Automatic Measurement of Instruction Cache Capacity

Kamen Yotov, Sandra Jackson, Tyler Steele
kyotov@cs.cornell.edu, {sjj3,ths22}@cornell.edu

Keshav Pingali, Paul Stodghill
{pingali,stodghil}@cs.cornell.edu

Department of Computer Science,
Cornell University,
Ithaca, NY 14853

Abstract.

There is growing interest in autonomic 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 may need the values of 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.

We have built a system called X-Ray[1], which uses micro-benchmarks to measure such parameter values automatically. The micro-benchmarks currently implemented in X-Ray can determine the latency of various instructions, the existence of important instructions like fused multiply-add, the number of registers of various kinds, and parameters of the memory hierarchy.

In this paper, we discuss how X-Ray determines the capacity of the instruction cache (I-cache), which is needed for important optimizations such as loop unrolling. We present the micro-benchmark used in X-Ray to measure I-cache capacity, the experimental methodology used to obtain accurate estimates, and experimental results on a large number of current platforms.

[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.

1 Introduction

There is growing interest in self-optimizing systems that can optimize their own behavior on different platforms without manual intervention [2, 8, 5]. 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 parameter values such as the number of registers and the capacity of the L1 cache [8, 9].

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 in ATLAS is limited by the number of registers on the target architecture. 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.

To address this need, we have developed a framework called X-Ray, which can be used to implement micro-benchmarks to measure relevant values of hardware parameters automatically. For portability, X-Ray is entirely implemented in ANSI C'89. Currently, X-Ray can determine the latency of various instructions, the existence of important instructions like fused multiply-add, the number of registers of various kinds, and parameters of the memory hierarchy.

In this paper, we describe how X-Ray measures the capacity of the instruction cache (I-cache). Neither well known benchmarks [3, 6], nor existing tools [4, 7] attempt to measure this parameter.

The I-cache capacity is needed in the implementation of important optimizations like loop unrolling, which is used to reduce loop overhead, to prepare the loop body for scheduling of operations, to improve processor pipeline utilization, to enable register allocation of array values, etc. [1]. If the loop is unrolled too few times, loop overhead can be substantial, and pipeline and register utilization can suffer, lowering performance. On the other hand, if the loop is unrolled too many times, I-cache misses may cause performance to drop. Therefore, compilers need I-cache capacity to estimate how many times a loop should be unrolled.

An example on Intel Itanium 2 is presented in Figure 1. This figure shows the sensitivity of performance to the unrolling of the K loop (KU) of Matrix-Matrix Multiply in ATLAS for two different cache blocking factors (NB). We have verified with hardware counters that the performance drop observed for $KU > 9$ is caused by excessive number of instruction cache misses.

The rest of this paper is organized as follows. In Section 2, we give an overview of the X-Ray framework. The major challenge is to ensure that the C compiler does not restructure the micro-benchmarks thereby polluting the timing results,

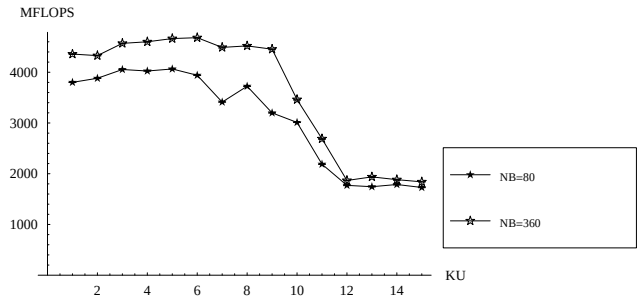


Fig. 1. Sensitivity of performance to K-unrolling on Intel Itaniun 2 in ATLAS

while enabling performance critical optimizations such as register allocation. In Section 3, we describe the micro-benchmark we use for measuring I-cache capacity. In Section 4, we present experimental results on a number of modern high-performance processors. We also compare I-cache capacity estimates from X-Ray with published values for these architectures. These comparisons show that the estimates of I-cache capacity that X-Ray produces are accurate to within 3% on most architectures.

2 The X-Ray Framework

Hardware parameters are measured by X-Ray *micro-benchmarks*. Figure 2 presents the general structure of a micro-benchmark in the X-Ray framework.

