X-Ray

A Tool for Automatic Measurement of Architectural Parameters

ABSTRACT

We describe the design and implementation of X-Ray, which is a tool that automatically measures and reports a large number of architectural parameters such as the frequency of the CPU, the latency and throughput of instructions, the existence of instructions such as a fused multiply-add, the number of registers, the organization of different levels of the memory hierarchy etc. In principle, some of this information can be obtained by consulting the appropriate manuals, but in practice, these manuals are either non-existent or obsolete. Therefore, a tool like X-Ray is useful in many contexts such as performance analysis of programs, and in self-tuning software systems like ATLAS and FFTW.

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 has a micro-benchmark generator that can be used to automatically produce the large number of micro-benchmarks needed for architecture parameter measurement.

In this paper, we describe the design of X-Ray, and compare it with existing tools. Our experiments show that X-Ray produces more accurate results on all the platforms we tested.

Categories and Subject Descriptors

C.4 [**Computer Systems Organization**]: *Measurement techniques*; D.4.8 [**Software**]: Operating Systems—*Performance: Measurements*; D.3.4 [**Software**]: Programming Languages—*Processors: Code Generation, Compilers, Optimization, Retargetable Compilers*

General Terms

Measurement, Performance, Experimentation

Keywords

Micro-benchmarks, Platform parameters, Self-tuning code

1. INTRODUCTION

Self-tuning software systems such as ATLAS [7], FFTW [3] and Spiral [5] are attracting a lot of attention from the research community because they provide an inexpensive way of generating highly optimized numerical libraries. To produce a library routine, these systems use a generate-and-test approach to search a large space of different implementations of that routine and find the one that gives best performance.

The search spaces in question are usually infinite; for example, a loop can be unrolled an unbounded number of times, so it is impossible to search the entire space exhaustively even for this trivial example. To complete the search process in reasonable time, the search space must be bounded. Therefore, these systems run micro-benchmarks to determine certain architectural parameters such as the number of registers and the size of the L1 data cache. These architectural parameters are used to limit the search; for example in ATLAS, the number of registers found by the micro-benchmarks is used to bound the search for how many times the innermost loop in matrix multiplication should be unrolled for optimal performance.

This approach to optimization in library generators can deal only with a small number of hardware resources. For example, optimizing the library for an additional level of cache can add an order of magnitude to the installation time. In practice, existing library generators deal with this problem by limiting the set of hardware resources that they optimize for. ATLAS for example, optimizes the use of registers and L1 data cache in linear algebra routines, but does not attempt to optimize the routines for other levels of the memory hierarchy.

[8] discusses how model-based optimization can be used to reduce the amount of search that a library generator must do. Performance models are used to predict good values for code optimization parameters such as loop unrolling factors. For some parameters, these values are close enough to optimal; for others, a quick local search can be used to determine the optimal value from the predicted value. It has been shown that this model-based approach is able to produce code that is comparable with that produced by the current generation of library generators.

The performance models used in this approach obviously depend on certain architectural parameters such as the size of each cache, the number of registers, the latency of certain instructions, etc. It is important to provide accurate values for these parameters. Furthermore, in the context of self-tuning, self-optimizing software, it is desirable that these architectural parameters be determined automatically by the

system by running carefully crafted micro-benchmarks.

In this paper, we present a set of micro-benchmarks that provide more accuracy and coverage than previous ones. We have implemented these algorithms in a system called X-Ray. For portability, X-Ray is implemented completely in C [2]. 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 will 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 will discuss the micro-benchmark algorithms that we have developed for determining features of the CPUs. Section 4 will discuss micro-benchmarks for measuring features of the memory hierarchy. We will present experimental results in Section 5 to show the effectiveness of our approach. In Section 6, we will discuss future work and our conclusions.

2. OVERVIEW OF X-RAY

Figure 1 is an overview of the approach taken by X-Ray. Rather than provide a fixed set of micro-benchmarks, X-Ray uses a micro-benchmark *generator* which can be used to produce the large number of micro-benchmarks that are needed for measuring architectural parameters. The key module for this is labelled *code generator* in Figure 1. The input to this module is a *configuration*, which can be thought of as an encoding of the desired micro-benchmark, and the output is a C program. Configurations are described in more detail later in this section.

To measure architectural parameters, X-Ray uses a set of configuration generators which provide a stream of configurations to the code generator. The C code produced by the code generator is compiled and run on the target architecture, and the results are used to determine what other configurations should be generated. Once this process is complete, the complete architectural description is produced.

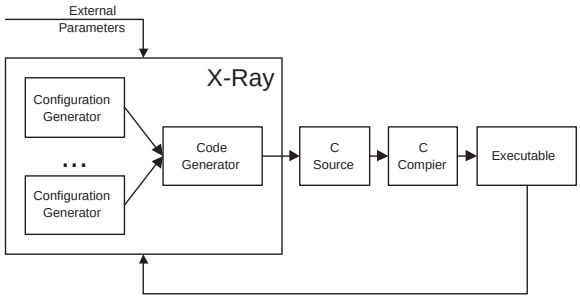

Figure 1: The X-Ray system architecture

Although we designed X-Ray so that it can run out of the box, there are a small number of parameters that must be provided by the user to X-Ray before it can run. One such parameter is t_{min}, which is discussed in Section 2.2. The other parameters control how performance jumps and plateaus in the experiments are determined. X-Ray provides default values for these parameters.

