Profile-Guided Post-Link Stride Prefetching *

Chi-Keung Luk, Robert Muth, Harish Patil, Richard Weiss†,
P. Geoffrey Lowney, Robert Cohn

Massachusetts Microprocessor Design Center
Intel Corporation
334 South Street, Shrewsbury, MA 01545

† Department of Computer Science
Smith College
Northampton, MA 01063

{chi-keung.luk, robert.muth, harish.patil, geoff.lowney, robert.cohn}@intel.com, rweiss@cs.smith.edu

ABSTRACT

Data prefetching is an effective approach to addressing the memory latency problem. While a few processors have implemented hardware-based data prefetching, the majority of modern processors support data-prefetch instructions and rely on compilers to automatically insert prefetches. However, most prefetching schemes in commercial compilers suffer from two limitations: (1) the source code must be available before prefetching can be applied, and (2) these prefetching schemes target only loops with statically-known strided accesses. In this study, we broaden the scope of software-controlled prefetching by addressing the above two limitations. We use profiling to discover strided accesses that frequently occur during program execution but are not determinable by the compiler. We then use the strides discovered to insert prefetches into the executable directly, without the need for re-compilation. Performance evaluation was done on an Alpha 21264-based system with a 64KB data cache and an 8MB secondary cache. We find that even with such large caches, our technique offers speedups ranging from 3% to 56% in 11 out of the 26 SPEC2000 benchmarks. Our technique has been incorporated into `Pixie` and `Spike`, two products in Compaq's Tru64 Unix.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Optimization*; C.1.1 [**Processor Architecture**]: Single Data Stream Architectures

General Terms

Languages, Performance

Keywords

Memory latency, data prefetching, address strides, profiling, post-link optimizations

*This work was performed while the authors were with Compaq Computer Corporation.

ICS'02, June 22-26, 2002, New York, New York, USA.

1. INTRODUCTION

Memory latency is one of the major obstacles that limit the performance of modern microprocessors. In many recent machines, it is quite common to take 100 cycles or more to fetch a word from memory, during which hundreds of instructions can potentially be executed on these multiple-issue machines. Even worse, the gap between processor and memory speeds is expected to grow in future processor generations.

Prefetching is a promising technique to cope with the memory latency problem, and can be controlled by hardware or software. Hardware-based schemes [9, 20, 28] typically look for patterns of previous accesses to predict future behavior at run-time, while software-based schemes [7, 25, 26] rely on either the programmer or the compiler to insert explicit prefetch instructions. Since software-controlled prefetching requires minimal hardware support, many microprocessors today already support prefetch instructions.

Most commercial machines rely on compilers to automatically insert prefetches by analyzing the *source code*. These compilers (e.g., the IBM [5] and HP [30] compilers) typically prefetch memory references in loops whose addresses have *static* strides. While this approach works reasonably well, it has two limitations. First, prefetching cannot be applied when the source code is not available for compilation. This is particularly a problem in optimizing commercial software, libraries, or legacy codes. Second, the scope of prefetching is largely constrained by the fact that the stride size must be known at compile time. In applications with features like dynamic memory allocation, pointers, and indirect references, there are still a non-trivial number of cache misses caused by references with strides that are not captured by the compiler.

In this study, we address the above two limitations by a technique called *profile-guided post-link stride prefetching*, which serves as a complement to compiler-inserted prefetching. It is based on an observation that although many strides are not compile-time constants, they are in fact highly predictable given the profile information collected from past runs. The strides predicted via profiling are then used to insert prefetches into the *executable* directly, without the need for re-compilation.

Our technique has been incorporated into two existing *post-link* profiling/optimization tools for the Alpha. More specifically, we extend `Pixie` [34], which originally instruments executables for control-flow profiling, to collect stride-related information. We also extend `Spike` [10, 12]—an optimizing tool for Alpha executables—to insert prefetches ac-

(a) Code

```
arc* arcin;
node* tail;
...
while (arcin) {
   tail = arcin→tail;
   ...
   arcin = (arc_t*)tail→mark;
}
```

(b) Memory Accesses

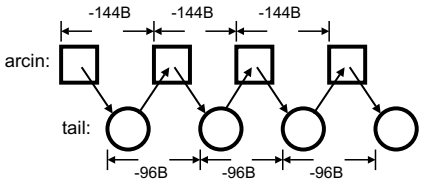


Figure 1: Abstract version of an important loop nest in the Spec2000 benchmark `mcf`.

cording to the stride profile collected. Performance results on an Alpha 21264 system (a DS20E) demonstrate that our technique speeds up SPEC2000 benchmarks that are already aggressively optimized by as much as 56%.

1.1 An Overview

The rest of this paper is organized as follows. We begin in Section 2 by performing two case studies. In Section 3, we describe how our technique is implemented using `Pixie` and `Spike`. Section 4 presents an experimental evaluation of our technique on an Alpha 21264 system using the SPEC2000 benchmarks. Section 5 discusses related work, and finally, we conclude in Section 6.

2. CASE STUDIES

In this section, we discuss the strided references exhibited in two SPEC2000 benchmarks: `mcf` and `equake`. They are selected because they are the two benchmarks in their corresponding groups (i.e. integer and floating point) that benefit the most from stride prefetching. In addition, their access patterns are relatively easy to understand without the need of looking at the entire program.

2.1 Mcf

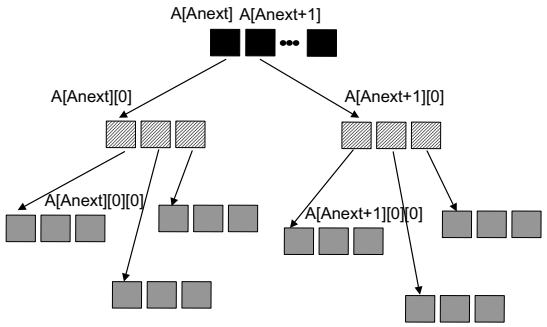
`Mcf` is an application used for single-depot vehicle scheduling. It is the integer benchmark in SPEC2000 that suffers the most from data cache misses. The DCPI [3] tool reports that 26% of the total stall time in `mcf` (running on a DS20E system which will be described in detail in Section 4.1) happens in the while-loop shown in Figure 1(a). It traverses a list structure via two pointers, `arcin` and `tail`, as shown in Figure 1(b). Profiling the values of these two pointers reveals that `arcin` and `tail` have strides of -144 bytes and -96 bytes, respectively, throughout the entire execution.[1] These strides originate from the fact that both `arcs` and `nodes` are allocated in memory via `calloc()`, and that the list structure is not changed once it is created. With these strides,

[1]The magnitude of these two strides depends on both the compiler and run-time system.

