 1
n ← 1
while (compact(⟨S, n⟩))
    n ← n × 2
n_old ← 0
while (true)
    n ← smallest n' ≤ n such that ¬compact(⟨S, n'⟩)
    if (n = n_old)
        A = n − 1
        C = S/2 × A
        exit
    S ← S × 2
```

Figure 8: Measuring Capacity and Associativity of L1 Data Cache

The algorithm in Figure 8 can be described as follows. Start with stride $S = 1$ and sequence length $n = 1$. Exponentially grow the length of the sequence $n$ until the set is no longer compact. Then exponentially grow the stride $S$ and for each stride, find the length $n$ for which the sequence $\langle S, n\rangle$ is not compact. This $n$ can be found by using binary search in the interval $[1, n_{old}]$, where $n_{old}$ is the value computed for $n$ for the previous value of $S$. Stop when $n = n_{old}$. At this point, we can declare that the cache has associativity $A = n - 1$ and capacity $C = \frac{S}{2} \times A$.

### 4.4.1 Measuring Block Size

For given cache parameters $C$ and $A$, a sequence $\left\langle \frac{C}{A}, A\right\rangle$ is compact. Furthermore, the sequence $\left\langle \frac{C}{A}, 2 \times A\right\rangle$ is non-compact, as it has $2 \times A$ addresses that map to the same cache set, while only $A$ can be accommodated at any given time. Let us name these $2 \times A$ different addresses as $a_0, a_1, \ldots, a_{A-1}, a_A, \ldots, a_{2\times A}$. Now suppose we offset addresses in the second half of the sequence $(a_{A+1}, a_{A+2}, \ldots, a_{2\times A})$ by a constant $b$, to get the new sequence $a_0, a_1, \ldots, a_A, a_{A+1} + b, \ldots, a_{2\times A} + b$, which we denote by $\left\langle \frac{C}{A}, 2 \times A, b\right\rangle$ and pictorially present in Figure 9.

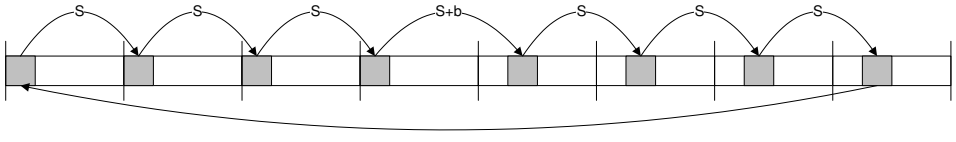


Figure 9: Modified address sequence for block size measurement

It is clear that when $b$ is smaller than the block size of the cache, all addresses in the new sequence will still map to the same cache set. On the other hand, if $b$ is equal to the block size of the cache, offsetting the second half of the sequence by $b$ will change the cache index of all these addresses and thus they will map to the next cache set, effectively making the whole sequence compact. Therefore we can use the simple code in Figure 10 to measure the

cache block size.

```
b ← 1
while ¬compact(⟨C/A, 2 × A, b⟩)
    b ← b × 2
return b
```

Figure 10: Algorithm for Measuring Block Size

### 4.4.2 Measuring Hit Latency

To measure the cache hit latency ($l_{hit}$), we only need to choose one compact set of addresses, like $\left\langle \frac{C}{A}, A \right\rangle$, or even $\langle 1, 1 \rangle$ and measure the time per access $t_{hit}$. We can use the CPU frequency $F_{CPU}$ to express this latency in cycles $l_{hit} = t_{hit} \times F_{CPU}$.

### 4.4.3 Correcting for hardware prefetching

Although the approach describes in this section works for most architectures, it does not work well on processors like the IBM Power 3 that have a hardware prefetch unit that can dynamically detect fixed-stride accesses to memory and prefetch the data. Our solution to this problem is to access the addresses in a set in pseudo-random order while ensuring that all addresses are indeed accessed. In this setting, the stride is not constant, so the hardware prefetcher will not detect any patterns to exploit.