2.1 Operating system calls

The C language as defined in [2] does not specify all of the functionality that is required in order 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 some operating systems do not support this interface in native mode. Because of this, we decided not to try to build a system that is operating system independent. Therefore, we have coded X-Ray so that it uses the necessary OS API's on Windows, Linux, Solaris, etc. We feel that this is a reasonable tradeoff - accessing the OS-specific API's allows us to produce much better results. There are far fewer operating systems than hardware configurations.

2.2 Measuring performance accurately

Even with access to an operating system's API for high performance timers, accurately timing operations that take a small number of CPU cycles to execute is difficult in C.

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, and divide that time by R. How do we choose R? If R is too small, the time for execution cannot be measured accurately, whereas if R is too big, the experiment will take longer than it needs to.

In X-Ray, one of the external parameters that must be provided is t_{min}, the minimal time that experiments must execute for accurate measurements. This parameter is platform dependent, but we have found that a value of $t_{min} = 0.5s$ works well in practice.

To determine R given t_{min}, we start by setting R to 1, and exponentially grow R until the experiment runs for at least t_{min} seconds, as shown below.

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

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, $\texttt{measure}_S(R)$ is a procedure that executes R repetitions of statement S, whose implementation we discuss next.

One implementation of the $\texttt{measure}_S$ procedure is the following.

```
measure_S(R)
{
   t_start = now();
   i = R;
loop:  S;
   if (--i)
      goto loop;
   t_end = now();
   return t_end - t_start;
}
```

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.

```
measure_S(R)
{
    t_start = now();
    i = R / U;
loop:
    S;
    S;
    ...repeat U times...
    S;
    if (--i)
        goto loop;
    t_end = now();
    return t_end − t_start;
}
```

2.3 Preventing unwanted compiler optimizations

Although the code shown above can be used to accurately measure the time for executing a statement S, optimizations by the back-end 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 the optimizations in the back-end C compiler. Unfortunately, disabling all optimizations 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 back-end 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 2. 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. This can be achieved by `use` assigning the each variable to a `volatile` global variable.

2.4 Configurations and Code Generator

In X-Ray, micro-benchmark code of the form shown in Figure 2 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 2.

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

Figure 2: Pseudo-code for measuring execution time of statement S

As we will see in Sections 3 and 4, 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. In this figure, the number of case labels W is the smallest multiple of n, not smaller than U. The configuration for this sequence is written 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.

An statement's *latency*, L_S, is the number of cycles after it is issued that the result of the instruction becomes available for subsequent instructions.

An statement's *initiation interval*, II_S, is the minimum number of cycles between the issue of two consecutive and independent instructions of the same type to the same functional unit. A fully-pipelined functional unit will have an initiation interval of 1 cycle. We will determine the initiation intervals of instructions, not functional units. If there are multiple functional units that can execute an instruction, the initiation interval will be less than 1.

3.1 CPU Frequency

The CPU frequency, F_{CPU}, is an important platform parameter, as many other parameters are measured relative

```
measure_S(R)
{
    initialize;
    volatile int v = 0;
    switch (v)
    {
    case 0:
        i = R/U;
        t_start = 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_end = now();
        if (!v)
            return t_end - t_start;
    }
    use;
}
```

Figure 3: Pseudo-code for measuring execution time of statement sequence $S_1|S_2|\ldots|S_n$

to it (i.e. in clock cycles). We will use the configuration $\langle p_1^1 = p_1^1 + p_2^1, \texttt{int}, 2\rangle$ for measuring CPU frequency. Because each statement instance depends on the previous instance, all instances need to be executed sequentially. Provided that an integer addition instruction has both latency and initiation interval of 1, the frequency F_{CPU} is the number of integer addition instructions that can be executed per second.

3.2 Instruction Properties

3.2.1 Latency

To measure the latency of an arbitrary instruction, we choose a C operator, op, that will compile to this instruction. Next, we compose a statement, S, of the form $p_1 = p_1$ op p_2. Since consecutive instances of S are dependent, $L_S = F_{CPU} \times t_S$, where $t_S = \texttt{Time}(C_S)$.

3.2.2 Initiation Interval

We measure instruction initiation interval by scheduling an increasing number of independent instructions until we determine that the CPU is saturated.

For example, to compute the initiation interval of double-precision floating-point multiply, we will use the configuration $C_i = \langle S_i, \texttt{double}, i+1\rangle$, where S_i is an statement that specifies i independent multiplies. Here is an example for S_3:

```
{
  p_1^1 = p_1^1 * p_4^1;
  p_2^1 = p_2^1 * p_4^1;
  p_3^1 = p_3^1 * p_4^1;
}
```

To determine saturation, we iterate from $i = 0$ until the performance levels off, say at $i = i_{saturated}$. Then we can compute $II = F_{CPU} \times t_{i_{saturated}}/i_{saturated}$.

Note that we implicitly assume that there are enough registers to saturate the functional unit. The exact constraint is $L/II \leq \#Registers$.

3.2.3 Existence

In certain cases it is not obvious how a certain C statement is translated. One very common case for numerical applications is $p_1^1 = p_1^1 + p_2^1 * p_3^1$. On some platforms, this statement will be compiled into a single fused multiply-add instruction (FMA).