(a) Code

```
for (i = 0; i < nodes; i++) {
   Anext = Aindex[i];
   Alast = Aindex[i + 1];
   ...
   Anext++;
   while (Anext < Alast) {
      ...
      sum0 +=  A[Anext][0][0]*v[col][0] +
               A[Anext][0][1]*v[col][1] +
               A[Anext][0][2]*v[col][2];
      sum1 +=  A[Anext][1][0]*v[col][0] +
               A[Anext][1][1]*v[col][1] +
               A[Anext][1][2]*v[col][2];
      sum2 +=  A[Anext][2][0]*v[col][0] +
               A[Anext][2][1]*v[col][1] +
               A[Anext][2][2]*v[col][2];
      ...
      Anext++;
   }
...
}
```

(b) Compiler's View



(c) Layout in Memory

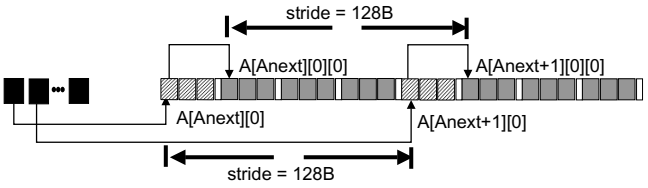


Figure 2: Abstract version of an important loop nest in the Spec2000 benchmark `equake`.

we can easily determine any future pointer values down the list from the current ones.

2.2 Equake

`Equake` is a program for simulating the propagation of elastic waves in large basins. The core computation lies in a sparse matrix multiplication routine, which contains the nested loop structure shown in Figure 2(a). Our focus is on the three dimensional matrix `A[ARCHmatrixlen][3][3]`, whose cache misses account for 70% of the total stall time. Since this matrix is dynamically allocated, the compiler treats it as an array of `ARCHmatrixlen` pointers, each pointing to an array of three pointers, each in turn pointing to an array of three doubles, as pictured in Figure 2(b). Nevertheless, since all these arrays are allocated consecutively without

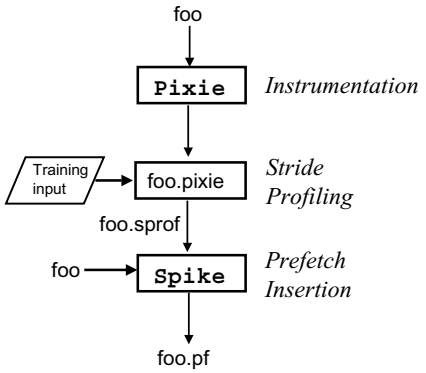


Figure 3: The three major steps of profile-guided post-link stride prefetching

intervention of allocation for other objects, the memory layout in fact looks like the one shown in Figure 2(c). As a result, there is a constant stride of 128 bytes between `&A[Anext][j]` and `&A[Anext+1][j]` (where `j` = 0 to 2), and also an 128-byte stride between `&A[Anext][j][k]` and `&A[Anext+1][j][k]` (where `j` = 0 to 2, `k` = 0 to 2).[2]

3. PROFILE-GUIDED POST-LINK STRIDE PREFETCHING

Before discussing the details of our technique, we first give an overview of it. Figure 3 shows the three major steps involved, including both the functionalities and software tools used. They are explained below:

Instrumentation: The first step is to *instrument* the given executable (i.e. `foo` in Figure 3) for profiling the strides at the loads that are likely to cause cache misses. We enhance `Pixie` to perform this new type of instrumentation in addition to the existing control-flow instrumentation.

Stride Profiling: The next step is *stride profiling*, where we run the instrumented executable (`foo.pixie`) with a training input set. It generates a profile (`foo.sprof`) of the strides found at the loads selected in the first step.

Prefetch Insertion: The final step takes the stride profile and inserts stride prefetches into the executable (`foo.pf`) accordingly. A prefetch-insertion module has been added to `Spike` including various optimizations that maximize the benefit of stride prefetching.

Having seen the overview, we now discuss the above three steps in a greater depth.

3.1 Instrumentation

The first step is to decide which loads in the executable need to be instrumented for stride profiling. Of course, the simplest way is to instrument *all* loads. However, doing this is expensive in terms of the overhead of stride profiling, and is also unnecessary since previous research shows that cache

[2]Again, the magnitude of these two strides depends on the compiler and run-time system.

(a) Naive Instrumentation

```
R1 ← R1 + 16;
R2 ← R2 + 8;
// ...
// both R1 and R2 are not redefined
// ...
   StrideProfile(R1+24); //instrumentation
R3 ← load 24(R1);
   StrideProfile(R2+64); //instrumentation
R4 ← load 64(R2);
   StrideProfile(R1+48); //instrumentation
R5 ← load 48(R1);
   StrideProfile(R2+96); //instrumentation
R6 ← load 96(R2);
```

(b) Optimized Instrumentation

```
R1 ← R1 + 16;
R2 ← R2 + 8;
   StrideProfile(R1, R2); //instrumentation
// ...
// both R1 and R2 are not redefined
// ...
R3 ← load 24(R1);
R4 ← load 64(R2);
R5 ← load 48(R1);
R6 ← load 96(R2);
```

Figure 4: Examples of a *naive* instrumentation and an *optimized* instrumentation for stride profiling

misses tend to be generated by only a small subset of static loads in the program [1, 27]. Thus, we use the following two heuristics to decide whether a load should be instrumented:

Not-scalar: Since references of scalars or small aggregates rarely miss in the cache, they can be ignored for prefetching. On Alpha, these references are typically made off the global pointer (`$gp`) or the stack pointer (`$sp`).

Not-compiler-prefetched: If a load is already prefetched by the compiler, it does not need to be considered for stride prefetching. Thus, such type of loads are also excluded for instrumentation.

While the above heuristics always accelerate stride profiling (as fewer loads are instrumented), their performance impact can be mixed, depending on the miss rates of the loads that are ignored for stride profiling. Later in Section *4.2.3*, we will measure their performance impact.

Once we decide which loads are important, we are ready to perform the actual instrumentation. A naive approach is to do an one-to-one instrumentation, like the pseudo assembly code shown in Figure 4(a), where one instrumentation call is inserted for each load. Nevertheless, the example also illustrates the following two opportunities for optimization.

First, it is not necessary to profile the effective address of the load: Profiling the value of the base register is already sufficient to detect the address strides (since the offset will remain the same). An implication of this is that loads that share the same base register can also share the instrumentation call. And a natural place to put the instrumentation call is immediately after the point where the base register is defined. For this purpose, we enhance `Pixie` to find the definition points of registers in a flow graph, using an al-

gorithm [31] resembling static single assignments (SSA) [2, 13].

Second, multiple instrumentation calls located at the same place can be combined into a single instrumentation call. In our example, applying these two optimizations together reduces the number of instrumentation calls from four to one, as shown in Figure 4(b). In general, as we will show in Section *4.2.2*, these optimizations are very effective at reducing both the number of instrumentation calls and the profiling time.

3.2 Stride Profiling

In this step, we run the instrumented program with a training input set in order to profile the address strides of the loads selected. Similar to the way that strides are detected in Chen and Baer's hardware prefetching scheme [9], a stride S is recognized if it appears twice in a row (i.e. $Address_i - Address_{i-1} = S = Address_{i+1} - Address_i$). When there are more than one stride, we record up to 10 of them per load, which we found to be sufficient in most cases. Each stride is associated with three pieces of information: its value (including sign), its frequency, and the average run-length of that stride. The *run-length* is defined as the number of consecutive instances of a given load that share the same stride. As we will discuss in Section 3.3, the run-length is useful for determining how far ahead a load should be prefetched.