Suppose the address sequence is $\{a_i\}_{i=0}^{n-1}$. One way to reorder this sequence is to choose a prime number $p > n$ and then, after visiting $m_i$, visiting element $a_{(i+p) \text{ modulo } n}$ instead of $a_{(i+1) \text{ modulo } n}$. As $p$ and $n$ are mutually prime, the recurrence $i = (i + p)$ modulo $n$ will generate all the integers between 0 and $n - 1$ before repeating itself. In X-Ray, this preprocessing is performed during array initialization so the code from Figure 8 can be used unchanged.

## 4.5 Lower Levels of the Memory Hierarchy

We will denote the cache at level $i$ as $\mathcal{C}_i$ and the parameters of that cache as $\mathcal{C}_i = \langle A_i, B_i, C_i \rangle$.

Measuring parameters of lower levels of the memory hierarchy is considerably more difficult than measuring the parameters of the L1 data cache. The algorithms described in Section 4.4 cannot be used directly for a variety of reasons.

1. In general, $\mathcal{C}_i$ is accessed only if $\mathcal{C}_{i-1}$ suffers a miss, and thus all sets of memory addresses that we want to test $\mathcal{C}_i$ against should be non-compact with respect to $\mathcal{C}_{i-1}$. This creates a problem when a lower cache level is less associative than a higher cache level (e.g. DEC Alpha 21264, IBM Power 3, etc.).
2. While $\mathcal{C}_1$ is typically virtually indexed, lower cache levels are always physically indexed on modern machines. This creates a problem because contiguous virtual addresses may not necessarily map to contiguous addresses in physical memory. Therefore, a sequence of fixed-stride accesses in the virtual address space may not necessarily map into such a sequence in physical memory.

We address these two issues next. Our algorithms assume that (i) $C_{i+1} \geq 2 \times C_i$ and (ii) $l_{i+1} \geq 2 \times l_i$, i.e. each cache level as at least twice bigger and at least twice slower as the cache directly above it. The size constraint is quite

natural and to the best of our knowledge is true for all machines in use today. The speed constraint is true most of the time, but there are specific isolated cases in which it is violated, e.g. the L1 cache on Pentium 4 with respect to floating-point numbers has a latency of 6 cycles, while the L2 cache has a latency of 7 cycles. In such rare cases, X-Ray would ignore the L1 cache and measure the parameters of the L2 cache. This is reasonable from a code optimization perspective because on such a machine, we would tile for the L2 cache, and not for the L1 cache.

### 4.5.1 Maintaining non-compactness with respect to $\mathcal{C}_{i-1}$

Intuitively, when measuring parameters for larger caches such as those at the lower levels of the memory hierarchy, the strides we use to generate address sequences will be bigger than those we use for higher cache levels. Because strides are always powers of 2, this means that all visited addresses will map to the same set in the higher levels. The problem is that we might not do enough accesses to exhaust the associativity of caches at the higher levels, and thus get an undesired hit at the higher levels. This is particularly true when the lower level cache is less associative than some higher level cache above it, which, as we pointed out, happens in practice.

First consider only two cache levels $\mathcal{C}_1$ and $\mathcal{C}_2$. Further let us suppose we already measured the $C_1$ and $A_1$ parameters of $\mathcal{C}_1$. When measuring the parameters of $\mathcal{C}_2$, instead of using $n$ monolithic strides of size $S$ (as shown in Figure 5), we can decompose each stride into $r+1$ smaller strides – $r$ of size $S_{old} = \frac{C_1}{A_1}$ and the last one of size $S - r \times \frac{C_1}{A_1}$, as shown in Figure 11.

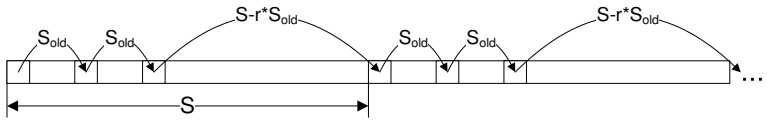


Figure 11: Stride decomposition for lower cache levels

By definition all $n \times r$ addresses we access this way will map to the same set in $\mathcal{C}_1$, and there will be $r$ groups of $n$ address, each of which maps into a different set in $\mathcal{C}_2$. Now for any given $n$ we can choose $r$ in such a way that $r \times n > A_1$ and thus guarantee non-compactness with respect to $\mathcal{C}_1$. In particular we can choose $r = \lceil \frac{A_1+1}{n} \rceil$.