If there is no FMA instruction, the compiler will need an extra register to store the intermediate value. This has an impact on how such sequences of statements need to be scheduled. For example, the ATLAS framework produces different code for the backend C compiler depending upon the existence of a FMA instruction.

In practice we can determine the existence of FMA instructions by comparing the performance of two configurations: one that uses FMA whenever it is available and one that always forces separate multiply and add. The two configurations in question are $C_{FMA} = \langle p_1^1 = p_1^1 + p_1^1 * p_1^1, \texttt{double}, 1\rangle$ and $C_{SMA} = \langle p_2^1 = p_1^1 * p_1^1 | p_1^1 = p_1^1 + p_2^1, \texttt{double}, 2\rangle$.

If the performance of C_{FMA} is approximately the same as C_{SMA} then, we can conclude that there is no FMA instruction.

3.3 Number of Registers

In this description, we will assume for simplicity that we want to determine the number of double-precision floating point registers. Finding the number of registers of different type is analogous and is supported by our implementation.

To determine the number of available registers, we will use a series of configurations C_i of the form $C_i = \langle S_i, \texttt{double}, i \rangle$, where S_i is,

{
$p_1^1 = p_1^1 + p_i^1$;
$p_2^1 = p_2^1 + p_1^1$;
...
$p_i^1 = p_i^1 + p_{i-1}^1$;
$p_1^1 = p_1^1 + p_i^1$;
}

As i grows, performance should remain constant as long as all the $p_k^1, 1 \leq k \leq i$ are allocated in registers. When performance drops, we have determined the point at which the compiler has run out of registers. At that point we conclude that there are $i - 1$ allocatable registers.

Our approach does not always count all of the ISA registers, for a number of reasons.

- Not all of the ISA registers are available for variable allocation. Sometimes specific registers are hardwired to specific values (the most common one being $r_0 = 0$). Sometimes the hardware organization does not allow arbitrary use of all registers (e.g. register windows);
- The backend C compiler may reserve some ISA registers for its own use (as frame pointers, etc.)

We conclude by noting that this approach works for all types of registers, including vector registers used in SIMD instructions by technologies like MMX, SSE, SSE2, 3DNow, AltiVec, etc.

3.4 Functional Units

Determining a processor's functional units accurately is essential for quality instruction scheduling. Unfortunately, determining the types and quantity of functional units is not always possible because of architectural bottlenecks in the processor. For instance, a small number of register file read ports or a small instruction window can prevent instructions from being issued at a rate that saturate the functional units. We discuss our experience with different architectures in Section 5.

One alternative to finding exactly the types and numbers of functional units is to find sets of independent instructions which can be executed at a sustained rate, usually 1 set per cycle. With our system, it is easy to find the rate at which a given set of independent instructions can be sustained. Our system might be used in this way by an instruction scheduler.

In the future, we want our system to automatically determine the performance of interesting sets of instruction sequences. This will require developing strategy a strategy for exploring and pruning the space of possible instruction sequences.

3.5 Parallelism: SMP and SMT

Our system is also able to determine Symmetric Multi-Processor (SMP) and Symmetric Multi-Threading (SMT) configurations. SMP is a term used for platforms with more than one physical processor, while SMT is a way to have more than one execution context (thread, architectural state) in a single physical processor. SMP and SMT are not mutually exclusive and often exist together. SMT implementations share certain physical processor resources and divide others. One of the main purposes of SMT is to "fill-in" slots in the physical execution pipeline, which will otherwise remain empty because of insufficient parallelism in the instruction stream.

To determine that the host is an SMP, we choose a configuration that saturates the integer functional units using the algorithm from Section 3.2.2. Then we spawn an increasing number of threads running instances of this configuration. Any speedups observed will be due to the existence of more than one physical processor. When we observe a performance drop, we know that we have exceeded the number physical processors.

From the OS point of view, an SMT processor will appear to be an SMP processor. That is, when asked how many physical processors are present on a host platform, both SMTs and SMPs return a larger greater than 1. An SMT can be distinguished from an SMP by determining that running the saturating configuration on two of the "processors" does not result in any performance speedup. Furthermore, by testing all pairs of "processors", X-Ray can determine which reside on different physical processors and which are executed concurrently on a SMT CPU.

One important note is that, for our experiments to be reliable, we must tell the OS to "glue" each thread to specific processors in cases where the target platform is SMP and/or SMT.

4. MEMORY HIERARCHY MICRO-BENCHMARKS

Most computers nowadays have multi-level memory hierarchies, such as three levels of cache and a translation look-aside buffer (TLB). Optimizing the cache behavior of programs requires knowledge of each level's parameters. In this section, we present algorithms for automatically measuring memory hierarchy parameters which 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 with caches, whose capacity is not a power of 2.

4.1 Cache Parameters

The three most important parameters of a cache, known as the ABC of caches, are its *associativity*, *block size*, and *capacity* [4]. In this paper, we measure block size and capacity in bytes. For example, the L1 data cache on the Intel P6 has a capacity C of 16KB, an associativity A of 4, and a block size B of 32 bytes. Therefore the cache contains $16384/32 = 512$ blocks which are divided into $512/4 = 128$ sets of 4 blocks each. There are other cache parameters such as replacement policy but we will not consider them in this paper. In particular, we will assume that the replacement policy is least-recently-used (LRU), which is reasonable because almost all caches implement variants of this policy.