As a tradeoff between accuracy and speed, stride profiling can be either *complete* or *sampled*. Complete profiling collects statistics from the entire execution without any skipping. In contrast, sampled profiling collects statistics from parts of the execution, with other parts skipped. We have examined two approaches to sampled profiling. The first is a *periodic* one, where statistics collection is turned on and off repeatedly during the entire execution. The second approach collects statistics for only the first N instances of each load being profiled, where N is reasonably large yet is still small compared to the total execution count. While the first approach works better for programs with different phases in the execution that lead to different strides, we found that the software overhead of turning on and off statistics collection is so substantial that there is no significant advantage to using periodic sampling. Thus, we focus on evaluating the second approach when we compare sampled profiling against complete profiling later in Section *4.2.4*.

3.3 Prefetch Insertion

In the final step, `Spike` reads the stride profile and inserts prefetches into the executable. There are three subtasks that `Spike` performs in this step: (i) determining the stride to be used for prefetching if multiple strides are detected for a particular load, (ii) computing the prefetching distance for each strided load, and (iii) minimizing the number of prefetches needed to be inserted. They are further discussed in the rest of this section.

3.3.1 Choosing from Multiple Strides

When multiple strides are detected for a given load, `Spike` has to decide which one should be used. Figure 5 shows for the SPEC2000 benchmarks the distribution of the most frequent stride vs. the less frequent ones for each static load that exhibits strided accesses. On average, the most frequent stride happens over 90% of time for integer and around 70% of time for floating point. To cope with multiple strides, there are two possible approaches, as shown in Figure 6. Figure 6(b) is the approach where the most frequent stride is selected from the profile. The prefetch offset k can then be *statically* computed. In contrast, Figure 6(c) shows the approach where the stride is computed at every iteration. This requires one free register (i.e. t in Figure 6(c)) and four instructions in addition to the prefetch itself. Since finding free registers at post-link time is challenging in terms of ensuring correctness, and we do not want to introduce spilling code to free up registers, we adopt the static approach in this study. Nevertheless, according to Figure 5, this would still allow us to cover a majority of strided accesses.

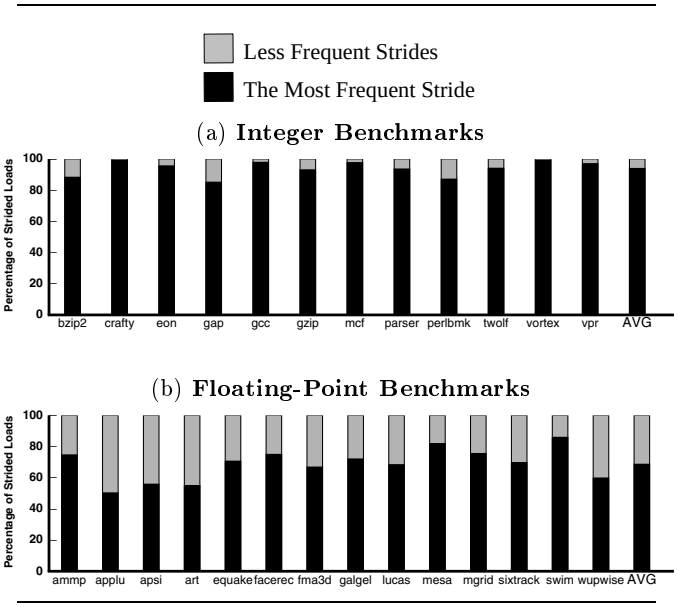


Figure 5: Distribution of the most frequent stride and less frequent ones.

3.3.2 Computing the Prefetching Distance

Once the stride is determined, the next step is to compute the *prefetching distance*—the number of iterations to prefetch ahead. A general formula for computing prefetching distance [26] is: $D = \lceil \frac{l}{w} \rceil$, where l is the expected miss latency and w is the estimated amount of computation between two consecutive references in cycles. For a strided load L, w can be approximated by the average body length of the innermost loop that encloses L. We have enhanced `Spike` to compute the average loop body length based on the control-flow feedback collected via regular `Pixie` profiling. For the miss latency l, we conservatively assume it to be a L2-cache miss, thereby ensuring sufficient time to bring in the data. Since w is now in terms of the average number of instructions executed in a loop, we also multiply l by the instruction-per-cycle (IPC). In our experiments, l and the IPC were assumed to be 100 cycles and 1.4, respectively.

Another consideration in deciding the prefetching distance is the *run-length* R of a stride. If R is significantly greater than the ideal prefetching distance $D = \lceil \frac{l}{w} \rceil$, the actual prefetching distance is simply D. However, if R is comparable to D, something less than D should be used instead. The rationale is that when we prefetch N references ahead,

(a) Original

```
R2 ← load 16(R1);
```

(b) Statically Chosen Stride

```
// k is a constant equal to
// 16 + stride * prefetching distance
prefetch k(R1);
R2 ← load 16(R1);
```

(c) Run-time Stride Detection

```
t ← R1 - t; // t has been holding
            // the previous value of R1.
            // So, (R1 - t) is the stride.
t ← prefetching distance * t;
t ← R1 + t;
prefetch 16(t);
t ← R1; // save R1 for computing
        // the next stride
R2 ← load 16(R1);
```

Figure 6: Two approaches to handling multiple strides

the first N out of the R references in the run will not be prefetched. Thus, the prefetching distance should not be so large that many references at the beginning of the run are not covered. In the case where $R \leq D$, the actual prefetching distance is set to $\frac{R}{2}$, the mid-point of the run, in an effort to balance between prefetch coverage and prefetch timeliness.

3.3.3 Prefetch Minimization

The final step is to optimize away prefetches that are redundant because there are already prefetches to the same cache lines. This type of redundancy typically occurs when nearby fields in a structure or consecutive array elements are separately prefetched. Our prefetch minimization algorithm consists of four steps, which are explained below using the example shown in Figure 7:

1. Prefetches that share the same base register are potentially redundant if their offsets are close enough. Hence, we first logically group prefetches at the point where their base registers are defined. For instance, all the six prefetches shown in Figure 7(a) are considered together for minimization at the definition point of `R1` (i.e. `R1 ← R1 + 8`). If the base register has multiple possible definitions, the confluence point of these definitions will be used instead (the confluence points are essentially where the ϕ functions in SSA are located).

2. Prefetches grouped at the same register definition point are then classified into two types: "must" or "may" prefetches, according to the likelihood that the prefetch will be executed between that definition point and the end of the program. "Must" prefetches are those that are certain to be executed, while "may" prefetches are those that are uncertain. For instance, the "must" prefetches with respect to the definition point of `R1` are `prefetch 16(R1)`, `prefetch 64(R1)`, `prefetch 72(R1)`, and `prefetch 118(R1)`. The remaining two prefetches are "may" prefetches. Both "must" and "may" prefetches are computed via a backward data-flow analysis.

3. We then compute the maximum possible number of cache lines spanned by "must" prefetches. With this information, we can find the minimum set of prefetches that span the same cache lines. For example, the four "must" prefetches in Figure 7(a) target addresses between `16(R1)` and `118(R1)`, spanning at most three 64-byte cache lines. These four prefetches can hence be reduced to three: `prefetch 16(R1)`, `prefetch 80(R1)`, and `prefetch 118(R1)`, which cover the same set of lines. Note that "may" prefetches are not considered in this step because we do not want to introduce additional overhead by executing prefetches that are actually not needed in the original program.

