X-Ray: A Tool for Automatic Measurement of Hardware Parameters

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

Abstract

There is growing interest in self-optimizing computing systems that can optimize their own behavior on different platforms without manual intervention. Examples of successful self-optimizing systems are ATLAS, which generates Basic Linear Algebra Subroutine (BLAS) Libraries, and FFTW, which generates FFT libraries.

Self-optimizing systems need values for hardware parameters such as the number of registers of various types and the capacities of caches at various levels. For example, ATLAS uses the capacity of the L1 cache and the number of registers in determining the size of cache tiles and register tiles.

In this paper, we describe X-Ray[1]*, a system for implementing micro-benchmarks to measure such hardware parameters. We also present novel algorithms for measuring some of these parameters. Experimental evaluations of X-Ray on traditional workstations, servers and embedded systems show that X-Ray produces more accurate and complete results than existing tools.*

1. Introduction

There is growing interest in self-optimizing systems that can optimize their own behavior on different platforms without manual intervention [9, 3, 6]. These systems are based on the generate-and-test paradigm: instead of writing a program, one implements a program generator that produces a large number of program variants, and determines empirically which variant performs best. To prevent a combinatorial explosion in the number of program variants that have to be considered, self-optimizing systems bound the search space by using hardware parameters values such as the number of registers and the capacity of the L1 cache [9, 10].

For software to be truly self-optimizing, the values of hardware parameters relevant for software optimization must be determined automatically. It is important to note that these values are not necessarily the same as the values one might find in a hardware manual. For example, loop unrolling must take into account the number of registers available to hold values computed in the loop body. However, most compilers set aside certain registers for holding special values such as the stack or frame pointer, so the number of registers available to the register allocator is usually less than the total number of architected registers. In practice, it is hard to find documentation even for hardware parameter values, let alone for values relevant to software optimization.

In this paper, we describe X-Ray, a framework for implementing micro-benchmarks to measure relevant values of hardware parameters automatically. For portability, X-Ray is entirely implemented in ANSI C'89. One of the interesting challenges of this approach is to ensure that the C compiler does not perform any high-level restructuring optimizations on our benchmarks that might pollute the timing results, while performance critical optimizations, such as register allocation, are still enabled.

2. The X-Ray Framework

Hardware parameters are measured by X-Ray *micro-benchmarks*. Figure 1 presents the general structure of a micro-benchmark in the X-Ray framework.

As an example, consider the measurement of the number of available registers of a particular data type T. One way to determine this value is to perform a number of experiments, all of which perform the same computa-

[1] 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.

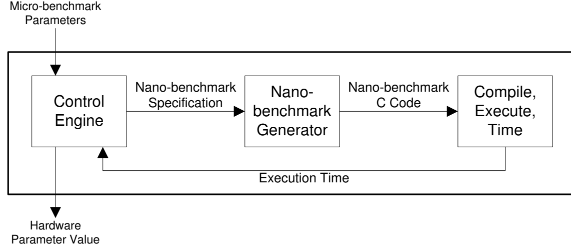

Figure 1. A micro-benchmark in X-Ray

tions but on a different number of variables (N) of type T. When N exceeds the number of available registers for type T, not all variables can be register allocated, and execution time should increase substantially. The number of available registers can be inferred from this cross-over point.

Some general conclusions can be drawn from this example. A micro-benchmark to determine the value of some parameter may need to time a number of different but related programs that we call *nano-benchmarks*. Since there may be no *a priori* bound on the number of required nano-benchmarks, we need a *Nano-benchmark Generator*, which can produce *Nano-benchmark C Code* from a high-level *Nano-benchmark Specification*. Finally, generation should happen on-the-fly since the results of one nano-benchmark may determine the nano-benchmark to be executed next.

In X-Ray, the execution of a micro-benchmark is orchestrated by its *Control Engine*, which chooses the nano-benchmarks to execute, the order in which they should be executed, and the appropriate parameters for each one. The Control Engine determines the value of the hardware parameter based on these timing results.

Some micro-benchmarks may also need the results obtained from running other micro-benchmarks. For example, to determine the latency of an instruction in cycles rather than in nanoseconds, the control engine needs to know the cycle time of the processor. This can be specified by the user or it can be measured by another micro-benchmark, as discussed in Section 3.