Figure 4 shows how a byte is accessed in the Intel P6 cache. For this cache organization, we need 7 bits to index one of the 128 sets and 5 bits to index the appropriate byte

within the 32-byte block. The highest 20 bits are the *tag* bits. Through the rest of the paper, we will refer to the width of the fields by t, i and b respectively.

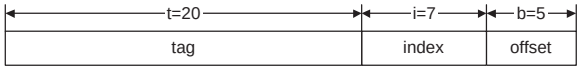

Figure 4: Memory Address decomposition on P6

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 *(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. 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 itself non-compact.

The micro-benchmarks discussed in this section access sequences of memory locations using a fixed-stride (always a power of 2), shown pictorially in Figure 5. These sequences can be characterized by three parameters

- starting address m_0,
- stride $S = 2^\sigma$, and
- number of elements in the sequence n.

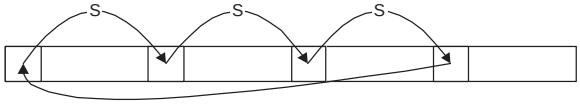

Figure 5: Fixed stride memory accesses ($n = 4$)

We will use $\langle m_0, S, n\rangle$ to refer to such a sequence and we will further require that $S = 2^\sigma$ and that the address m_0 is aligned on a cache block. 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 σ increases), the maximum length of a compact sequence with that stride gets smaller until it bottoms out at A.

THEOREM 1. *Consider a cache $\langle A, B, C\rangle$. A sequence $R = \langle m_0, S, n\rangle$ is compact iff $n \leq \max(\frac{C}{S}, A)$.*

Given the containment property of compact sets, it is easy to show that this theorem is equivalent to the following one.

THEOREM 2. *Consider a cache $\langle A, B, C\rangle$. For a given stride $S = 2^\sigma$, let $n_c = \max(\frac{C}{S}, A)$. The sequence $R_c = \langle m_0, S, n_c\rangle$ is compact while the sequence $R_{nc} = \langle m_0, S, n_c + 1\rangle$ is not.*

PROOF. Depending on the size of the stride $S = 2^\sigma$ there are three different cases:

- $i + b \leq \sigma$. This implies that $\frac{C}{A} \leq S$ and therefore successive addresses in the sequence $\{m_0 + i \times S\}_{i=0}^{n}$ have the same index and the same offset but different tags. Thus they all map to the same cache set. Because $n_c = \max(\frac{C}{S}, A) = A$, the sequence R_c has exactly A elements, so it is compact. On the other hand R_{nc} has $A + 1$ elements and thus it is non-compact.
- $b \leq \sigma < i + b$. This implies that $B \leq S < \frac{C}{A}$ and therefore successive addresses in the sequence $\{m_0 + i \times S\}_{i=0}^{n}$ will have the same offset but different indices and sometimes also different tags. $i + b - \sigma$ different indices will be accessed, which depends on the position of σ in the interval $[b, i + b)$. Each of the corresponding cache sets will be able to fit A addresses, so we are looking at compact sequences of length $n_c = 2^{i+b-\sigma} \times A = \frac{2^{i+b}}{2^\sigma} \times A = \frac{C}{A \times S} \times A = \frac{C}{S}$. Going to sequences of length $n_c + 1$ will cause $A + 1$ different addresses to be mapped to the same set, making the whole set non-compact.
- $\sigma < b$. In this case $S < B$ and therefore successive addresses in the sequence $\{m_0 + i \times S\}_{i=0}^{n}$ can have different offsets, indices or tags. In particular $\frac{B}{S} = 2^{b-\sigma}$ consecutive accesses will be in the same block. Furthermore all cache sets will be potentially used (2^i of them) and each one of them can hold A different addresses. In summary, we will be able to build a compact sequence of length $n_c = 2^{b-\sigma} \times 2^i \times A$, which we know from the previous case to be $n_c = \frac{C}{S}$. Again, if an extra element is added to the sequence, $A + 1$ elements will map to one cache set and thus the sequence will be non-compact.

□

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 [4]. Pseudocode for this approach is shown in Figure 6.

```
while (--R) // repeat many (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 execution time.

For these reasons, we use an approach that was first introduced by the Calibrator project [6]. 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 7 shows the pseudocode. 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 many (R) times
    p = *(void **)p;
```

Figure 7: Improved strided 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 micro-benchmark code generator, discussed in Section 2, with the configuration $C_M = \langle Q_M, \texttt{void *}, 1\rangle$ where Q_M is the operation $p_{11} = \texttt{*(void **)}p_{11}$ and p_{11} is the only register needed of type "`void *`". The execution time of this configuration will depend on how p_{11} and the corresponding array of pointers A are initialized. We will always initialize p_{11} to the first element of A and we will discuss different ways of initializing A in the following sections.