Before we conclude this section, there are two things left to address. First, if we have $m > 2$ different cache levels and have measured the first $m-1$ of them, we need to make sure they all miss when measuring $\mathcal{C}_m$. We can achieve this using the above approach by treating the higher $m-1$ levels as a single cache with capacity $C_{\mathsf{up}} = C_{m-1}$ and associativity $A_{\mathsf{up}} = C_{\mathsf{up}} \div \min_{i=1}^{m-1} \left(\frac{C_i}{A_i}\right)$. The proof of this fact follows easily from Theorem 1 and is left to the reader.

Second, we have assumed thus far that the strides needed to detect lower levels are bigger than the strides to detect higher levels. This is not always true in practice, e.g. one model of AMD AhtlonMP has 64KB 2-way L1 cache (32KB strides needed) and 256KB 16-way L2 cache (16KB strides needed). Although we cannot use the stride decomposition technique discussed above for such a machine, there is no problem in cases like this. Suppose we use a stride $S < \frac{C_1}{A_1}$. Therefore $S = \frac{C_1}{A_1} \div 2^k$ for some $k > 0$, and thus addresses accessed $2^k$ strides apart map to the same L1 cache set. Furthermore we know that $C_2 \geq 2 \times C_1$, so with stride $S$ we need to access at least $\frac{2 \times C_1}{S}$ addresses. If we substitute $S$ from above, we get $\frac{2 \times C_1}{\frac{C_1}{A_1} \div 2^k} = 2^k \times 2A_1$. This means that $2A_1$ different addresses will be mapped to a single L1 cache set – an obvious cache miss.

### 4.5.2 Maintaining physical contiguity of memory

To apply these algorithms for cache parameter measurement to physically indexed caches, we need physically contiguous memory. There are two ways to acquire such memory:

1. request physically contiguous pages from the OS, or
2. 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. Another, somewhat smaller problem is that such memory regions typically consist of many pages and TLB misses might introduce 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 memory allocation and deallocation routines in the OS-specific part of X-Ray, 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 [10]. When such support becomes generally available, our OS-specific solution will not be required.

### 4.5.3 Summary

We can use the algorithms in Figures 8 and 10 on lower cache levels, provided that we do memory allocation as discussed in Section 4.5.2 and that we update the address sets $\langle S, n, S_{old}, r\rangle$ and $\langle S, n, b, S_{old}, r\rangle$ to include the new parameters $S_{old}$ and $r$, discussed in Section 4.5.1.

Finally, we were pleasantly surprised that our cache parameter measurement algorithm was able to handle cache exclusion, present in current AMD processors, although it was not designed with this in mind. We discuss this in more detail in Section 5.2.2.

## 4.6 Measuring TLB Parameters

The general structure of a virtual memory address is shown in Figure 12 (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 Translation Look-aside Buffer (TLB) is used to cache and reuse the results.

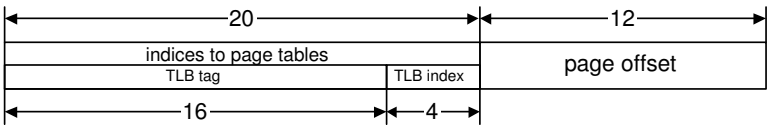


Figure 12: Memory Address decomposition on P6

A TLB has a certain number of *entries* $E$ each of which can cache the address translation for a single page. Although a TLB does not store the data itself, for our purposes it can be looked at as if it is a normal data cache

$\mathcal{C}_{TLB} = \langle A, B, C \rangle$, where $B$ is the page size and $C = E \times B$. Furthermore a TLB uses the upper portion of the virtual address the way a normal cache does (for encoding index and tag). All these observations imply that we can use our general cache parameter measurement algorithm to measure TLB parameters.