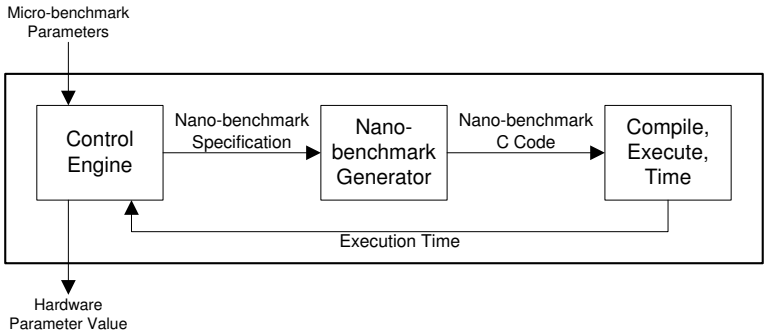


Fig. 2. A micro-benchmark in X-Ray

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 computations 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.

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);
```

Fig. 3. Nano-benchmark timing

Figure 3 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 3 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 $\texttt{measure}_S$ is shown in Figure 4(a). This code incurs considerable loop overhead, so we unroll the loop U times (Figure 4(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 4(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.

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 4(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 4(c) and 4(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 4(c). Generating code of the form shown in Figure 4(d) is more complex and requires the first element of the tuple to be a

```
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)

Fig. 4. Implementation of $\texttt{measure}_S$

function f : integer $\rightarrow$ string, which computes the code for statement S_i from the case label i.

2.3 Implementing a new micro-benchmark

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 4(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.

The X-Ray implementation of many useful micro-benchmarks is described in detail in [11, 10].

3 Measuring I-cache capacity

To estimate I-cache capacity, X-Ray measures the execution time of code sequences of different sizes. These sequences are carefully chosen so that the processor can run them at full speed unless they are too long to fit completely in the I-cache.

More precisely, the X-Ray micro-benchmark generates a sequence of nano-benchmarks. Each nano-benchmark measures the average time needed to execute one statement of a code sequence of specific length N. The micro-benchmark uses these nano-benchmarks to determines the largest value of N for which there is no significant increase in the average execution time per statement. The capacity of the I-cache is declared to be the binary code size for this longest code sequence.

Although this is straight-forward in theory, there are several practical problems we had to address to make this idea work.

3.1 Nano-benchmark

Figure 5 shows the nano-benchmark generated by X-Ray. The basic statements used in the loop body by X-Ray are assignment statements that increment one integer variable with the value of another integer variable. The C compiler is also advised to assign these variables to registers. Therefore, most compilers will map each assignment statement to a single register-to-register integer add instruction since such an instruction is available on all ISAs.

Each case statement in Figure 5 consists of a number of independent assignment statements. The idea is to provide enough instruction level parallelism in each case statement to avoid stalls caused by dependencies. This way we ensure that instructions are dispatched at the highest possible rate by the processor, so the slowdown caused by I-cache misses will be prominent. We have found that using four independent assignment statements per case is adequate on all current architectures.

```
volatile int v = 0;
volatile int p0 = v, p1 = v, p2 = v, p3 = v, p4 = v;

switch (v)
{
case 0 :
    i = R/N;
    ts = now ();
start :
case 1 : {p1+ = p0; p2+ = p0; p3+ = p0; p4+ = p0; }
case 2 : {p1+ = p0; p2+ = p0; p3+ = p0; p4+ = p0; }
    ...
case N : {p1+ = p0; p2+ = p0; p3+ = p0; p4+ = p0; }
    if(−−i)
        goto start;
finish :
    te = now ();
    if(!v)
        return te − ts;
}

v = p0;  v = p1;  v = p2;  v = p3;  v = p4;
```

Fig. 5. Nano-benchmark code generated by X-Ray for measuring I-cache capacity

Therefore, the X-Ray nano-benchmark is parameterized by N, the number of cases in the switch statement, and B, the number of independent assignment statements per case. Not surprisingly, the specification X-Ray uses for its I-cache nano-benchmark is the following.

$$S_{N,B} = \langle [1, N] \mapsto \{p_1 += p_0; p_2 += p_0; \ldots; p_B += p_0; \}, \langle p_0, p_1, \ldots, p_B : int \rangle \rangle$$

Currently we measure the binary code size of the sequence by using an extension to the C language available in the GCC family of compilers, namely taking the address of a code label. Using this feature, the binary size of the code shown above would be computed by `(char *)&&finish - (char *)&&start`.