4.4 Measuring L1 Data Cache Parameters

Theorem 1 suggests a method for determining the capacity and associativity of the cache. We can 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 $\frac{C}{A}$; since 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. It relies on a measurement `compact`$(\langle m_0, S, n\rangle)$, which determines whether this sequence is compact by comparing the per element access time to that of $\langle m_0, 1, 1\rangle$. These times are measured as explained in Section 4.3.

```
S ← 1
n ← 1
while (compact(⟨m0, S, n⟩))
    n ← n × 2
n_old ← 0
while (true)
    n ← smallest n' ≤ n such that ¬compact(⟨m0, 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 size of the sequence n until the set is no longer compact. Then exponentially grow the stride S and for each stride, find the minimum sequence length n for which $\langle m_0, S, n\rangle$ 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 $C = \frac{A \times S}{2}$.

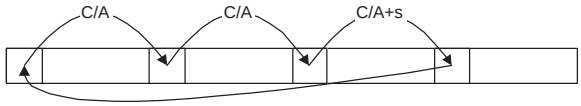

Figure 9: Array Configuration for Measuring B

To find the block size B of the cache, we perform the experiment shown in Figure 9. We can describe the sequence of memory addresses in this experiment by $\langle m_0, s\rangle$, where m_0 is the starting address of the array.

We access $A + 1$ different memory addresses, where the first A are separated by strides of $\frac{C}{A}$ and the last one is separated from them by a stride of $\frac{C}{A} + s$. Here s is a positive integer that is varied over a range of values. From Figure 4, we see that if $s < b$, the set of memory locations is not compact because all $A + 1$ locations map to the same cache set. On the other hand, if $s \geq b$ the set is compact because the last address maps to a different cache set. The algorithm for measuring the block size B of the cache is presented in Figure 10.

```
s ← 1
while (¬compact(⟨m0, s⟩))
    s ← s + 1
B ← s
```

Figure 10: Algorithm for Measuring Block Size

Finally, we can measure the cache *hit latency* L and the cache *miss latency* M. This is simply the time per access of the array configurations $\langle m_0, \frac{C}{A}, A\rangle$ and $\langle m_0, \frac{C}{A}, A + 1\rangle$ respectively. Note that to measure L we can also use $\langle m_0, 1, 1\rangle$, so knowing A and C is not a requirement for successful measurement of the hit latency.

4.5 Discussion

Although our approach works for most architectures, it does not work well on processors like the IBM Power3 that have a hardware prefetch unit that can detect fixed-stride accesses to memory and exploit them by prefetching the data. Notice that the strides in the innermost loop of the code of Figure 8 are constant, so the timing results will be affected by such hardware prefetching.

The solution is to access the addresses in a set in pseudo-random order while ensuring that all addresses are touched. In this setting, the stride is not constant, so the hardware prefetcher will not detect any patterns for prefetching. Suppose the memory address sequence is $\{m_i\}_{i=0}^{n-1}$. In this case, after accessing m_i, instead of going to m_{i+1} we access $m_{(i+p)\%n}$. We choose $p > n$ to be a prime number, and thus p and n are mutually prime so the recurrence $i = (i + p)\%n$ will generate all the integers between 0 and $n - 1$ before repeating itself. This computation is performed during array initialization so the code from Figure 8 can be used unchanged.

4.6 Higher Cache Levels Parameters

We will denote the cache at level i with $\mathcal{C}_i$ and the ABC of this level by $\mathcal{C}_i = \langle A_i, B_i, C_i \rangle$. The hit and miss latencies of $\mathcal{C}_i$ we will denote by L_i and M_i respectively. In fact $M_i = L_{i+1}$. The goal of this section is to explain the measurement of A_i, B_i, C_i, L_i, and M_i.

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 restriction prohibits us from using the algorithms in Figures 8 and 10 directly. There are cases when caches at lower levels have higher associativity, which in turn allows a set of memory addresses to be compact with respect to $\mathcal{C}_{i-1}$ while it is non-compact with respect to $\mathcal{C}_i$ (e.g. DEC Alpha 21264), which is a problem for the current algorithm.

The solution we provide relies on the assumption $C_{i-1} \leq S_i = \frac{C_i}{A_i}$. This restriction allows us to make small stride accesses targeted at $\mathcal{C}_{i-1}$ inside the big stride targeted at $\mathcal{C}_i$.Figure 11 is the equivalent of Figure 5 for higher cache levels. Instead of making monolithic strides of size S, we *decompose* each stride into A_{i-1} pieces. The first $A_{i-1} - 1$ of them are of size $S_{i-1} = \frac{C_{i-1}}{A_{i-1}}$ each, while the last one is sized appropriately to have a total step size of S.

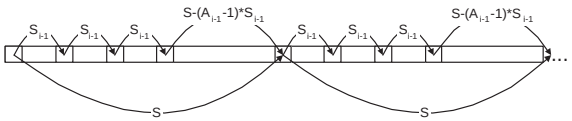

Figure 11: Access Pattern for Higher Level Caches

All the accesses in the first S_{i-1} small strides map to the same cache set with respect to $\mathcal{C}_{i-1}$. Because we make A_{i-1} accesses in each round, we are using all the slots in this set. Therefore when time comes to make the big stride S, the new small strides will all be misses with respect to $\mathcal{C}_{i-1}$.

We can recursively apply this decomposition algorithm for the S_{i-1} strides. Because each one of them is of size S_{i-1} and by our assumption $C_{i-2} \leq S_{i-1}$, we can decompose them into A_{i-2} pieces each, and so forth. This way we guarantee that every memory access at the current level will cause a cache miss on all lower levels. Finally to avoid problems with hardware stride access predictors, we can scramble the order we are accessing memory addresses at each level the same way as we did in Section 4.4. Applying all these fixes to $\langle m_0, S, n \rangle$ we get a new memory address sequence which we denote by $\langle m_0, S, n \rangle'$. Analogously we can define $\langle m_0, s \rangle'$ from $\langle m_0, s \rangle$ to compute the block size.

Now we can use the algorithms from Section 4.4 with the following slight modifications:

- `compact` should now empirically check for compactness based on $\langle m_0, S, n \rangle'$ and $\langle m_0, s \rangle'$ with respect to $L_i = M_{i-1}$ instead of $\langle m_0, S, n \rangle$ and $\langle m_0, s \rangle$ with respect to L_1.
- The initial value for S should be C_{i-1} instead of 1.
- M_i will be the timing per access of $\langle m_0, S_i, A_i + 1 \rangle'$.

The modified algorithm is presented in Figure 12. If we set $C_0 = 1$ and $M_0 = \texttt{Time}(\langle m_0, 1, 1 \rangle)$ we can also use it for