2.1. Nano-benchmarks

Even with access to a 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 required to execute a C 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_S ← 1;
while (measure_S(R_S) < t_min)
    R_S ← R_S × 2;
return (measure_S(R_S) ÷ R_S);
```

Figure 2. Nano-benchmark timing

Figure 2 shows the timing strategy used in X-Ray nano-benchmarks. In this code, measure$_S$(R_S) measures the time required to execute R_S repetitions of statement S. To determine a reasonable value for R_S, the code in Figure 2 starts by setting R_S to 1, and then doubles it until the experiment runs for at least t_{min} seconds. The value of t_{min} can be specified by the user and defaults to 0.25 seconds in the current implementation.

A simplistic implementation of measure$_S$ is shown in Figure 3(a). This code incurs considerable loop overhead, so we unroll the loop U times (Figure 3(b)).

Another problem is that restructuring compiler optimizations may corrupt the experiment. For example, consider the case when we want to measure the latency of a single addition. In our framework, we would measure the time taken to execute the C statement $p_0 = p_0 + p_1$. It is important to allocate p_0 and p_1 in registers, but it is crucial that the compiler not replace the U statements in the loop body by the statement $p_0 = p_0 + U \times p_1$, since this would prevent the code from timing the original statement correctly.

To solve such problems, we need to generate programs which the compiler can aggressively optimize without disrupting the sequence of operations whose execution time we want to measure. We solve this problem using a `switch` statement on a `volatile` variable v as shown in Figure 3(c). The semantics of C require that v be read from memory; therefore the compiler cannot assume anything about which `case` of the `switch` is selected. Because there is potential control flow to each of the `case` blocks, it is impossible for the compiler to combine or reorder them in any way.

The final problem is that if the compiler is able to deduce that the result of the computations performed in S is not used in the rest of the code, it might perform dead-code elimination and remove all instances of S altogether. To prevent this unwanted optimization, all variables that appear in S are assigned to values read from appropriately typed `volatile` variables in the `initialize` statement; similarly, their final values are copied back to the same `volatile` variables in the `use` statement.

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

```
measure_S(R) {
    t_s = now();
    i = R;
loop:  S;
    if (--i)
        goto loop;
    t_e = now();
    return t_e - t_s;
}
```
(a)

```
measure_S(R) {
    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)

```
measure_S(R) {
    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)

```
measure_S(R) {
    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. Implementation of $\texttt{measure}_S$

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 greater than or equal to U.

2.2. Nano-benchmark Generator

The X-Ray nano-benchmark generator accepts as an input a nano-benchmark specification and produces nano-benchmark C code structured as shown in Figures 3(c),3(d).

The nano-benchmark specification is a tuple which contains a statement S to be timed and type information for all variables in S. For example, to measure the latency of double-precision floating point ADD operation, we use the nano-benchmark specification $\langle p_1 = p_1 + p_2, \langle p_1, p_2 : \texttt{F64}\rangle\rangle$, which means that we time the statement $p_1 = p_1 + p_2$, where p_1 and p_2 are variables of type double (defined as `F64` in X-Ray). Given this specification, the nano-benchmark generator can produce code as shown in Figure 3(c). Generating code of the form shown in Figure 3(d) is more complex and requires the first element of the tuple to be a function $f : \mathsf{integer} \rightarrow \mathsf{string}$, which computes the code for statement S_i from the case label i.

2.3. Implementing a new micro-benchmark

As we will see in Section 3, implementing a new micro-benchmark in X-Ray requires:

1. Implementing the nano-benchmarks for all timing experiments. If their code fits the template in Figure 3(d), nano-benchmark specifications are enough;

2. Implementing the micro-benchmark control engine to describe which nano-benchmarks to run, with what parameters, in what order, and how to produce a final result from the external parameters and the timings.

3. CPU Micro-benchmarks

3.1. CPU Frequency

CPU frequency ($\mathsf{F}_{\mathsf{CPU}}$) is an important hardware parameter because other parameters are measured relative to it (in clock cycles). X-Ray assumes that dependent integer additions can be executed at the rate of one per cycle, which is valid for most current processors. The assumption of dependence is important because modern architectures can often issue two or more independent integer addition operations in one cycle, so timing independent addition operations would be misleading.

For this micro-benchmark we use a nano-benchmark with specification $S = \langle p_0 = p_0 + p_1, \langle p_0, p_1 : \texttt{int}\rangle\rangle$. Given the time $\texttt{time}(S)$ in nanseconds required to execute the statement S, we compute the CPU frequency in MHz as $\mathsf{F}_{\mathsf{CPU}} \leftarrow 1000 \div \texttt{time}(S)$.

As we will see in Section 5, the assumption that dependent integer additions are executed at the rate of one

per cycle may not be correct for some processors. In that case, all timing measurements reported by X-Ray must be scaled by an appropriate constant to obtain the actual values. This is not a serious problem since self-optimizing software uses timing measurements mostly to choose between different code sequences, so relative rather than absolute times are needed.

3.2. Instruction Latency

The latency $\mathsf{L}_{\mathsf{O,T}}$ of an operation (instruction) O, with operands of type T, is the number of cycles after one such instruction is dispatched until its result becomes available to subsequent dependent instructions.

We use a nano-benchmark with specification $S_{O,T} = \langle p_0 = O(p_0, p_1), \langle p_0, p_1 : T\rangle\rangle$. We then compute the instruction latency in clock cycles as $\mathsf{L}_{\mathsf{O,T}} \leftarrow \mathtt{time}\,(S_{O,T}) \div (1000 \div \mathsf{F}_{\mathsf{CPU}})$.