We are currently looking into other ways of doing this measurement if a compiler that supports this feature is not available. One possibility is to generate a program listing which includes the generated assembly instructions along with their code addresses and deduce the addresses of the labels of interest by analyzing the listing.

3.2 Micro-benchmark

Figures 7 and 8 show how the average execution time per statement of the nano-benchmark varies as a function of the computed value of binary code size. It can be seen that on many architectures such as the IBM Power 4, there are

```
μ ← 0;
σ ← 0;
N ← 256;
while (N < 256 + S)
    τ ← time (S_{N,B}) ;
    μ ← μ + τ;
    σ ← σ + τ^2;
    N ← N + 1;
μ ← μ ÷ S;
σ ← sqrt((σ − μ^2 × S) ÷ (S − 1));
while (time (S_{N,B}) < μ + T × σ)
    N ← N × 2;
R ← N;
L ← N ÷ 2;
while (R − L > 1)
    N ← (R + L) ÷ 2;
    if (time (S_{N,B}) ≥ μ + T × σ)
        R ← N;
    else
        L ← N;
return L × B;
```

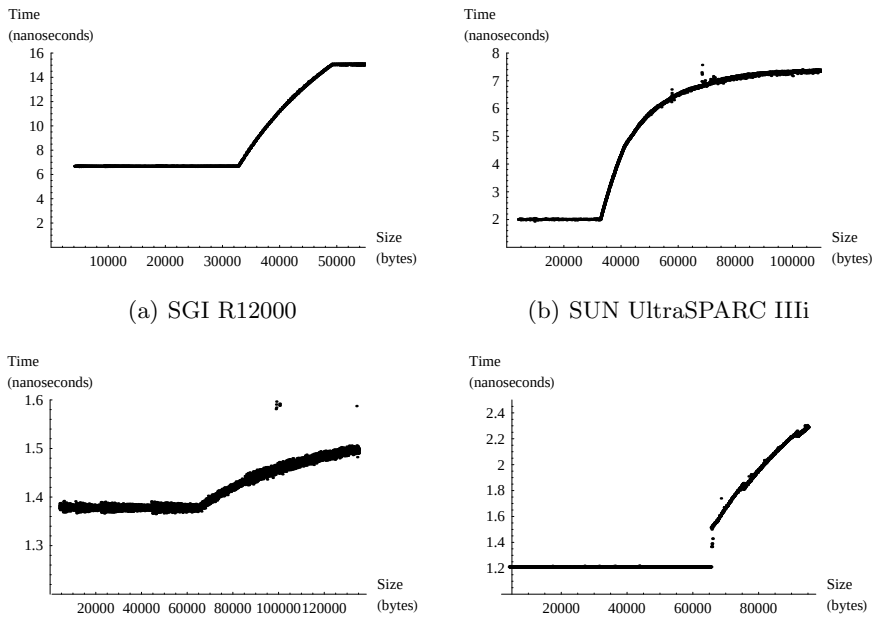
Fig. 6. Control engine script for I-cache micro-benchmark

significant fluctuations in I-cache access time even when loop bodies are small enough to fit comfortably in the I-cache. In particular, access time can increase significantly when the size of the loop body is increased by a small amount, but decreases when the size of the loop body is increased further. This effect is not entirely noise because some part of it is reproducible. Figures 9 and 10 show the distribution of the average access times for various architectures when the loop body is small enough to fit in the I-cache.

Consequently, the micro-benchmark cannot just look for an increase in the statement execution time to determine the capacity of the I-cache; furthermore, performing the measurement for each code size some number of times and using the average time does not always help since some fluctuations occur for different values of code size.

The solution used by X-Ray is to estimate first the mean and standard deviation of the fluctuations in statement execution time when the loop body is small enough to fit in the I-cache. An increase in execution time is declared to be significant when the jump exceeds some multiple of the measured standard deviation of the fluctuations.