4. Finally, any "may" prefetches whose data addresses are already covered by the span of "must" prefetches can be removed. Thus, in the example, `prefetch 32(R1)` is eliminated but `prefetch 1024(R1)` is not. The ultimately optimized code is shown in Figure 7(b), where either one or two prefetches are saved in execution, depending on the direction of the if-then-else statement.

(a) Before Prefetch Minimization

```
R1 ← R1 + 8;
prefetch 16(R1);
prefetch 64(R1);
if (...)  then
   prefetch 32(R1);
   ...
else
   prefetch 1024(R1);
   ...
endif
prefetch 72(R1);
prefetch 118(R1);
...
```

(b) After Prefetch Minimization

```
R1 ← R1 + 8;
prefetch 16(R1); // Beginning of a 3-line span
prefetch 80(R1); // One line from the beginning
if (...)  then
   // prefetch 32(R1) is combined into the span
   ...
else
   prefetch 1024(R1);
   ...
endif
// prefetch 72(R1) is combined into the span
prefetch 118(R1); // End of the span
...
```

Figure 7: An example of prefetch minimization, with a cache line size of 64 bytes assumed.

4. EXPERIMENTAL EVALUATION

We evaluated profile-guided stride prefetching on an Alpha 21264-based system. The experimental framework is depicted in Section 4.1, and the performance results are discussed in Section 4.2.

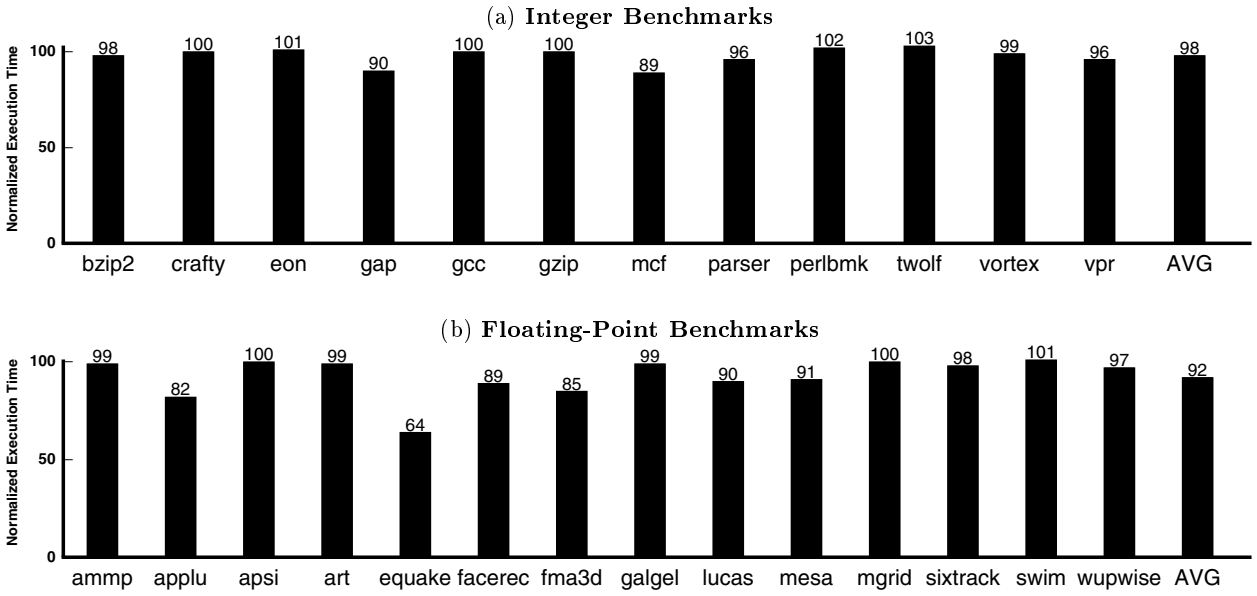


Figure 8: Performance of profile-guided stride prefetching on SPEC2000. Execution time is normalized to an aggressively optimized baseline.

Table 1: The memory hierarchy of our DS20E machine.

Line Size	64 bytes
I-Cache	64KB, 2-way set-associative
D-Cache	64KB, 2-way set-associative
Memory Parallelism	32 in-flight loads, 32 in-flight stores, 8 in-flight cache block fills, 8 cache victims
Off-Chip Unified L2-Cache	8MB, direct-mapped
L1-to-L2 Miss Latency	12 cycles (plus any delays due to contention)
L1-to-Memory Miss Latency	80 cycles (plus any delays due to contention)
L1-to-L2 Bandwidth	6.9 GB/sec
L2-to-Memory Bandwidth	2.6 GB/sec

4.1 Framework

The test bed of our experiments was a DS20E Alpha workstation [11], equipped with a 667-MHz 21264 processor [23] and 2GB of main memory. The Alpha 21264 is an out-of-order superscalar machine which can execute up to four instructions per cycle. The memory hierarchy is the most relevant component to our study, and hence is summarized in Table 1.

We used the entire SPEC2000 suite [17] as our benchmarks. The *training* data sets were used to generate the stride profile, while the *reference* data sets were used for performance measurement. For most benchmarks, the median execution time of five runs was reported. However, `galgel` and `sixtrack` required more runs due to the relatively large variances observed in their execution times.

The benchmarks were first compiled using the standard Compaq compiler version 6.3 with -O5 optimizations under Tru64 Unix V5.1. At this optimization level, the compiler inserts prefetches which are estimated as beneficial. The benchmarks were then further optimized by `Spike` to improve their I-cache performance. Thus, the baseline of our experiments was the best that we could get prior to stride prefetching.

4.2 Results

We first report the overall performance improvement due to profile-guided stride prefetching, followed by the overhead of stride profiling. Next, we measure the performance impact of the two heuristics used in instrumentation, as well as that of prefetch minimization. Finally, we demonstrate the effectiveness of sampled stride profiling.

4.2.1 Performance of Stride Prefetching

Our first set of results are shown in Figure 8, where the execution time of stride prefetching is normalized to that of the baseline. The first observation made from Figure 8 is that stride prefetching is more effective on the floating-point side than on the integer side. This is somewhat expected as floating-point codes typically contain more regular data accesses and are more loop intensive. Nevertheless, we still see over 4% speedups in four out of the 12 integer benchmarks. The performance gains on the floating-point side are quite substantial: Six programs are sped up by at least 10%, with `equake` up by 56%. Although we do suffer slowdowns in a few benchmarks, they are all quite mild (at most 3% in the case of `twolf`). On average, stride prefetching speeds up the integer benchmarks by 2% and the floating-point benchmarks by 9%.