3.3. Instruction Throughput

The throughput $\mathsf{TP}_{\mathsf{O,T}}$ of an operation (instruction) O, with operands of types T, is the rate in cycles at which the CPU can issue independent instructions of that type. On modern processors the throughput of an instruction is usually much smaller than its latency, because of pipelining and super-scalar execution.

To measure $\mathsf{TP}_{\mathsf{O,T}}$, we could use a nano-benchmark specification as follows.

$$
\begin{array}{l}
S_{O,T,N,B} = \\
< \\
\quad \{ \\
\quad\quad p_{(i\times B+0)\%N} = O(p_{(i\times B+0)\%N}, p_N); \\
\quad\quad p_{(i\times B+1)\%N} = O(p_{(i\times B+1)\%N}, p_N); \\
\quad\quad \ldots \\
\quad\quad p_{(i\times B+B-1)\%N} = O(p_{(i\times B+B-1)\%N}, p_N); \\
\quad \}, \\
\quad \langle p_0, p_1, \ldots, p_N : T\rangle \\
>
\end{array}
$$

Note that this specification generates code of the form shown in Figure 3(d). It is further parameterized by N and B, which control the number of independent instructions to generate. For example, the sequence of statements generated for $N = 3$ and $B = 1$ is the following.

$$
\begin{array}{l}
\mathsf{case}\ 0 : \{p_0 = O(p_0, p_3);\} \\
\mathsf{case}\ 1 : \{p_1 = O(p_1, p_3);\} \\
\mathsf{case}\ 2 : \{p_2 = O(p_2, p_3);\} \\
\mathsf{case}\ 3 : \{p_0 = O(p_0, p_3);\} \\
\ldots \\
\mathsf{case}\ \mathtt{W} : \{p_2 = O(p_0, p_3);\}
\end{array}
$$

In general, we generate B simple statement per case label because otherwise we cannot measure ILP for statically scheduled VLIW cores. We then measure the instruction throughput in clock cycles as follows.

$$
\begin{array}{l}
N \leftarrow 2; \\
\mathtt{while}\left(\dfrac{\mathtt{time}\left(S_{O,T,N,1}\right)}{\mathtt{time}\left(S_{O,T,N-1,1}\right)} > 1-\epsilon\right) \\
\quad N \leftarrow N+1; \\
B \leftarrow 2; \\
N \leftarrow N-1; \\
\mathtt{while}\left(\dfrac{\mathtt{time}\left(S_{O,T,N\times B,B}\right)\div B}{\mathtt{time}\left(S_{O,T,N\times(B-1),B-1}\right)\div(B-1)} > 1-\epsilon\right) \\
\quad B \leftarrow B+1; \\
\mathsf{TP}_{\mathsf{O,T}} \leftarrow \dfrac{\mathtt{time}\left(S_{O,T,N\times(B-1),B-1}\right)\div(B-1)}{1000\div\mathsf{F}_{\mathsf{CPU}}};
\end{array}
$$

The nano-benchmark code for $S_{O,T,N,B}$ exhibits instruction- level parallelism (ILP) on the order of $N \times B$. The control engine times the nano-benchmark for $B = 1$ and successively growing values of N while performance continues to increase due to the additional ILP. When the performance levels off for some N, the control engine starts growing B to check if increasing ILP between case labels improves performance.

3.4. Instruction Existence

The existence of certain instructions can influence the code produced by some self-optimizing systems; for example, ATLAS exploits the existence of fused multiply-add (FMA). We determine whether a fused multiply-add instruction exists by comparing the throughput of a simple multiply with that of a fused multiply-add. Similarly, many embedded processors do not have dedicated floating-point hardware, but use an emulation library instead. In X-Ray, we measure the latency of a floating-point ADD, and assert that a hardware floating-point unit exists if the latency is less than 10 cycles.

3.5. Number of Registers

To measure the number of registers NR_{T} of type T, we use a nano-benchmark with specification $S_{T,N} = \langle p_{i\%N} = p_{i\%N} + p_{(i+N-1)\%N}, \langle p_0, p_1, \ldots, p_N : T\rangle\rangle$. For example, the sequence of statements generated for $N = 4$ is as follows.

$$
\begin{array}{l}
\mathsf{case}\ 0 : p_0 = p_0 + p_3; \\
\mathsf{case}\ 1 : p_1 = p_1 + p_0; \\
\mathsf{case}\ 2 : p_2 = p_2 + p_1; \\
\mathsf{case}\ 3 : p_3 = p_3 + p_0; \\
\mathsf{case}\ 4 : p_0 = p_0 + p_3; \\
\ldots \\
\mathsf{case}\ \mathtt{W} : p_3 = p_3 + p_0;
\end{array}
$$