More precisely, X-Ray measures the statement execution times for nano-benchmarks of the form shown in Figure 5 for $N \in [256, 256 + S - 1]$, where the sample size S is a parameter we currently set to 8. It computes the mean μ and the standard deviation σ of these times, and uses $\mu + T \times \sigma$ as the threshold



(a) SGI R12000

(b) SUN UltraSPARC IIIi

(c) IBM Power 4

(d) IBM Power 5

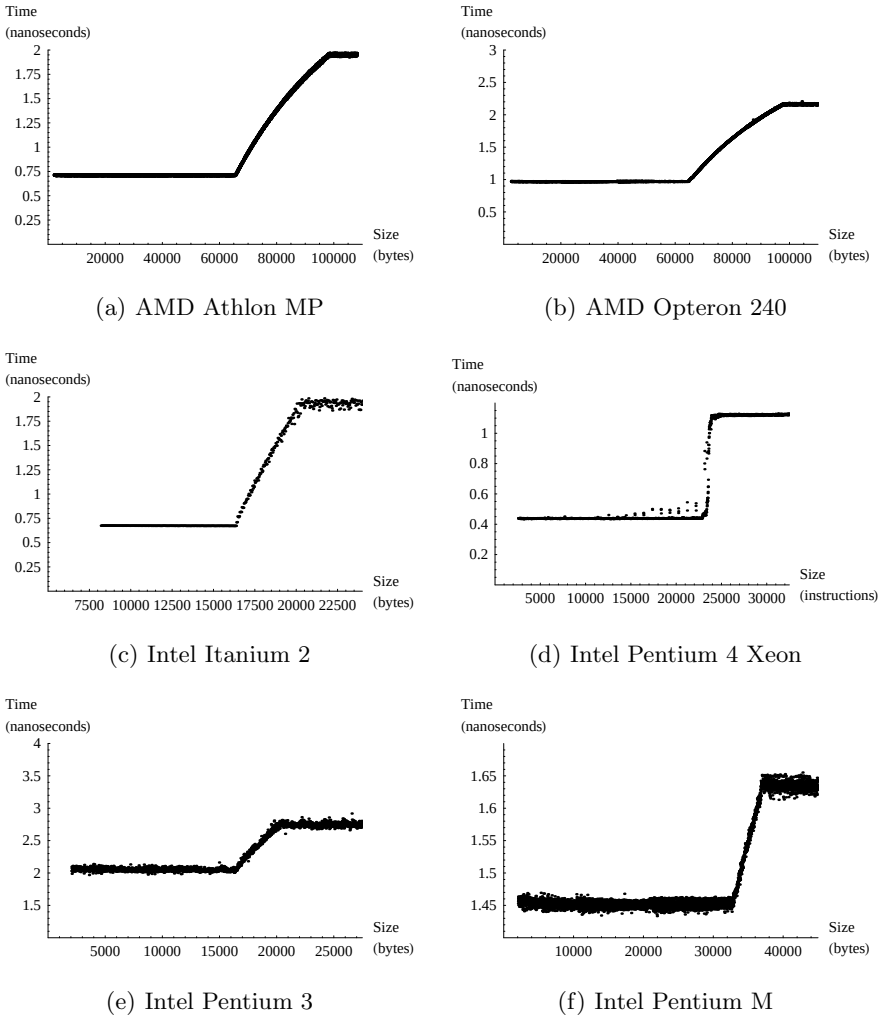
Fig. 7. Execution time per statement on RISC architectures

above which a change in execution time is declared to be significant; currently, we set the parameter T to 2 since this seems to work well in practice.

The sensitivity of Intel Pentium 4 is shown in Figure 8(d). On this architecture we observed that for some values of N, well before the I-cache edge, there are significant, but isolated fluctuations. To avoid confusing these fluctuations with the actual edge, X-Ray applies a smoothing function, which takes the minimum timing in a small neighborhood of N, namely $\left[N - \frac{I}{2}, N + \frac{I}{2}\right]$. I is a parameter of the I-cache micro-benchmark. In our experiments we found that $I = 5$ works well in practice.

These considerations lead to the actual control engine algorithm specified in Figure 6.

This code can be summarized as follows. First, the control engine computes the mean μ and the standard deviation σ of the timings for $N \in [256, 256 + S - 1]$. Then it starts with $N = N_{min} = 256$ and doubles N until timing exceeds the threshold $\mu + T \times \sigma$ for some $N = N_{max}$. After that it performs a binary search in the interval $[N_{max} \div 2, N_{max})$ to find the maximum N, whose timing is below the threshold $\mu + T \times \sigma$. Finally it returns the number of instructions in the sequence, which is $L \times B$. The actual size of the binary code is computed as part of the nano-benchmark (executed for $N = L$) as discussed above.