To understand the performance results in a greater depth, we used Atom-based cache simulation [35] to estimate the data traffic between the D-cache and the L2 cache. The

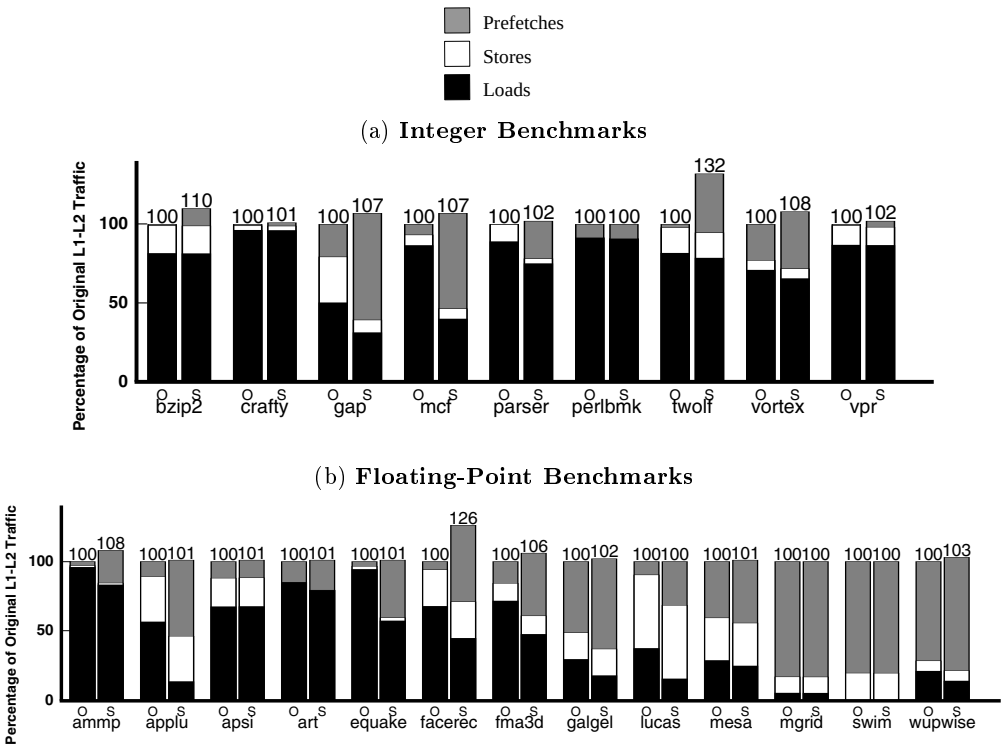


Figure 9: Data traffic between the D-cache and the L2 cache (O = original, S = with stride prefetching).

results are shown in Figure 9, where the two bars of each benchmark correspond to the original case (the **O** bar) and the case with stride prefetching (the **S** bar). Each bar represents a percentage of the original amount of data transfer between the D-cache and the L2 cache, and it is further divided into three categories of D-cache misses: *load*, *store*, and *prefetch*. Ideally, we would like to convert all load misses into prefetch misses, while at the same time generate no extra prefetches (Note: D-cache *hits* will not show up in Figure 9 as they do not generate any requests to the L2 cache). Three benchmarks (`eon`, `gcc`, `gzip`) are missing in Figure 9 owing to a bug in Atom that prevents it from instrumenting the spiked version of these three programs. Fortunately, as we see Figure 8, these three benchmarks are of less interest since stride prefetching has little impact on their performance.

Figure 9 illustrates that the performance benefit we see in Figure 8 is due to the successful conversion of load misses into prefetches by stride prefetching. This is most noticeably in cases like `mcf`, `applu`, and `equake`. Figure 9 also shows that stride prefetching increases the total traffic by 10% or less—the only two exceptions are `twolf` and `facerec`. Overall, bandwidth does not appear to be a serious problem here.

4.2.2 Overhead of Stride Profiling

Recall from Figure 4 in Section 3.1 that there are two approaches to instrumenting programs for stride profiling: *naive* or *optimized*. Figure 10 compares the number of static instrumentation points generated by each approach. Obviously, the optimized approach is very effective at reducing the number of instrumentation points—it is reduced by a half or more in most cases. This advantage is a consequence of sharing instrumentation calls across loads that have the same base registers and of merging instrumentation calls at the same program points.

Figure 11 shows how much profiling time is actually saved by this reduction in instrumentation calls. The profiling time is expressed as a percentage of the time taken to run the *uninstrumented* program with the same training input. We first note that the stride profiling overhead is in the same order as that of some other software-based value profilers [6], which typically slow down the program by 10 to 30 times. Optimized instrumentation is very helpful to floating-point benchmarks—it cuts down their profiling time by nearly two-thirds on average. Nevertheless, it is less effective on the integer side. In fact, optimized instrumentation performs a little worse than naive instrumentation in `crafty`, `gap`, `gzip`, and `perlbmk`. A possible reason for this is that putting instrumentation at the definition point of the load's base register (as done in optimized instrumentation) may introduce additional overhead on other paths that go through the definition point. For these cases, other means like sampling become more important for reducing profiling overhead, as we will discuss in Section 4.2.4.

4.2.3 Performance Impact of Instrumentation Heuristics and Prefetch Minimization

In Section 3.1, we have introduced two heuristics for selecting critical loads to profile: *not-scalar* and *not-compiler-prefetched*. While they always reduce profiling time, their performance impact is less clear. Hence it is measured and shown in Figure 12. For each benchmark, the first three bars (from left to right) correspond to the following three scenarios: (i) *all* loads were instrumented, (ii) only *not-scalar*

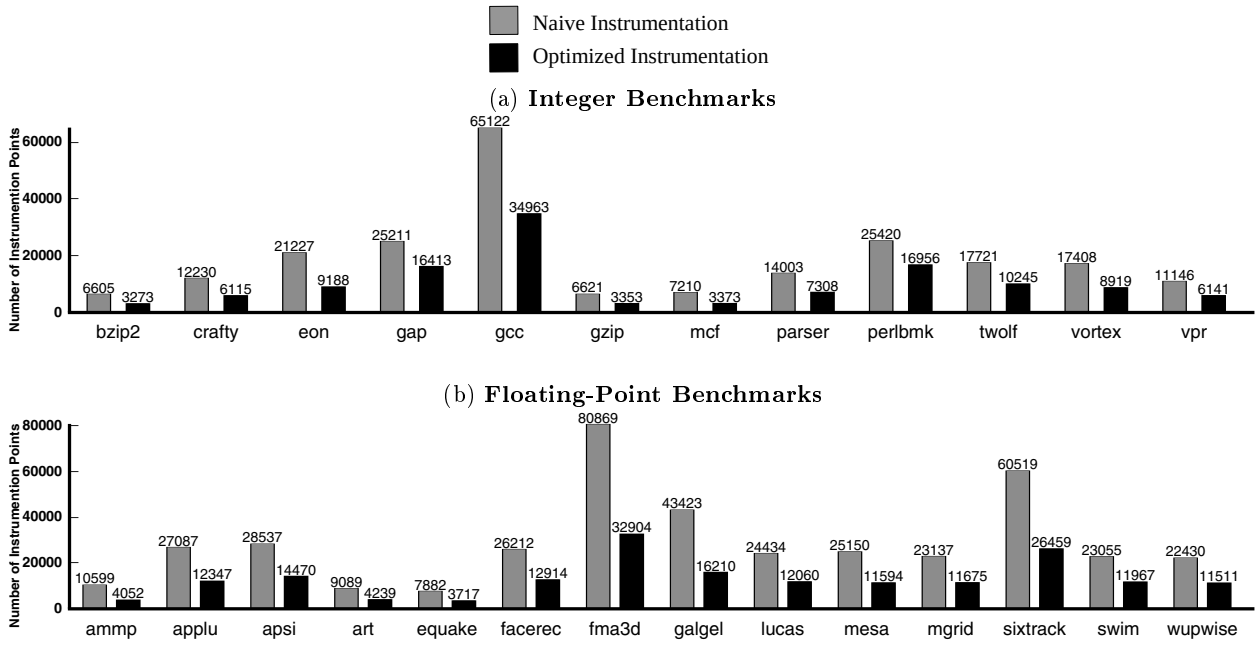


Figure 10: Number of static instrumentation points with *naive* and *optimized* instrumentation.

loads were instrumented, and (iii) only *not-scalar* and *not-compiler-prefetched* loads were instrumented. To isolate the performance impact of these heuristics, we disabled prefetch minimization (described in Section 3.3.3) in the first three bars. The last bar corresponds to adding prefetch minimization to scenario (iii). Note that the last bar is equivalent to the case already shown in Figure 8.