If all of p_i are allocated in registers, the time per operation is much smaller than when some are allocated in memory. The goal is to determine the maximum N, for which no variables are allocated to memory. The control engine doubles N until it observes a drop in performance. After that it performs a binary search in the interval $[N \div 2, N)$. The actual control engine algorithm is as follows.

```
N ← 4;
while ( time(S_{T,N}) / time(S_{T,2}) < 1 + ε )
    N ← N × 2;
R ← N;
L ← N/2
while (R − L > 1)
    P ← (R+L)/2;
    if ( time(S_{T,P}) / time(S_{T,2}) < 1 + ε )
        R ← P;
    else
        L ← P;
NR_T ← L;
```

3.6. SMP and SMT

To measure the number of processors in a SMP architecture, X-Ray uses the throughput nano-benchmark of Section 3.3 with specification $S_{ADD,I32,N,B}$, where N and B are the values for which maximum throughput is achieved. The number p of threads running concurrent instances of this configuration that exhibit no slowdown compared to running a single thread characterizes the number of physical processors 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 configuration concurrently on both. If there is no slowdown, the two virtual processors do not share a physical processor.

4. Cache Micro-benchmarks

In this section we summarize our approach for measuring memory hierarchy parameters. A full description along with detailed proofs is given in [11]. The most well-known benchmark for measuring memory hierarchy parameters is the Saavedra benchmark [7], but the timing results are usually inspected manually to determine memory hierarchy parameters. Other approaches use hardware counters [1, 2], but these are not very portable. Our approach produces the hardware parameter values directly; moreover, our results are accurate even for arbitrary cache associativity, cache exclusion, and hardware stride prefetching.

We focus on measuring *associativity*, *block size*, *capacity*, and *hit latency* [4] of caches. The first three parameters are sometimes referred to as the $\langle A, B, C\rangle$ of caches, while the last parameter will be referred to as l_{hit}. The description of the algorithms given below makes use of a parameter $T = \frac{C}{A}$ that we call the *stride* of the cache.

4.1. Sequences and compact sequences

X-Ray determines memory hierarchy parameters by measuring the average time l to repeatedly access the elements of different address sequences. When each access is a cache hit, $l = l_{hit}$ is relatively small, and we say that the sequence is *compact*. When each access is a cache miss, $l = l_{miss}$ is relatively large, and we say that the sequence is *non-compact*. Sequences which are neither compact nor non-compact we call *semi-compact*.

To measure the capacity and the associativity of the L1 data cache, X-Ray uses sequences of N addresses, where successive addresses are separated by a stride $S = 2^{\sigma}$. Such sequences are completely characterized by their starting address m_0, stride S and number of elements N. We use the notation $\langle m_0, S, N\rangle$ to represent them.

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

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

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

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

Proof. Omitted. □

4.2. Measuring L1 Cache Parameters

Cache Latency

We determine the cache hit latency l_{hit} by measuring the average time to repeatedly access the elements of $\langle m_0, 1, 1\rangle$, which is obviously compact.

Capacity and Associativity

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

Pseudo-code for measuring C and A of the L1 data cache is shown below. We use is_compact (W) to empirically determine if W is compact by comparing the average time to repeatedly access the elements of W with the cache hit latency l_{hit}.

```
S ← 1;
N ← 1;
while (is_compact (⟨m_0, S, N⟩))
    N ← 2 × N;
repeat
    S ← 2 × S;
    N_old ← N;
    N ← min N' ∈ [1, N_old] : ¬is_compact (⟨m_0, S, N'⟩);
until (N = N_old);
A ← N − 1;
C ← S/2 × A;
```

The algorithm can be described as follows. Start with the sequence $\langle m_0, S, N\rangle = \langle m_0, 1, 1\rangle$, which is compact, and keep doubling N until the sequence is not compact. Let N_{old} 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, declare $A = N - 1$ and $C = \frac{S}{2} \times A$.

Block Size

For a cache with stride T and associativity A, the sequence $\langle m_0, T, 2A\rangle$ is non-compact since all $2A$ addresses map to the same cache set. This sequence can also be expressed as $\langle m_0, T, A\rangle \cup \langle m_0 + C, T, A\rangle$. If we offset the second half of the sequence by a constant δ as shown in Figure 4, we get a set of addresses $D = \langle m_0, T, A\rangle \cup \langle m_0 + C + \delta, T, A\rangle$.

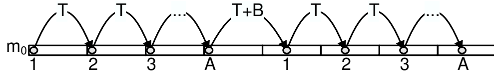

Figure 4. Sequence for measuring B

The addresses in each of the two subsequences map to a single cache set. When $0 \leq \delta < B$ this is the same cache set and D is non-compact, and when $\delta \geq B$ the cache sets are different and D is compact. Pseudo-code for the algorithm is shown below.

```
δ ← 1
while (¬is_compact (⟨m_0, T, A⟩ ∪ ⟨m_0 + C + δ, T, A⟩))
    δ ← 2 × δ;
return δ;
```

4.3. Measuring Parameters for Lower Levels

We can use the algorithms in Section 4.2 to measure parameters of a lower level cache l, provided we ensure that the memory accesses miss in all higher level caches $i < l$. We accomplish this by using sequences of sequences. For lack of space, we omit the details and refer the interested reader to a companion paper [11].