```
// measure A_i and C_i
S ← C_{i-1}
n ← 2
while (compact(⟨m_0, S, n⟩'))
    n ← n × 2
n_old ← 0
while (true)
    n ← arg min_{1≤n'≤n} ¬compact(⟨m_0, S, n'⟩')
    if (n = n_old)
        A_i ← n − 1
        C_i ← S/2 × A_i
        break;
    S ← S × 2
    n_old ← n

// measure B_i
s ← 1
while (¬compact(⟨m_0, s⟩'))
    s ← s + 1
B_i ← s

// measure L_i and M_i
L_i ← M_{i-1}
M_i ← Time(⟨m_0, S_i, A_i + 1⟩')
```

Figure 12: General Cache Parameter Measurement

measuring the parameters of $\mathcal{C}_1$. Therefore it can be used for measuring the parameters of all cache levels.

4.7 Measuring TLB Parameters

Hardware architectures employ *virtual memory*, which is a mechanism of using secondary storage to make applications believe that more memory is available than is physically present. The unit of granularity of virtual memory is a *page*.

User-level code is only exposed to virtual memory addresses, the general structure of which is presented in Figure 13 (the specific field widths are from Intel P6). The address is divided into two main portions. The lo-order bits contain a page offset (which is not subject to translation), while the hi-order bits are used for indexing page tables during the translation process.

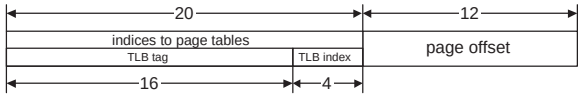

Figure 13: Memory Address decomposition on P6

Because the translation from virtual to physical addresses is an expensive procedure to perform on each memory access, a Translation Lookaside Buffer (TLB) is used to cache and reuse the results.

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 many 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.

Measuring TLB parameters is generally a harder problem because they often support variable page size and rarely use LRU replacement policy (they often do round-robin or even random replacement). Also, TLBs are often fully-associative. In this paper we will stick to the assumptions of a single page size and LRU replacement policy. As we will see in Section 5 this will often allow us to measure TLB parameters successfully.

As we saw with measuring higher level cache parameters, it is important to make sure that the TLB is in fact accessed and also to make everything else work at full speed so that the effects of TLB can be isolated and measured. In our context that means that no cache misses should generally happen at any level. Also in order to make sure that TLB is in fact always accessed we need to impose the restriction that the L1 data cache should be physically tagged.

In order to avoid cache misses we need to tweak our array initialization in a specific way. At this point we will assume that have the parameters for $\mathcal{C}_1 = \langle A_1, B_1, C_1 \rangle$.

TLBs are usually highly associative and pages are relatively big (compared to cache lines), which results in a large number of relatively big steps in our algorithms. In such circumstances it is easy for $\mathcal{C}_1$ to miss. Let us assume for now that the current stride S we want to use is larger than $S_1 = \frac{C_1}{A_1}$. In this case we can do $A_1 - 1$ strides of size S which will make full use of that single $\mathcal{C}_1$ set, then make one slightly larger stride of size $S + B_1$, which will force the use of a different set in $\mathcal{C}_1$. We can repeat this pattern for generating a compact set of memory addresses with respect to $\mathcal{C}_1$. The pictorial view is presented in Figure 14.

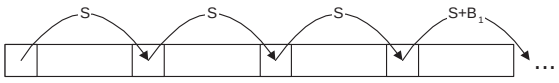

Figure 14: Access Pattern for TLB Parameter Measurement

One restriction of this approach is that with each stride we are accessing a memory address that maps into a separate slot in $\mathcal{C}_1$, and therefore the total number of accesses should be less than the total number of slots in $\mathcal{C}_1$, i.e. $\frac{C_1}{B_1}$. 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.

Using this specialized array configuration we can just use the general algorithm for Section 4.6 without modification.

The algorithm we have presented here has a number of restrictions, which we plan to address in our future work. The possible enhancements are discussed in Section 6.

5. EXPERIMENTAL RESULTS

In this section, we will present and discuss the hardware characteristics that were deduced by X-Ray when it was run on a number of different platforms. In particular, we will show results for the following,

- Intel P6 (Pentium Pro, II, III), using 2xPIII@1GHz and 4xPIII@500MHz
- Intel P7 (Pentium 4), using 2xP4@2.4GHz
- AMD Athlon MP, using 2xAMP@1.8GHz
- Sun UltraSPARC III, using 2xUSIII@900MHz
- IBM Power 3, using 2xP3@375MHz

We established the following protocol for these experiments,

- We boosted the priority of the process running X-Ray and the other tools in order to increase the accuracy of the results.
- All the results presented in the following sections are from a single execution of the corresponding micro-benchmarks. Results vary slightly between different executions.

In order to establish the benefits of using our algorithms, we compared X-Ray against three other similar systems: ATLAS, Calibrator [6] and MOB [1].

The ATLAS framework contains a small set of micro-benchmarks to measure L1 data cache size, number of registers, multiply latency, and existence of fused multiply add. However, it is not a goal of ATLAS to report very accurate values. This is because ATLAS does not use the hardware parameters to directly compute optimization parameters for loop tiling, loop unrolling, etc. Rather, it uses search to find the best optimization parameters and is, thus, able to tolerate inaccuracies hardware parameters.