(a) AMD Athlon MP

(b) AMD Opteron 240

(c) Intel Itanium 2

(d) Intel Pentium 4 Xeon

(e) Intel Pentium 3

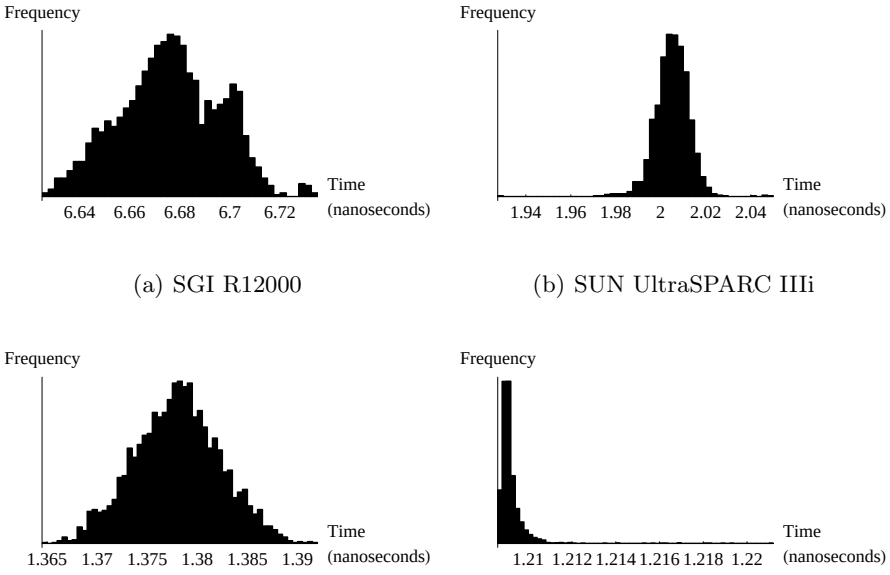
(f) Intel Pentium M

Fig. 8. Execution time per statement on x86 and EPIC architectures

4 Experimental Results

We tried the I-cache capacity micro-benchmark, described in Section 3 on a variety of modern architectures. The results obtained on ten of them are presented in Table 1. X-Ray was able to estimate the I-cache capacity within 3% of the actual value, except on the Intel Pentium 4, where the error was about 6%.

All running times are reported for a smoothing interval of $I = 5$. This is only really necessary on the Pentium 4 architecture. No smoothing ($I = 1$) is required for other architectures, which can dramatically decrease runtime (up to five times in this case).



(a) SGI R12000

(b) SUN UltraSPARC IIIi

(c) IBM Power 4

(d) IBM Power 5

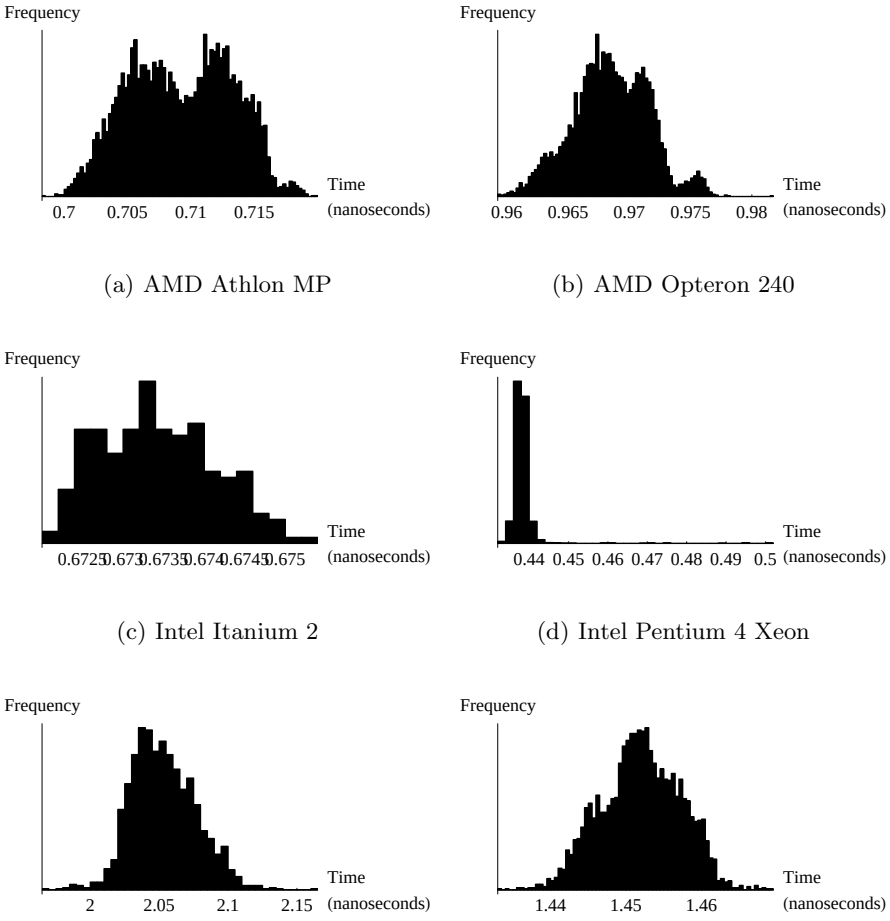
Fig. 9. Hit-time distribution on RISC architectures