Some complications that arise when trying to do this are outlined below, together with solutions.

1. *Variable page size*: measuring parameters for caches with variable block size is not possible with our current algorithms. As we discuss in Section 4.5.2, modern operating systems use only a single page size transparently, and therefore there is no immediate threat for measurement failure. Furthermore, [10] suggests that whenever such transparent operating system support becomes available, TLB misses will be automatically minimized to 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 ensure that the TLB is accessed when memory references are issued by the processor. Fortunately, nowadays all L1 data caches are physically tagged; otherwise a complete cache flush is necessary on every context switch. This guarantees a TLB query with each memory access.

Finally, we need to make sure we do not introduce noise in our measurements from misses at various cache levels. To achieve this we make sure that all accesses are L1 data cache hits. In theory this restriction limits the TLB associativity we can measure, though this is not an issue with any of the architectures we have looked at. For the following discussion, we assume that we have already measured the relevant L1 data cache parameters $\langle A_1, B_1, C_1 \rangle$

We achieve compactness with respect to the L1 data cache by modifying our address sequences as follows. Suppose one of the measurement algorithms from Section 4.4 applied to the TLB requests a timing of a sequence $\langle S, n \rangle$.

- If $S \geq \frac{C_1}{A_1}$, slice the sequence of $n$ addresses in groups of $A_1$ addresses. Modify the stride of all accesses that cross group boundaries to $S + B_1$.
- If $S < \frac{C_1}{A_1}$, slice the sequence of $n$ addresses in groups of $\frac{C_1}{S}$ addresses. Modify the stride of all accesses that cross group boundaries to $S + B_1$.

It is easy to see that the resulting address sequences are compact with respect to the L1 data cache. With this modification, we can use the existing algorithms to measure TLB parameters.

## 4.7 Discussion

An important feature of our cache capacity measurement algorithm is that the number of addresses it accesses in the micro-benchmark is close in magnitude to the associativity of the cache. In contrast, the number of addresses accessed by algorithms based on the one discussed in Hennessy & Patterson [6], is close in magnitude to the capacity of the cache itself.

The approach taken by X-Ray is superior because with fewer addresses, non-compactness produces a very pronounced performance drop, which is much easier to detect automatically.

# 5 Experimental Results

In this section, we will present and discuss the results from running X-Ray on a number of different hardware architectures. In particular, we will show results for the following,

- Intel Itanium 2, 2-way SMP, 1.5GHz
- AMD Opteron 240, 2-way SMP, 1.4GHz
- Sun UltraSPARC IIIi, 2-way SMP, 1GHz
- Intel Pentium 4 Xeon, 2-way SMP, 2.2GHz
- AMD Athlon MP 2800+, 2-way SMP, 2.25GHz
- SGI R12000, 300MHz
- IBM Power 3, 8-way SMP, 375MHz

We also compare our results with those from three other similar platform-independent tools:

- **Calibrator v0.9e** [7] is a memory system benchmark aimed at measuring capacity, block size, and latency at each level of the memory hierarchy and the corresponding parameters for TLBs, such as number of entries, page size, and latency.
- **lmbench v3.0a3** [9, 11, 8] is a suite of benchmarks for measuring operating systems parameters such as thread-creation time and context-switch time. In version 3, the authors have included several hardware benchmarks for measuring CPU frequency, 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 parameters for TLBs.

We also looked at ATLAS v3.6.0 [12], which has a set of micro-benchmarks to aid its empirical search engine. These micro-benchmarks measure the latency of floating-point multiplication, the number of floating-point registers, the existence of a fused multiply-add instruction and the capacity of the L1 data cache. ATLAS needs only approximate values for these hardware parameters because it uses these values to bound the search space for optimization parameters, and not to estimate optimal values for these parameters. The micro-benchmarks in ATLAS are of limited precision, although they are perfectly adequate for what is needed of them in the context of the ATLAS system. Therefore, we decided not compare X-Ray directly to it.

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 are the best ones we obtained in several trial runs.