Figure 12 shows that little performance is lost with *not-scalar* and *not-compiler-prefetched* heuristics. In fact, they do improve performance in a few benchmarks, most prominently in `galgel`. This is because these heuristics avoid adding stride prefetches for loads that tend to hit in the cache or those that are already prefetched by the compiler.

Comparing the third and fourth bars in Figure 12 also reveals the effectiveness of prefetch minimization. While the average performance gain is small (1% for the floating-point benchmarks), it is particularly effective in `eon`, `applu`, and `galgel`.

4.2.4 *Effectiveness of Sampled Stride Profiling*

Our final set of results demonstrate the effectiveness of sampled stride profiling. This is the approach where we collect stride statistics for only the first 10,000 occurrences of each of the loads selected. The purpose of this experiment is to find out whether profiling time can be substantially saved by sampling while maintaining sufficient accuracy so that the performance gains are mostly preserved.

A comparison in the overhead of complete and sampled stride profiling is shown in Figure 13. As we can see, sampling speeds up the profiling process by 58% for integer and by 41% for floating-point. With sampling, the slowdowns of stride profiling are 2 to 15 times, which are quite encouraging given that the slowdowns of software-based value profilers [6] are typically 10 to 30 times.

The performance impact of sampled stride profiling is shown in Figure 14. The good news is that sampling does not degrade performance in most cases. The only exception is `mcf`, where about one-tenth of the strided loads found by stride profiling are different between complete and sampled stride profiling. Overall, sampling only the first 10,000 references works reasonably well for SPEC2000.

5. RELATED WORK

Prefetching has been an active area of research since it was first introduced. Meanwhile, many machines have adopted some form of prefetching, mostly in software [5, 15, 23, 30] with a couple in hardware [18, 37]. One way to categorize prefetching schemes is based on the data access patterns that they target. Hence, they can be broadly classified as sequential [33], strided [9, 14, 19], streamed [16, 21, 32], data-dependency based [28], pointer based [22, 25, 29], and address-correlation based [8, 20, 24]. Since our focus is on strided accesses, we compare our scheme with the other approaches to stride prefetching in the rest of this section.

Hardware-based stride prefetching has been studied by a number of researchers [9, 14, 19]. The general idea is to use a hardware table to remember for each load executed the last address loaded A_{i-1} and the difference, say S, between A_{i-1} and A_{i-2} (the second-to-last address loaded). Later on, when the same load is seen again with a new address A_i, a prefetch for the address $A_i + S$ will be launched if $A_i - A_{i-1}$ equals S, provided that the load's information has not been displaced from the table. The table entry for that load is then updated with the new address and the latest stride computed.

Comparing hardware-based vs. software-based stride prefetching (such as our scheme), the advantages of the hardware-based approach are that it requires no re-compilation or post-link optimizations, and it also poses no instruction over-

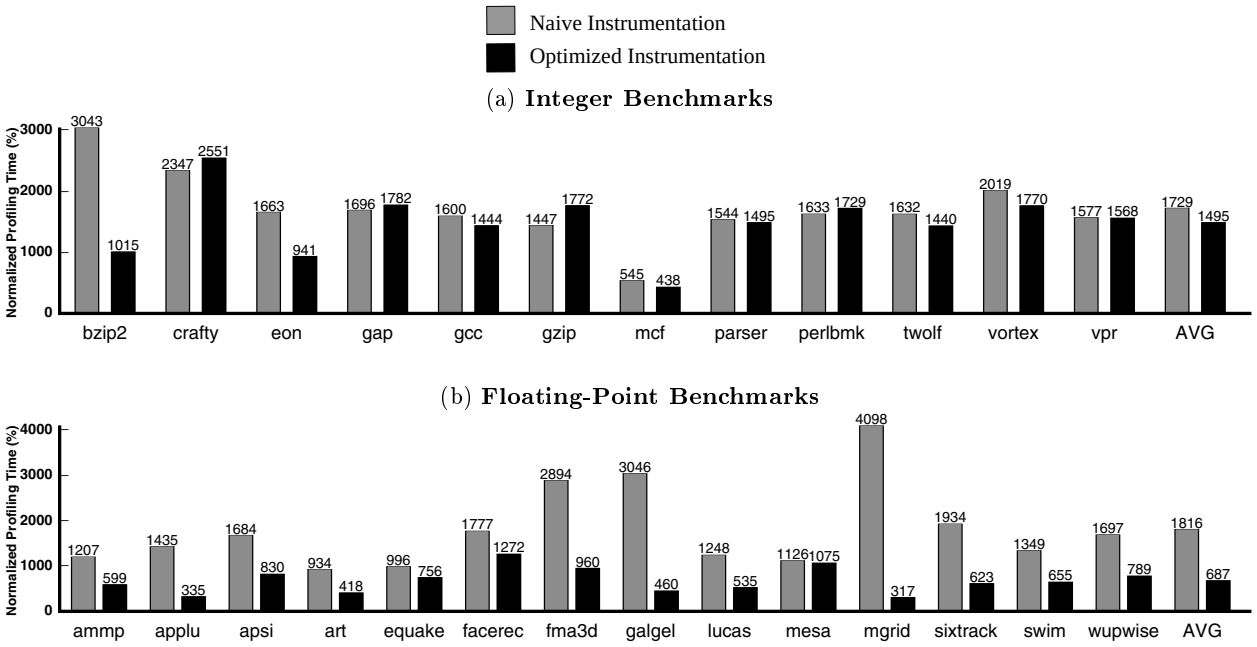


Figure 11: Overhead of stride profiling with *naive* and *optimized* instrumentation. The profiling time is normalized to the time taken to run the original (i.e. not instrumented) program on the same input (i.e. the training data sets).

head. On the other hand, the software-based approach has the advantages of being more flexible (e.g., in choosing the prefetching distance), requiring no hardware table (whose capacity limits the number of static loads that can be considered for prefetching), and most importantly being applicable to most existing machines which already support prefetch instructions.

On the software side, two compiler-based stride prefetching schemes have been proposed. Stoutchinin *et al.* [36] used the compiler to identify places in the program where strided accesses are likely to occur. In particular, the compiler focuses on prefetching *recurrent pointer updates*. In contrast, similar to our approach, Wu *et al.* [38] used stride profiling to guide prefetching. An advantage of these approaches is that the compiler could perform more detailed program analysis than a post-link tool, thereby leading to lower profiling overhead and/or higher performance. However, the major advantage of our approach is that it can be applied in the absence of the source code.