4.4. Implementation of is_compact

We represent our sequences of memory addresses with arrays of pointers (`void *`) instead of arrays of integers (`int`) as in the Saavedra benchmark. We initialize the array 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.

For a correct implementation it is important to repeatedly access all elements of the sequence, but the order in which we access them is irrelevant. To prevent hardware constant stride prefetchers, from interfering with our timings, we initialize the array elements by chaining the pointers so that we visit the elements in a pseudo-random order.

We perform the timings using a nano-benchmark with specification $\langle p = *(\texttt{void}**)p, \langle p : \texttt{void}*\rangle\rangle$. The fact that we can use the same nano-benchmark generator for measuring both CPU parameter values and cache parameter values demonstrates the flexibility of the X-Ray architecture.

5. Experimental Results

In this section, we present experimental results obtained by using X-Ray to measure the hardware parameters of a number of desktop and embedded platforms. Embedded processors are particularly challenging because there are many variations even within a single processor family (in fact, some companies like Tensilica make customizable embedded processors).

We compare our results to the actual values of the hardware parameters, as well as to the values obtained by lmbench v3.0-a4 [5]. We were unable to build or run lmbench on some of the architectures. Tables 1 and 2 show a summary of the experimental results for CPU features and the memory hierarchy respectively. In these tables we use the following special keywords:

- **!exist** – a micro-benchmark for measuring this hardware parameter does not exist in lmbench;
- **!os** – 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. Support for super-pages is very OS-specific, so we targeted Linux as a proof of concept. We are currently working on the implementation for Solaris, IRIX and AIX. Similarly lmbench relies on various OS features, which were not available on some of the platforms.