| Parameter | Tool | Itanium 2 | Opteron 240 | UltraSPARC IIIi | Pentium 4 | Athlon MP | R12000 | Power 3 |
|---|---|---|---|---|---|---|---|---|
| Frequency | Actual | 1500 | 1400 | 1000 | 2200 | 2250 | 300 | 375 |
| | X-Ray | 1488.16 | 1379.31 | 994.21 | 4324.09 | 2129.19 | 298.26 | 375.43 |
| (MHz) | lmbench | 1497 | 1392 | 1001 | 2164 | 2117 | 292 | 375 |
| FP Add | Actual | 4 | 4 | 4 | 5 | 4 | 2 | 4 |
| Latency | X-Ray | 3.98 | 3.96 | 4.04 | 10 | 4 | 2 | 4 |
| (cycles) | lmbench | 4.01 | 4.04 | 3.88 | 5 | 4 | 2.01 | 4.01 |
| FP Multiply | Actual | 4 | 4 | 4 | 7 | 4 | 2 | 4 |
| Latency | X-Ray | 3.98 | 4 | 4.11 | 14.09 | 4 | 2 | 4 |
| (cycles) | lmbench | 4.01 | 4.04 | 3.79 | 6.99 | 4 | 2.01 | 4.01 |
| FP Add | Actual | 0.5 | 1 | 1 | 1 | 1 | 1 | 0.5 |
| Initiation Interval | X-Ray | 0.51 | 0.99 | 1 | 2.03 | 1.01 | 1 | 0.5 |
| (cycles) | lmbench | 0.34 | 1.48 | 2.73 | 1.02 | 0.73 | 3.44 | 1.34 |
| FP Multiply | Actual | 0.5 | 1 | 1 | 2 | 1 | 1 | 0.5 |
| Initiation Interval | X-Ray | 0.51 | 1 | 0.99 | 4 | 1.01 | 1 | 0.5 |
| (cycles) | lmbench | 0.34 | 1.48 | 2.85 | 1.05 | 0.75 | 3.44 | 1.34 |
| FMA existence | Actual | yes | no | no | no | no | yes | yes |
| | X-Ray | yes | no | no | no | no | yes | yes |
| (yes/no) | lmbench | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| Native 'int' | Actual | 128 | 16 | 32 | 8 | 8 | 32 | 32 |
| Registers | X-Ray | 123 | 14 | 24 | 5 | 5 | 22 | 28 |
| (count) | lmbench | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| FP 'double' | Actual | 128 | 16 | 32 | 8 | 8 | 32 | 32 |
| Registers | X-Ray | 128 | 16 | 31 | 8 | 8 | 32 | 32 |
| (count) | lmbench | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| SMP | Actual | 2 | 2 | 2 | 2 | 2 | 1 | 8 |
| | X-Ray | 2 | 2 | 2 | 2 | 2 | 1 | 8 |
| (count) | lmbench | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| SMT per CPU | Actual | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| | X-Ray | 2 | 1 | 1 | 1 | 1 | 1 | 1 |
| (count) | lmbench | !supported | !supported | !supported | !supported | !supported | !supported | !supported |

Table 1: Summary of Experimental Results: CPU Features

## 5.1 CPU Parameters

Table 1 shows the CPU parameters measured by X-Ray and lmbench. Calibrator and MOB address only the memory hierarchy, so we do not discuss them here. The parameters which lmbench does not measure are marked "!supported". Now we discuss the individual parameters in greater detail.

### 5.1.1 CPU Frequency

Processor frequency is not a valuable parameter by itself; its only purpose is to express the latencies of other hardware parameters in clock cycles. Therefore it is only important that it be a consistent base for relative comparisons. In addition, the actual CPU frequency may differ from the advertised frequency by 5%.

On most of the architectures, the results of both X-Ray and lmbench (in Table 1) were close to the advertised actual frequencies. The only exception is the Pentium 4, where X-Ray measured double the actual value. The reason for this inconsistency is that the X-Ray methodology for computing frequency assumes that integer adds have an initiation interval of 1, but the Pentium 4 has a double-pumped integer ALU unit. In effect, the integer add instruction has an initiation interval of 0.500 on the P4. For relative comparisons of instruction latencies, this anomaly is not important.