Neither Calibrator or MOB was designed to measure CPU characteristics. We will compared our results to the results of these two tools for measuring memory hierarchy parameters.

5.1 CPU Frequency

Table 1 presents the results from running the X-Ray CPU Frequency micro-benchmark on the target architectures. The results are in cycles per second.

Architecture	Result (MHz)	Error (%)
PIII@1GHz	981.482	-1.85%
PIII@500MHz	495.025	-1.00%
P4@2.4GHz	4740.582	-1.24% (+97.52%)
AMP@1.8GHz	1731.705	-3.79%
USIII@900MHz	898.896	-0.12%
P3@375MHz	374.860	-0.04%

Table 1: CPU frequency reported by X-Ray

The results are very accurate, loosing on average 1.34% of the advertised frequency. The error is generally larger when the frequency is higher.

The only real surprise in this experiment was the P4, in which the frequency measured by X-Ray was twice the actual CPU frequency. The reason for this is simple: we assume that integer adds have an initiation interval of 1 in order to compute frequency, but the P4 has a double-pumped integer ALU unit. In effect the integer add instruction has an initiation interval of 0.500 on the P4.

5.2 Instruction Latency and Initiation Interval

Table 2 presents the results from running the X-Ray latency (L) and initiation interval (II) benchmarks on the floating point (FP) Add and FP multiply instructions using double precision arithmetic. The results on in cycles.

Architecture	Add		Multiply	
	L	II	L	II
PIII@1GHz	3.015	1.015	4.987	2.000
PIII@500MHz	2.985	1.000	5.046	2.000
P4@2.4GHz	9.934	2.018	13.934	3.967
AMP@1.8GHz	4.000	1.000	4.000	1.000
USIII@900MHz	4.002	1.000	4.003	1.006
P3@375MHz	4.000	0.500	4.000	0.500

Table 2: FP Add and FP Multiply Parameters

These results are very accurate (only 0.08% off on average, with the maximum difference of 1.5%). There are two aspects of these results that deserve explanation:

- On P4, the latencies are twice higher than the official CPU specification. This is because they are relative to the integer ALU frequency, which is twice faster than the rest of the core.
- On Power3 the initiation interval of both instructions is 0.5, which means that the CPU is dispatching 2 instruction per cycle. This is because the Power3 has two ALUs. We will discuss this further in Section 5.2.1.

5.2.1 Functional Units

Our results show that the Power3 can dispatch 2 independent FP operations per cycle. This is also true for integer operations and is easy to verify using our framework. Our results also show that the dispatch 2 dependent integer operations per cycle (otherwise its frequency would have been twice as large). Therefore, we can conclude that there must be 2 independent integer ALUs.

Unfortunately this reasoning is not always that straightforward. As discussed in Section 3.4, there are often bottlenecks in CPUs that prevent us from obtaining a completely accurate description of the functional units. Consider the PIII, which also has 2 integer ALUs. This is measured by experiments (not shown here) that report the PIII's initiation interval for integer operations as also 0.500. However, further experiments reveal a bottleneck.

Recall that the initiation interval tests measure the performance of a sequence of instructions, p_{11}+=p_{11}; p_{12}+=p_{12}; If we perform the test with more registers, e.g. using p_{11}+=p_{13}; p_{12}+=p_{13};, initiation interval on the PIII increases to 0.667. Furthermore if we introduce yet another register, p_{11}+=p_{13}; p_{12}+=p_{14};, initiation interval rises further to 1.000. The explanation of this phenomenon is that PIII has only 2 register read ports and, thus, can read no more than 2 registers per cycle.

A different effect shows appears in the P4. The P4 also has 2 integer ALUs and more read ports. All three versions of the initiation interval test, given above, report an initiation interval of 0.667. We believe that this can be explained by the following: because the ALUs operate on twice higher frequency and because of the two instructions in the initiation interval tests are independent, it should be possible to dispatch 4 integer instructions per core cycle. However, the instruction cache on P4 can only deliver 3 instructions to the core each cycle, so one slot per core cycle in not utilized. Thus, the reported initiation interval is 1.5 cycles.

None of these reasons for decreased performance were due to the nature of the functional units themselves. It is likely that, as CPU manufacturers add more functional units to their processors and as long as these bottlenecks remain, the utilization will drop further. Therefore we made the decision to try to find the different sets of independent instructions that can be sustained per cycle, subject to all bottlenecks in the CPU. We can then use this sets with a suitable pattern-matching algorithm to perform scheduling.

5.3 Number of Registers

The results from finding the number of integer and double-precision floating-point registers on our target architectures are presented in Table 3.

Architecture	Integer	FP
PIII@1GHz	5	8
PIII@500MHz	5	8
P4@2.4GHz	5	8
AMP@1.8GHz	5	8
USIII@900MHz	25	32
P3@375MHz	25	32

Table 3: Number of registers reported by X-Ray

Although the number of integer registers found is lower than the number reported by the manufacturers, recall that X-Ray determines the number of registers that the compiler is able to use for program variables. The rest are special registers to handle stack and frame pointers, return addresses, etc.