Feature	Tool	UltraSPARC IIIi	R12000	Power 3	Pentium 4	Itanium 2	Athlon MP	Opteron 240	ARM SA1110	xScale PXA250	Xtensa LX	R4400
Frequency (MHz)	Actual	1,000.000	300.000	375.000	3,060.000	1,500.000	2,250.000	1,400.000	200.000	400.000	350.000	150.000
	X-Ray	994.206	299.928	375.434	6,097.130	1,488.344	2,129.190	1,394.471	203.586	393.489	343.225	145.889
	lmbench	1,001.001	300.003	374.953	3,049.710	1,497.903	2,117.317	1,381.025	202.312	!os	!os	152.001
Latency ADD I32 (cycles)	Actual	1.000	1.000	1.000	0.500	1.000	1.000	1.000	1.000	1.000	1.000	1.000
	X-Ray	1.000	1.000	1.000	1.001	1.000	1.000	1.000	1.001	1.006	1.000	1.000
	lmbench	0.991	1.102	1.009	0.488	1.004	1.005	0.994	0.997	!os	!os	1.075
Latency MULTIPLY I32 (cycles)	Actual	?	6.000	3.000	14-18.000	4.000	4.000	3.000	3.000	2.000	2.000	12.000
	X-Ray	6.000	5.989	3.000	29.987	3.985	4.000	3.013	3.095	1.977	1.977	14.873
	lmbench	27.868	6.057	3.007	13.907	5.168	4.003	2.983	3.704	!os	!os	15.726
Throughput ADD I32 (cycles)	Actual	0.500	0.500	0.500	0.250	0.167	0.333	0.333	1.000	1.000	1.000	1.000
	X-Ray	0.509	0.503	0.497	0.679	0.169	0.386	0.345	1.001	1.003	0.996	1.000
	lmbench	0.522	0.509	0.490	0.375	0.469	0.540	0.360	0.899	!os	!os	0.814
Throughput MULTIPLY I32 (cycles)	Actual	?	6.000	3.000	5.000	0.500	2.000	1.000	2.000	1.000	1.000	12.000
	X-Ray	4.963	5.989	2.972	9.337	0.506	2.016	1.024	1.986	0.998	0.996	14.870
	lmbench	27.868	5.938	3.007	3.512	0.485	2.012	1.022	3.562	!os	!os	13.441
NR I32 (count)	Actual	32	32	32	8	128	8	16	16	16	16	32
	X-Ray	22	22	28	5	123	5	14	12	13	11	17
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist
FMA I32 (boolean)	Actual	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE	FALSE	FALSE
	X-Ray	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	FALSE	TRUE
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist
FPU F32 (boolean)	Actual	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
	X-Ray	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE	TRUE
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist
Latency ADD F32 (cycles)	Actual	4.000	2.000	4.000	5.000	4.000	4.000	4.000	n/a	n/a	4.000	4.000
	X-Ray	3.963	2.000	4.000	10.013	3.984	4.000	4.052	n/a	n/a	3.927	3.935
	lmbench	3.814	2.007	4.012	4.849	4.284	4.003	3.964	n/a	n/a	!os	4.179
Latency MULTIPLY F32 (cycles)	Actual	4.000	2.000	4.000	7.000	4.000	4.000	4.000	n/a	n/a	4.000	7.000
	X-Ray	4.037	2.000	5.034	14.079	3.984	4.000	4.052	n/a	n/a	3.927	6.870
	lmbench	3.804	2.007	4.014	6.892	4.014	4.003	3.964	n/a	n/a	!os	7.323
Throughput ADD F32 (cycles)	Actual	1.000	1.000	0.500	1.000	0.500	1.000	1.000	n/a	n/a	1.000	3.000
	X-Ray	1.000	1.000	0.497	2.030	0.506	1.000	1.013	n/a	n/a	0.996	2.913
	lmbench	2.667	1.004	0.501	2.108	0.549	1.522	1.231	n/a	n/a	!os	2.943
Throughput MULTIPLY F32 (cycles)	Actual	1.000	1.000	0.500	2.000	0.500	1.000	1.000	n/a	n/a	1.000	3.000
	X-Ray	1.009	1.000	0.500	4.004	0.506	1.000	1.013	n/a	n/a	0.996	2.957
	lmbench	2.503	1.004	0.501	2.209	0.515	1.522	1.611	n/a	n/a	!os	3.130
NR F32 (count)	Actual	32	32	32	8	128	8	16	n/a	n/a	16	32
	X-Ray	32	32	32	8	128	8	16	n/a	n/a	17	24
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	n/a	n/a	!exist	!exist
FMA F32 (boolean)	Actual	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	n/a	n/a	TRUE	FALSE
	X-Ray	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	n/a	n/a	TRUE	FALSE
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	n/a	n/a	!exist	!exist
FPU F64 (boolean)	Actual	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE
	X-Ray	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE	FALSE	TRUE
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist
Latency ADD F64 (cycles)	Actual	4	2	4	5	4	4	4	n/a	n/a	n/a	4
	X-Ray	4	2	4	10.005	3.984	4	4.052	n/a	n/a	n/a	3.935
	lmbench	3.824	2.007	4.012	4.819	4.014	4.003	3.964	n/a	n/a	n/a	4.2
Latency MULTIPLY F64 (cycles)	Actual	4	2	4	7	4	4	4	n/a	n/a	n/a	8
	X-Ray	4	2	4	14.023	3.986	4	4.052	n/a	n/a	n/a	7.826
	lmbench	3.804	2.007	4.012	6.892	4.014	4.003	3.964	n/a	n/a	n/a	8.397
Throughput ADD F64 (cycles)	Actual	1	1	0.5	1	0.5	1	1	n/a	n/a	n/a	3
	X-Ray	1.009	1	0.5	2.032	0.506	1.008	1.013	n/a	n/a	n/a	2.935
	lmbench	2.712	1.004	0.501	2.095	0.515	1.845	2.022	n/a	n/a	n/a	2.838
Throughput MULTIPLY F64 (cycles)	Actual	1	1	0.5	2	0.5	1	1	n/a	n/a	n/a	4
	X-Ray	1	1	0.5	3.998	0.506	1	1	n/a	n/a	n/a	3.957
	lmbench	2.86	1.004	0.501	2.209	0.515	2.687	2.022	n/a	n/a	n/a	3.748
NR F64 (cycles)	Actual	32	32	32	8	128	8	16	n/a	n/a	n/a	24
	X-Ray	32	32	32	8	128	8	16	n/a	n/a	n/a	24
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	n/a	n/a	n/a	!exist
FMA F64 (cycles)	Actual	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	n/a	n/a	n/a	FALSE
	X-Ray	FALSE	TRUE	TRUE	FALSE	TRUE	FALSE	FALSE	n/a	n/a	n/a	FALSE
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	n/a	n/a	n/a	!exist