### 5.1.2 Instruction Latency and Initiation Interval

X-Ray can measure the latency and initiation interval of any instruction; for lack of space, we show only the results for double-precision floating-point addition and multiplication.

One detail is that lmbench measures these values in different units. It measures instruction latency $l$ in nanoseconds and instruction parallelism $p$ as a factor of improvement over $l$. To produce directly comparable results, we converted

the latency to cycles, using the lmbench's measured processor frequency $F_{CPU}$: $l_{cycles} = l \times F_{CPU}$. Then we converted the parallelism to our notion of initiation interval using $II = \frac{l_{cycles}}{p}$.

Similarly, Calibrator does not implement CPU frequency measurement, but rather requires the user to input it as an external parameter. We used the advertised frequency of the processors to provide a value for this external parameter.

Both X-Ray and lmbench measure latencies very accurately, but lmbench does not measure some initiation intervals correctly. Two points deserve mention:

- X-Ray numbers on Pentium 4 are twice the actual values because they are relative to the integer ALU frequency, which, as mentioned in Section 5.1.1, is twice higher than the rest of the core.
- On Itanium 2 and Power 3, the initiation interval of both instructions is 0.5, which means that the CPU is dispatching 2 instruction per cycle. This is because these architectures have two FP ALUs.

  Paradoxically, X-Ray found that not every CPU with two fully pipelined ALUs has an initiation interval of 0.5. For example the Pentium III and Pentium 4 both have two integer ALUs, but their initiation intervals are 0.67. This anomaly arises from the fact that there are other bottlenecks in the CPU which prevent it from issuing the two integer instructions at a sustained rate. On the Pentium III the bottleneck is that there are only two register read ports, while on the Pentium 4 the bottleneck is the bandwidth between the instruction cache and the CPU core.

  This paradox highlights a recurring theme in our investigation. When there is a disagreement between X-Ray and the vendor's manuals, the values measured by X-Ray are more likely to be relevant to program optimization.

### 5.1.3 Number of Registers

In the suite of tools we looked at, X-Ray was the only tool that tried to measure the number of CPU registers. In some architectures, some registers are not available to a compiler for register allocation of program variables, because these registers are reserved for other uses such as to hold the value zero or to hold the stack pointer. It is important to note that X-Ray measures the number of registers that are available for register allocation of program variables by a compiler, not the number of hardware registers. This information is more useful for a self-tuning system since it controls program transformations such as register tiling. This is an instance in which X-Ray gives useful information that cannot be found in hardware manuals.

We present the measurements for native integer and double-precision floating-point registers in Table 1. The number of floating-point registers measured by X-Ray matches the actual number on all architectures except the UltraSPARC IIIi, which reserves a number of registers for holding the value zero, for implementing register windows, etc.[1]

We were also able to accurately determine the number of vector registers (MMX, SSE, SSE2) by supplying the appropriate ISA extension switch to GCC and using the available intrinsic functions. The only change needed was the register type in the testing configuration.

[1] The default compiler settings were to generate code for an older version of the ISA, so X-Ray measured only 15 floating-point registers. After supplying the appropriate switch for the IIIi model, X-Ray successfully found all registers.

### 5.1.4 Other CPU features

As shown in Table 1, X-Ray was able to determine the presence or absence of a fused multiply-add and the correct values for SMP and SMT on all target platforms.

## 5.2 Memory Hierarchy

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:

- **!supported** – the specific tool does not claim to be able to measure this hardware parameter;
- **skipped** – the benchmark completed but did not produce a value for this parameter;
- **aborted, segfault, bus error** – an abnormal termination of some kind occurred prevented the benchmark from completion;
- **os support** – OS-specific support is required for X-Ray to complete this measurement and we have not implemented such support yet.

### 5.2.1 L1 Data Cache

As Table 2 shows, X-Ray successfully found the correct values for all L1 cache parameters on all the architectures 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.

### 5.2.2 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 4.5.2. 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 architectures 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.