At the post-link level, Barnes *et al.* [4] used cache simulations to guide the insertion of stride prefetches into x86 binaries. Comparing their work against ours, we do not use cache simulations to select which loads need to be prefetched. Instead, we rely on the simple heuristics described in Section 3.1. As a result, we achieve lower profiling overhead than the cases where cache simulations are used. This reduced overhead is necessary for building a practical product.

6. CONCLUSIONS

We have extended the scope of automatic software-controlled prefetching in this study. By using profiling to detect statically unknown strides, a larger range of programs can be prefetched. And by inserting prefetches directly into executables, prefetching can be applied even without source code. The performance benefit of our technique is substantial: It accelerates over one-third of the SPEC2000 benchmarks by 3% to 56% on an Alpha 21264-based system. We also demonstrate that profiling overhead can be largely reduced by instrumentation heuristics and sampling. Our technique has been implemented as a product in the Compaq's Tru64 Unix.

7. ACKNOWLEDGMENTS

We thank John Williams for his help in incorporating stride profiling into `Pixie`.

8. REFERENCES

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of load/store instruction latencies. In *Proceedings of the 26th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 139–152, December 1993.

[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. In *Proceedings of the 15th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1990.

[3] J. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihi. Continuous profiling: Where have all the cycles gone. In *Proceedings of the 16th Symposium on Operating System Principles*, October 1997.

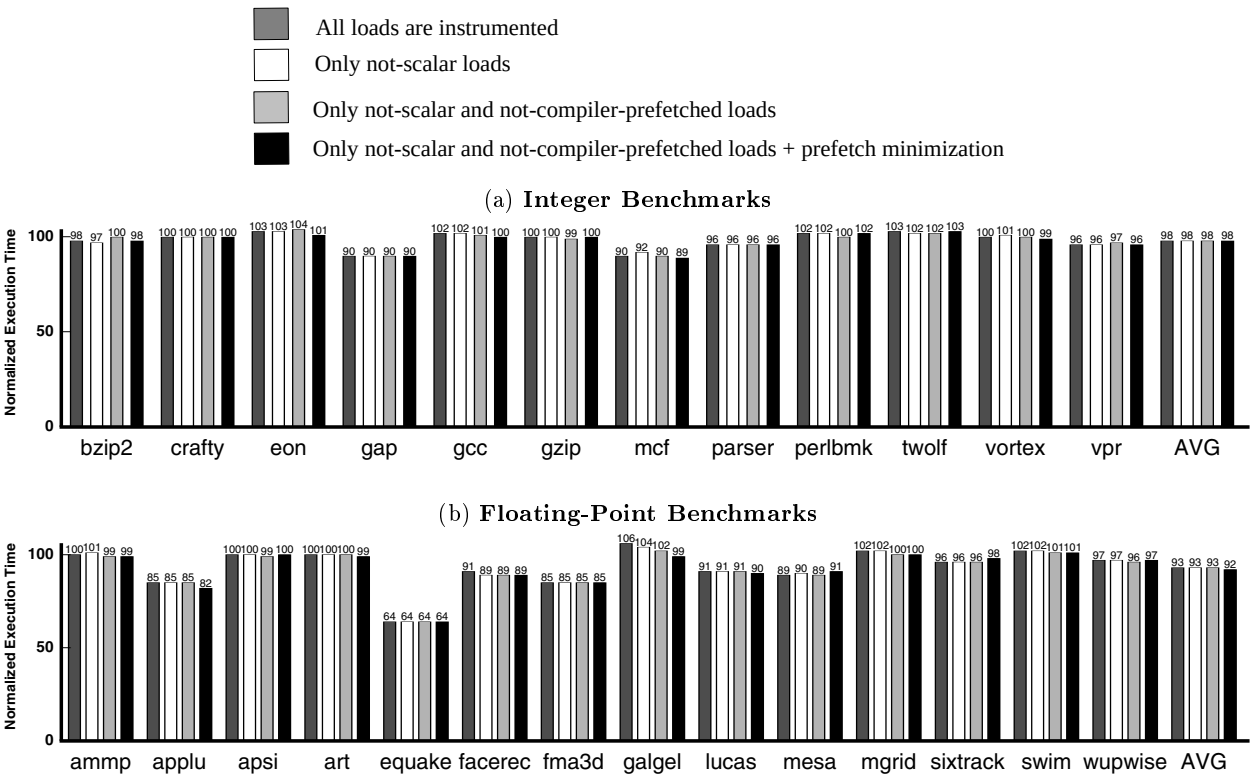


Figure 12: Performance impact of the two instrumentation heuristics (*not-scalar* and *not-compiler-prefetched*) and *prefetch minimization*. Execution time is normalized to the baseline.

[4] R. Barnes, R. Chaiken, and D. M. Gillies. Feedback-directed data cache optimizations for the x86. In *Second ACM Workshop on Feedback-Directed Optimizations*, November 1999.

[5] D. Bernstein, D. Cohen, A. Freund, and D. E. Maydan. Compiler techniques for data prefetching on the PowerPC. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques*, pages 19–26, June 1995.

[6] B. Calder, P. Feller, and A. Eustace. Value profiling and optimization. *Journal of Instruction Level Parallelism*, March 1999.

[7] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, April 1991.

[8] M. Charney and A. Reeves. Generalized correlation based hardware prefetching. Technical Report EE-CEG-95-1, Cornell University, Feb 1995.

[9] T.-F. Chen and J.-L. Baer. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, 44(5), May 1995.

[10] R. Cohn, D. Goodwin, and P. G. Lowney. Optimizing Alpha executables on Windows NT with Spike. *Digital Technical Journal*, 9(4):3–20, 1997.

[11] Compaq Computer Corporation. *AlphaServer DS20E Product Information*. http://www.compaq.com/alphaserver/ds20e/index.html.

[12] Compaq Computer Corporation. *Spike for Tru64 UNIX*. http://www.tru64unix.compaq.com/spike.

[13] R. Cytron, J. Ferrante, B. K. Rosen abd M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependency graph. Technical Report Technical Report RC14756, IBM, March 1991.

[14] F. Dahlgren and P. Stenström. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 7(4), 1996.

[15] G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing software data prefetches with rotating registers. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques*, September 2001.

[16] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.