Table 1. Summary of Experimental Results for CPU Features

Feature	Tool	UltraSPARC IIIi	R12000	Power 3	Pentium 4	Itanium 2	Athlon MP	Opteron 240	ARM SA1110	xScale PXA250	Xtensa LX	R4400
L1 Cache Capacity (KB)	Actual	64	32	64	8	16	64	64	8	32	16	16
	X-Ray	64	32	64.5	8	16	64	64	8	32	16	16
	lmbench	64	32	64	8	16	64	64	8	!os	!os	16
L1 Cache Associativity (count)	Actual	4	2	128	4	4	2	2	32	32	2	1
	X-Ray	4	2	129	4	4	2	2	32	32	2	1
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!exist	!os	!os	!exist
L1 Cache Block Size (bytes)	Actual	32	16	128	64	64	64	64	32	32	64	16
	X-Ray	32	16	128	64	64	64	64	32	32	64	16
	lmbench	32	32	128	64	64	64	64	32	!os	!os	16
L1 Cache Latency (cycles)	Actual	2.000	2.000	2.000	2.000	2.000	2.000	3.000	2.000	3.000	2.000	3.000
	X-Ray	2.093	2.011	1.986	4.109	2.009	1.992	3.090	1.993	3.053	1.969	2.956
	lmbench	2.000	2.010	2.023	2.226	2.005	3.170	3.110	2.002	!os	!os	3.140
L2 Cache Capacity (KB)	Actual	1024	2048	6144	512	256	512	1024	n/a	n/a	n/a	n/a
	X-Ray	!os	!os	!os	512	256	576	1088	n/a	n/a	n/a	n/a
	lmbench	1024	2048	6144	512	256	512	1536	n/a	n/a	n/a	n/a
L2 Cache Associativity (count)	Actual	?	?	?	8	8	16	16	n/a	n/a	n/a	n/a
	X-Ray	!os	!os	!os	8	8	18	17	n/a	n/a	n/a	n/a
	lmbench	!exist	!exist	!exist	!exist	!exist	!exist	!exist	n/a	n/a	n/a	n/a
L2 Cache Block Size (bytes)	Actual	64	128	128	128	128	64	64	n/a	n/a	n/a	n/a
	X-Ray	!os	!os	!os	128	128	64	64	n/a	n/a	n/a	n/a
	lmbench	64	128	128	128	128	64	64	n/a	n/a	n/a	n/a
L2 Cache Latency (cycles)	Actual	?	?	?	?	6.000	?	?	n/a	n/a	n/a	n/a
	X-Ray	!os	!os	!os	41.530	5.980	36.000	22.800	n/a	n/a	n/a	n/a
	lmbench	4.410	13.860	17.190	20.360	6.104	19.907	9.819	n/a	n/a	n/a	n/a
Main Memory Latency (cycles)	Actual	?	?	?	?	?	?	?	?	?	?	?
	X-Ray	!os	761.920	!os	36.747	297.650	471.350	136.210	41.259	134.986	10.793	13.820
	lmbench	172.810	122.060	18.317	17.963	301.878	432.911	140.947	41.805	!os	!os	15.060

Table 2. Summary of Experimental Results for Memory Hierarchy

- **?** – we could not obtain official information about the actual value of this hardware parameter.

X-Ray and lmbench measure some hardware parameters in different units. To allow direct comparison, we normalized lmbench results as follows.

- lmbench measures the processor clock cycle c_{lmb} and various latencies l_{lmb} in nanoseconds. We compute the processor frequency in MHz as $f_{lmb} = 1000 \div c_{lmb}$, and latency in cycles as $l_{lmb} \div c_{lmb}$.
- Instead of measuring instruction throughput, lmbench measures available instruction parallelism p_{lmb}. We compute instruction throughput in cycles as $t_{lmb} = l_{lmb} \div p_{lmb}$, where l_{lmb} is the latency in cycles of the corresponding instruction, computed as shown above.

We now discuss some of the more interesting results.

5.1. UltraSPARC IIIi and R12000

X-Ray measured all parameters accurately on both architectures. Lmbench measured all parameters it supports accurately on the R12000, but gave less accurate results on the UltraSPARC IIIi, especially for instruction latency and throughput.

5.2. Power 3

X-Ray detected an integer fused multiply-add instruction although there is not one in the ISA. We verified that even though our measurement sequence (`r0=r0+r0*r0`) is translated into separate dependent MULTIPLY and ADD instructions, the hardware can achieve the same throughput as if there was no ADD. Therefore, the multiply-add sequence can be used to generate high-performance code for this architecture. X-Ray measured the data cache as 129-way set associative instead of 128-way set associative. This resulted in capacity of 64.5KB compared to the documented one of 64KB. Lmbench gave accurate results on the parameters it is able to measure.

5.3. Pentium 4 Xeon

These processors feature two double-pumped integer ALUs, which led X-Ray to believe that the frequency is twice higher than the actual. This is not a problem as long as all other timings are measured relative to this

frequency. Indeed, as Table 1 shows, all timing values measured by X-Ray are twice larger than the actual values (except for Throughput ADD I32).

The throughput of ADD I32 is quite interesting. Because of the two integer ALUs and integer ADD latency of 0.5 cycles, we expect an effective throughput of 0.25 cycles, which translates to 0.5 cycles relative to our frequency. Instead X-Ray measured 0.679, which is 50% greater (3 integer adds per cycle instead of 4). This problem occurs because the instruction cache on Pentium 4 can only deliver 3 instructions per cycle to the instruction dispatch engine, preventing the integer pipes from achieving the maximum throughput. Although we do not present the results here, X-Ray was able to measure accurately the number of vector registers (MMX, SSE, and SSE2), as well as the latencies and throughputs of the corresponding SIMD instructions.

Lmbech results are close to those of X-Ray but noticeably less accurate. However, lmbench found the advertised frequency instead of double the value as X-Ray did.

5.4. Itanium 2

X-Ray produced accurate results for all parameters. Lmbench results were slightly less accurate, with one major problem – the throughput of ADD I32. This processor is able to execute 6 independent ADD operations per cycle, and lmbench measured throughput of only 0.469. X-Ray measured the correct throughput of 0.169.

Measuring the number of F32 registers illustrates a different point. This processor has 128 floating-point registers but two of them 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.