| Parameter | Tool | Itanium 2 | Opteron 240 | UltraSPARC IIIi | Pentium 4 | Athlon MP | R12000 | Power 3 |
|---|---|---|---|---|---|---|---|---|
| L1 Data Cache Capacity (KB) | Actual | 16 | 64 | 64 | 8 | 64 | 32 | 64 |
| | X-Ray | 16 | 64 | 64 | 8 | 64 | 32 | 64.5 |
| | Calibrator | 16 | 64 | 64 | 8 | 64 | 32 | 64 |
| | lmbench | segfault | bus error | 64 | 8 | skipped | 32 | 64 |
| | MOB | 4 | skipped | aborted | 8 | aborted | make failed | skipped |
| L1 Data Cache Block Size (bytes) | Actual | 64 | 64 | 32 | 64 | 64 | 16 | 128 |
| | X-Ray | 64 | 64 | 32 | 64 | 64 | 16 | 128 |
| | Calibrator | 64 | 32 | 32 | 32 | 64 | 64 | 128 |
| | lmbench | segfault | bus error | 32 | 64 | skipped | 32 | 128 |
| | MOB | 104 | skipped | aborted | aborted | aborted | make failed | skipped |
| L1 Data Cache Associativity (count) | Actual | 4 | 2 | 4 | 4 | 2 | 2 | 128 |
| | X-Ray | 4 | 2 | 4 | 4 | 2 | 2 | 129 |
| | Calibrator | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| | lmbench | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| | MOB | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| L1 Data Cache Hit Latency (cycles) | Actual | 2 | 3 | 2 | 2 | 3 | 2 | 2 |
| | X-Ray | 1.99 | 3 | 2 | 4.32 | 3.02 | 2.02 | 2.01 |
| | Calibrator | 2 | 3.02 | 1.99 | 2.02 | 3.17 | 2.07 | 2 |
| | lmbench | segfault | bus error | 2 | 2.06 | skipped | 2.01 | 2.01 |
| | MOB | 5.03 | skipped | aborted | aborted | aborted | make failed | skipped |
| L2 Cache Capacity (KB) | Actual | 256 | 1024 | 1024 | 512 | 512 | 512 | 512 |
| | X-Ray | 256 | 1088 | os support | 512 | 576 | os support | os support |
| | Calibrator | 256 | 768 | 1024 | 384 | 384 | 2048 | 6144 |
| | lmbench | segfault | bus error | 1024 | 512 | 512 | 2048 | 6144 |
| | MOB | 262144 | skipped | aborted | aborted | aborted | make failed | 0 |
| L2 Cache Block Size (bytes) | Actual | 128 | 64 | 64 | 128 | 64 | 128 | 128 |
| | X-Ray | 128 | 64 | os support | 128 | 64 | os support | os support |
| | Calibrator | 128 | 64 | 64 | 128 | 64 | 128 | 128 |
| | lmbench | 128 | 64 | 64 | 128 | 64 | 128 | 128 |
| | MOB | skipped | skipped | aborted | aborted | aborted | make failed | skipped |
| L2 Cache Associativity (count) | Actual | 8 | 16 | ? | 8 | 16 | ? | ? |
| | X-Ray | 8 | 17 | os support | 8 | 18 | os support | os support |
| | Calibrator | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| | lmbench | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| | MOB | !supported | !supported | !supported | !supported | !supported | !supported | !supported |
| L2 Cache Hit Latency (cycles) | Actual | ? | ? | ? | ? | ? | ? | ? |
| | X-Ray | 5.98 | 22.8 | 12.89 | 41.52 | 36 | 13.69 | 18.13 |
| | Calibrator | 4.16 | 13.13 | 12.41 | 17.75 | 18.25 | 11.94 | 8.52 |
| | lmbench | segfault | bus error | 15.15 | 20.36 | 2.69 | 13.86 | 17.19 |
| | MOB | 5.54 | skipped | aborted | aborted | aborted | make failed | 5.54 |
| L3 Cache Capacity (KB) | Actual | 6144 | | | | | | |
| | X-Ray | 6144 | | | | | | |
| | Calibrator | 6144 | | | | | | |
| | lmbench | segfault | | | | | | |
| | MOB | 4096 | | | | | | |
| L3 Cache Block Size (bytes) | Actual | 128 | | | | | | |
| | X-Ray | 128 | | | | | | |
| | Calibrator | 128 | | | | | | |
| | lmbench | segfault | | | | | | |
| | MOB | skipped | | | | | | |
| L3 Cache Associativity (count) | Actual | 24 | | | | | | |
| | X-Ray | 24 | | | | | | |
| | Calibrator | !supported | | | | | | |
| | lmbench | !supported | | | | | | |
| | MOB | !supported | | | | | | |
| L3 Cache Hit Latency (cycles) | Actual | ? | | | | | | |
| | X-Ray | 19.12 | | | | | | |
| | Calibrator | 14.31 | | | | | | |
| | lmbench | segfault | | | | | | |
| | MOB | 5.64 | | | | | | |
| Main Memory Latency (cycles) | Actual | ? | ? | ? | ? | ? | ? | ? |
| | X-Ray | 297.65 | 136.21 | os support | 761.92 | 471.35 | os support | os support |