[17] J. L. Henning. SPEC CPU2000: measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[18] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, and A. Kyker. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*,

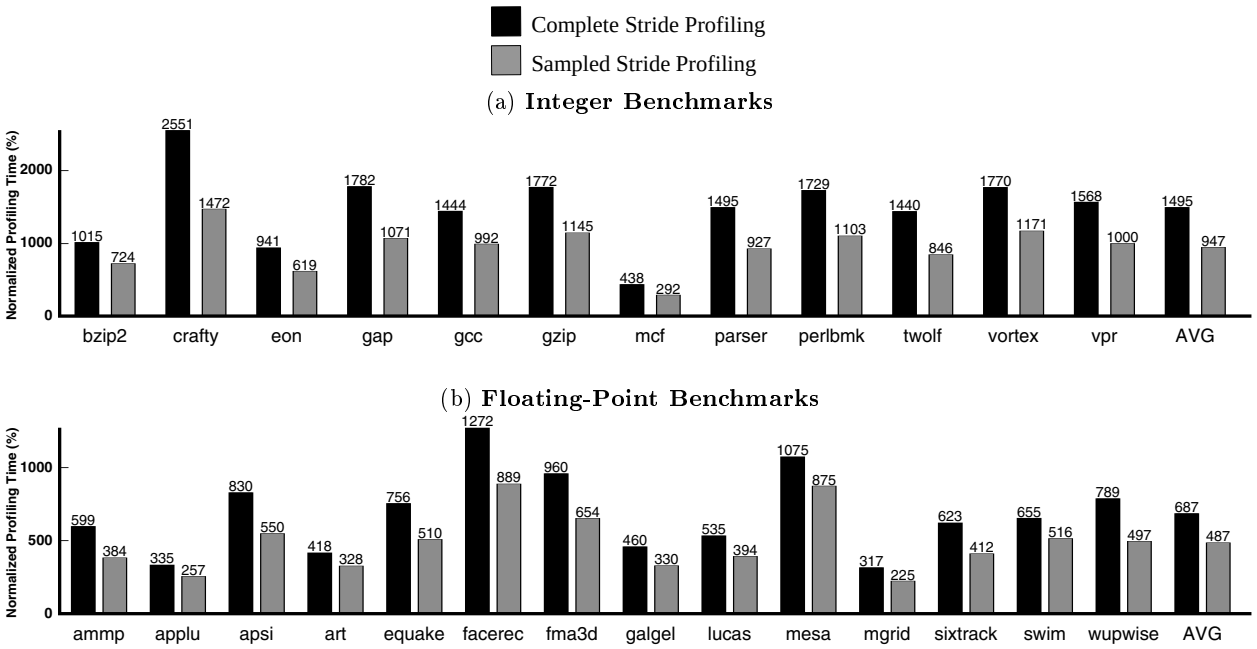


Figure 13: Overhead of *complete* and *sampled* stride profiling. The profiling time is normalized to the time taken to run the original (i.e. not instrumented) program on the same input.

Q1, February 2001.

[19] Y. Jegou and O. Temam. Speculative prefetching. In *Proceedings of the 1993 International Conference on Supercomputing*, pages 57–66, 1993.

[20] D. Joseph and D. Grunwald. Prefetching using Markov predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[21] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, May 1990.

[22] M. Karlsson, F. Dahlgren, and P. Stenström. A prefetching technique for irregular accesses to linked data structures. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture*, January 2000.

[23] R. E. Kessler. The Alpha 21264 Microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[24] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction and dead-block correlating prefetchers. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 144–154, June 2001.

[25] C.-K. Luk and T. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, October 1996.

[26] T. C. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, October 1992.

[27] T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 314–320, December 1997.

[28] A. Roth, A. Moshovos, and G. Sohi. Dependence based prefetching for linked data structures. In *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 115–126, October 1998.

[29] A. Roth and G. Sohi. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 111–121, May 1999.

[30] V. Santhanam, E. Gornish, and W.-C. Hsu. Data prefetching on the HP PA8000. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 264–273, June 1997.

[31] R. M. Shapiro and H. Saint. The representation of algorithm. Technical Report TR CA-7002-1432, Computer Associates, February 1970.

[32] T. Sherwood, S. Sair, and B. Calder. Predictor-directed stream buffers. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, December 2000.

[33] A. Smith. Sequential program prefetching in memory hierarchies. *IEEE Computer*, 11(2):7–21, 1978.

[34] M. D. Smith. Tracing with pixie. Technical Report CSL-TR-91-497, Stanford University, November 1991.

[35] A. Srivastava and A. Eustace. Atom: A system for

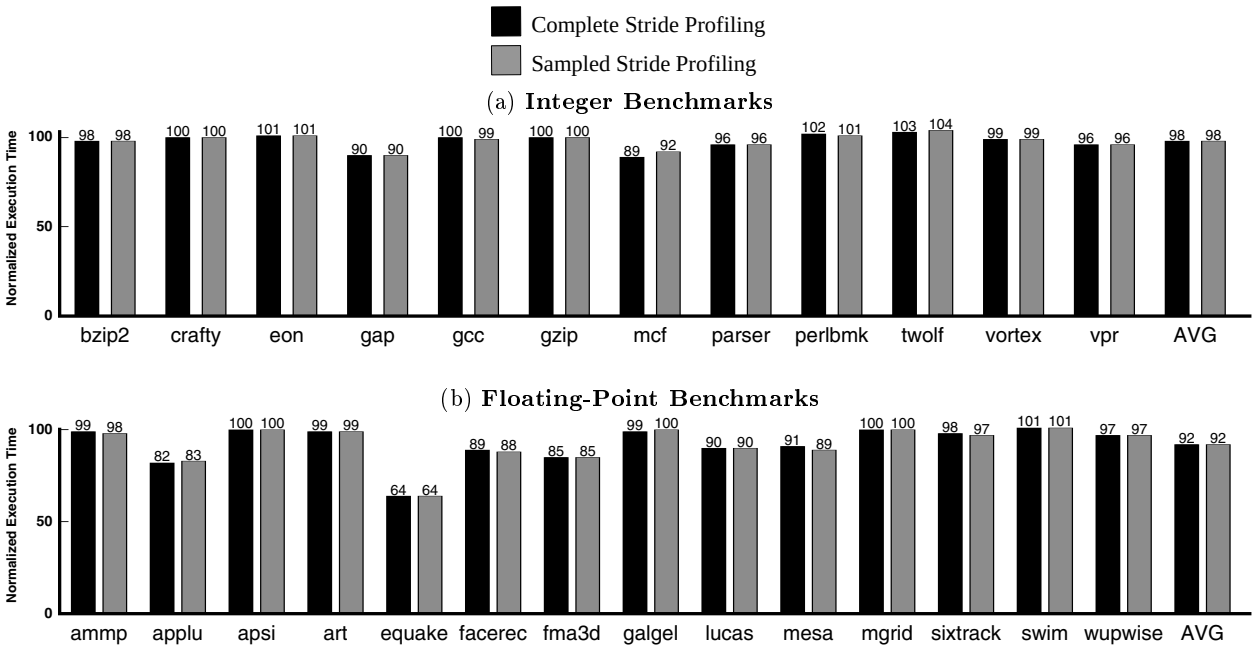


Figure 14: Performance impact of *sampled* stride profiling. Execution time is normalized to the baseline.

building customized program analysis tools. In *Proceedings of the ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, pages 196–205, 1994.

[36] A. Stoutchinin, J. Amaral, G. Gao, J. Dehnert, S. Jain, and A. Douillet. Speculative prefetching of induction pointers. *Lecture Notes in Computer Science*, (2027):289–303, 2001.

[37] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. *POWER4 System Microarchitecture*. IBM, October 2001.

[38] Y. Wu, M. Serrano, R. Krishnaiyer, W. Li, and J. Fang. Value profile guided prefetching for irregular code. In *Proceedings of the International Conference on Compiler Construction*, 2002.