5.5. Athlon MP and Opteron 240

X-Ray measured all CPU feature parameters accurately. Lmbench gave less accurate results, especially for instruction throughput.

The memory hierarchy numbers for these machine are interesting because they expose the fact that the L1 and L2 caches 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. 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 self-optimizing code that wants to perform an optimization like cache tiling.

5.6. Xtensa LX

Xtensa LX is a configurable, extensible processor core designed by Tensilica. The hardware parameters of different Xtensa LX cores can be very different. This feature of the Xtensa LX processor makes it a challenging target for X-Ray.

Frequency

The processor frequency measured by X-Ray (343.225MHz) was 2% different from the actual value (350MHz). This inaccuracy can be explained by the loop overhead incurred from the code shown in Figure 3(d). The 256 case statements have a latency of 1 cycle for a total of 256 cycles, and the loop-back code at the end has a total latency of 5 cycles. Therefore the measurement error is $5 \div 261 \approx 2\%$. While we can partially compensate for this [8], we do not feel that this is necessary because we only use frequency to measure other parameters relative to it (in clock cycles).

Number of Registers

X-Ray measured 11 integer registers and 17 floating-point registers, while there are 16 architecturally available of both types. We verified that register spills occurred when using more than 11 integer variables or more than 16 floating-point variables. The cost of spilling one floating-point registers was not sufficient for X-Ray to declare that a phase transition had happened. Of course, we could lower the threshold at which X-Ray declares that a phase transition has happened, but this might have a negative impact on other platforms where measurement noise is relatively high. In practice, it is likely that this performance penalty will not be statistically significant even if the number of variables is one or two more than the number of available registers. We are also looking into more robust phase transition detection algorithms.

Other Configurations

- We introduced two more integer ADD and one more integer MULTIPLY functional units. X-Ray correctly measured the new Throughput ADD

I32 and Throughput MULTIPLY I32 as 0.335 and 0.496 cycles respectively.
- We changed the data cache configuration to 6KB, 3-way set associative with 32 byte blocks. X-Ray correctly measured the new cache parameters.
- We replaced the data cache with a 128-bit single precision fixed point SIMD unit. After the appropriate descriptions were added, X-Ray correctly measured the latency and throughput of ADD and MULTIPLY, along with the number of vector registers (1.000, 1.977, 0.998, 0.996, and 16 respectively).

5.7. MIPS R4400

X-Ray measured all parameters accurately. Lmbench accurately measured all parameters it supports. There are two details worth noting.

- The latency of MULTIPLY I32 measured by X-Ray is about 15 cycles, while the actual latency is 12 cycles. The reason behind this mismatch is that the R4400 has special registers `hi` and `lo`, which hold the result of integer multiply. Therefore the code sequence we use (`r0=r0*r1`) is translated to the assembly sequence $\langle$`hi,lo`$\rangle$ `= r0 * r1; r0 = lo; noop; noop`. The two `noop` instructions are necessary because access to `lo` is asynchronous and the compiler needs to make sure that the value can be copied before it is destroyed. Therefore, although the latency of an integer multiply is 12 cycles, it cannot be sustained by code.
- X-Ray measured significantly fewer registers than are architecturally available. We examined the generated assembly files and confirmed that it is the policy of the native compiler to reserve the rest of the registers.

6. Future Work

We are actively developing new micro-benchmarks inside the X-Ray framework. Our current focus includes measuring other parameters of the memory hierarchy such as parameters of instruction caches, and replacement policy and bandwidth of different cache levels, as well as determining all bundles of instructions that can be issued in a single CPU cycle at a sustained rate.

X-Ray can be downloaded at `http://iss.cs.cornell.edu/Software/X-Ray.aspx`.

Acknowledgements

We thank Darin Petkov for performing the Xtensa measurements, and Carl Staelin for his assistance with lmbench, as well as for numerous constructive suggestions at improving X-Ray and this paper.

References

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

[2] J. Dongarra, S. Moore, P. Mucci, K. Seymour, and H. You. Accurate cache and TLB characterization using hardware counters. In *Proceedings of the International Conference on Computational Science (ICCS) 2004, Krakow, Poland*, 2004.

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

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

[5] L. McVoy and C. 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, Jan. 1996.

[6] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. W. Singer, J. Xiong, F. Franchetti, A. Gačić, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".

[7] R. H. Saavedra and A. J. Smith. Measuring cache and TLB performance and their effect of benchmark run. Technical Report CSD-93-767, Feb. 1993.

[8] C. Staelin and L. McVoy. mhz: Anatomy of a microbenchmark. In *USENIX 1998 Annual Technical Conference, January 15–18, 1998. New Orleans, Louisiana*, pages 155–166, Berkeley, CA, USA, June 1998.

[9] R. C. Whaley, A. Petitet, and J. 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`).

[10] K. Yotov, X. Li, G. Ren, M. Garzaran, D. Padua, K. Pingali, and P. 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".

[11] K. Yotov, K. Pingali, and P. Stodghill. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS'05*, June 2005.