| | Calibrator | 281.45 | 126.81 | 164 | 372.42 | 400.57 | 110.92 | 136.22 |
| | lmbench | segfault | bus error | 172.81 | 368.31 | 197.58 | 122.06 | 160.84 |
| | MOB | skipped | skipped | aborted | aborted | aborted | make failed | skipped |

Table 2: Summary of Experimental Results: Memory Hierarchy Features

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. 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.

The cache access latency figures for all tools 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.

### 5.2.3 TLB measurement

As discussed in Section 4.6, measuring TLB parameters accurately is very hard. As a proof of concept, we implemented a kernel module under Linux, which allows the allocation of physically contiguous memory for a sequence of contiguous virtual pages. Running the modified measurement algorithms discussed in Section 4.6, 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 (65 vs. actual 64). This problem may be similar to the one we discussed about the L1 data cache of Power 3 in Section 5.2.1.

The other tools produced some TLB results none of which were correct, so we do not show them in detail.

# 6 Conclusions and Future Work

This paper presents (i) an approach that an autonomic, self-tuning system can use to measure hardware parameters, and (ii) an implementation of this approach called X-Ray. It is important to note that X-Ray is designed with the assumption that it will be used by other systems rather than human beings. Such systems need to perform the measurements on their own without human intervention. Therefore, consulting the vendor's manual is not only impossible but may even be undesirable because the system is interested in the effective value of an hardware parameter rather than the actual value listed in the manual.

We are actively developing new micro-benchmarks inside the X-Ray framework. Our current focus, and ideas for future work include:

- measuring the parameters of instruction caches,
- measuring other parameters of the memory hierarchy, like bandwidth and parallelism at each level, cache replacement policy and write mode,
- studying the effects of victim caches on the presented algorithms and resolving any issues that may arise,
- designing a strategy for finding all sets of instructions that can be issued in a single CPU cycle at a sustained rate,
- implementing the OS-specific part of the framework for as many different OS as possible, and

- improving the approach of TLB parameter detection for legacy systems with small page sizes.

The implementation of the framework is freely available and will be disclosed in the final paper.

# References


[1] Autonomic Computing: creating self-managing computing systems. `http://www.research.ibm.com/autonomic/`.

[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] International Organization for Standardization. *ISO/IEC 9899-1999, Programming Languages: C*, 1999. Technical Committee: JTC 1/SC 22/WG 14.

[5] Matteo Frigo and Steven G. Johnson. FFTW: An adaptive software architecture for the FFT. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, volume 3, pages 1381–1384, Seattle, WA, May 1998.

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

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

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

[9] 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.

[10] 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.

[11] 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.

[12] 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`).

[13] Kamen Yotov, Xiaoming Li, Gang Ren, Michael Cibulskis, Gerald DeJong, Maria Garzaran, David Padua, Keshav Pingali, Paul Stodghill, and Peng Wu. A comparison of empirical and model-driven optimization. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 63–76. ACM Press, 2003.