Architecture	Actual Size	Measured Size	Error	Time (seconds)
SGI R12000	32768 bytes	32108	-2.01%	534
SUN UltraSPARC IIIi	32768 bytes	32768	0.00%	321
IBM Power 4	65536 bytes	64956	-0.89%	350
IBM Power 5	65536 bytes	65016	-0.79%	365
AMD Athlon MP	65536 bytes	65496	-0.06%	904
AMD Opteron 240	65536 bytes	65480	-0.09%	647
Intel Itanium 2	16384 bytes	16352	-0.20%	101
Intel Pentium 4 Xeon	12000 uops	11245	-6.29%	187
Intel Pentium 3	16384 bytes	15940	-2.71%	285
Intel Pentium M	32768 bytes	33040	0.83%	295

Table 1. I-cache capacity experimental results

4.1 Intel Pentium 4

The Intel Pentium 4 is an interesting architecture because it translates x86 CISC instructions to RISC-like micro-ops before caching them in its I-cache. Moreover, the I-cache does not have a conventional design but is organized as a trace cache. Because of all this, the information we can determine about the capacity of this cache is limited. The architecture manual reports I-cache size in number of micro-ops, and we have verified that each of our addition statements translates to a



(a) AMD Athlon MP

(b) AMD Opteron 240

(c) Intel Itanium 2

(d) Intel Pentium 4 Xeon

(e) Intel Pentium 3

(f) Intel Pentium M

Fig. 10. Hit-time distribution on x86 and EPIC architectures

single CISC instruction which in turn translates to a single micro-op according to the architecture manual. Therefore X-Ray is able to measure the capacity in micro-ops.

However, this information may not be very useful for self-optimizing software systems because to use it, one needs to consider how many micro-ops each CISC instruction translates to, and to avoid the cases of isolated performance hits visible in Figure 8(d).

5 Conclusions and Future Work

To the best of our knowledge, X-Ray is the first system that can measure I-cache capacity. The micro-benchmark seems to be fairly accurate on all current architectures. The techniques described in this paper for eliminating fluctuations and for smoothing are useful in other contexts as well. For example, we successfully applied them to improve the accuracy of the micro-benchmark for measuring the number of registers in X-Ray.

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 bandwidth of different levels of the memory hierarchy, 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.

References

1. R. Allan and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann Publishers, 2002.
2. Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".
3. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, 1990.
4. 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.
5. Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan W. Singer, Jianxin Xiong, Franz Franchetti, Aca Gačić, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nick Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".
6. Rafael H. Saavedra and Alan Jay Smith. Measuring cache and TLB performance and their effect of benchmark run. Technical Report CSD-93-767, February 1993.
7. 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.
8. 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).
9. Kamen Yotov, Xiaoming Li, Gang Ren, Maria Garzaran, David Padua, Keshav Pingali, and Paul Stodghill. Is search really necessary to generate high-performance BLAS? *Proceedings of the IEEE*, 93(2), 2005. special issue on "Program Generation, Optimization, and Adaptation".
10. Kamen Yotov, Keshav Pingali, and Paul Stodghill. Automatic measurement of memory hierarchy parameters. In *SIGMETRICS'05*, June 2005.
11. Kamen Yotov, Keshav Pingali, and Paul Stodghill. X-ray: A tool for automatic measurement of hardware parameters. In *QEST'05*, September 2005.