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.

Finally, one point that deserves mentioning is that on the UltraSPARC architecture, the default compiler settings caused the compiler to generate code for an older version of the ISA, so X-Ray found only 16 FP registers. After supplying the appropriate switch to enable the compiler to generate code for the USIII, X-Ray successfully found all 32 FP registers.

5.3.1 Other CPU features

X-Ray was able to determine the presence or absence of a FMA instruction on the target platforms. It correctly reported that the Power3 was the only platform with such an instruction.

X-Ray was also able to correctly determine the number of multiple physical (SMP) and virtual (SMT) processors.

5.4 Memory Hierarchy

The results from measuring the L1 data cache parameters on the target machines are presented in Table 4.

We successfully found the correct values L1 cache parameters on all the architectures. A small exception was Power3, where we classified the cache as 129-way set associative, while it is only 128-way set associative. What is interesting

Architecture	Associativity	Block Size	Capacity
PIII@1GHz	4	32	16K
PIII@500MHz	4	32	16K
P4@2.4GHz	4	64	8K
AMP@1.8GHz	2	64	64K
USIII@900MHz	1	32	64K
P3@375MHz	129	128	64.5K

Table 4: L1 Data Cache Parameters Reported by X-Ray

about this case is that there was no performance loss moving from 128 to 129 steps, but there was one moving from 129 to 130. So there is some way in which Power3 manages to fit such the 129 step memory access pattern into its L1 data cache. This phenomenon also resulted in a cache capacity of 64.5KB. Furthermore none of the three tools against which we compared X-Ray were able to measure any parameters of the Power3's L1 data cache. We believe that the reason for these results is the hardware stride predictor.

Our algorithm also successfully measured the parameters of the L2 cache and of the data TLB on the Intel systems (PIII and P4). X-Ray does not correctly report the L2 cache on the other platforms, because this test is vulnerable to the effects of other processes on the machine. We are currently working to improve the stability of the algorithm on multi-user machines. We will present more complete results in the final paper.

Calibrator was able to accurately find the capacity and block size of the L1 data cache parameters on all the x86 targets (PIII, P4 and Athlon) and the UltraSPARC, while it did not come up with a result for Power3 at all. Further more Calibrator measured the L2 cache parameters on PIII and P4, while it failed to find anything on UltraSPARC and Power3. On Athlon it produced a wrong result (320KB L2 cache capacity).

We were able to run MOB only on the Athlon and the Power3 target platforms. On the rest the executable would raise a signal and abort during instruction cache tests. It did not find any cache parameters on Power3 and found a wrong L1 cache size (256KB), reporting it as a unified instruction and data cache.

We are contacting the authors of Calibrator and MOB to verify that we used their tools appropriately. Any improved results will be included in the final paper.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented X-Ray, a tool that automatically measures architectural parameters using a set of micro-benchmarks. The implementation of X-Ray uses a novel framework for generating these micro-benchmarks. We designed and implemented algorithms for measuring CPU frequency, instruction latency, initiation interval and existence, testing of the CPU functional units, finding parallelism in the form of SMP and SMT, and analysing the memory hierarchy. X-Ray is written in C and produces micro-benchmark code in C, so it is very portable.

The algorithms for measuring CPU parameters are new and provide a solid foundation for high-performance code-generation. The basic approach for measuring parameters of the memory hierarchy has been known for a long time, but the algorithms presented here are a big improvement in terms of accuracy and flexibility. They are also the first set of micro-benchmarks that by design are able to accurately find cache parameters for caches whose capacity is not a power of 2.

We are investigating the following topics.

- How do we measure other cache parameters such as write mode, replacement policy, unified code and data organization, virtual/physical indexing, etc.?
- Can we do a better job of measuring higher cache level and TLB parameters?
- How do we measure instruction cache parameters?
- How well does X-Ray do in measuring parameters of challenging architectures like Intel Itanium and AMD Opteron?

The implementation of the framework is freely available and will be disclosed in the final version for confidentiality reasons.

7. REFERENCES

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

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

[3] 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.

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

[5] Jeremy Johnson, Robert W. Johnson, David A. Padua, and Jianxin Xiong. Searching for the best FFT formulas with the SPL compiler. *Lecture Notes in Computer Science*, 2017:112–??, 2001.

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

[7] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. `http://www.netlib.org/lapack/lawns/lawn147.ps`.

[8] 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.