# Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching

Youfeng Wu
Programming Systems Research
Intel Labs
2200 Mission College Blvd
Santa Clara, CA 95052
youfeng.wu@intel.com

## ABSTRACT

Irregular data references are difficult to prefetch, as the future memory address of a load instruction is hard to anticipate by a compiler. However, recent studies as well as our experience indicate that some important load instructions in irregular programs contain stride access patterns. Although the load instructions with stride patterns are difficult to identify with static compiler techniques, we developed an efficient profiling method to discover these load instructions. The new profiling method integrates the profiling for stride information and the traditional profiling for edge frequency into a single profiling pass. The integrated profiling pass runs only 17% slower than the frequency profiling alone. The collected stride information helps the compiler to identify load instructions with stride patterns that can be prefetched efficiently and beneficially. We implemented the new profiling and prefetching techniques in a research compiler for Itanium™ Processor Family (IPF), and obtained significant performance improvement for the SPECINT2000 programs running on Itanium machines. For example, we achieved a 1.59x speedup for *181.mcf*, 1.14x for *254.gap*, and 1.08x for *197.parser*. We also showed that the performance gain is stable across input data sets. These benefits make the new profiling and prefetching techniques suitable for production compilers.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *Compiler, Code generation, Memory management, Optimization*

## General Terms

Languages, Performance, Design, Experimentation, Algorithms

## Keywords:

Integrated stride and frequency profiling, Data prefetching, Strongly single-strided loads, Phased multi-strided loads, Performance evaluation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

*PLDI'02*, June 17-19, 2002, Berlin, Germany.
Copyright 2002 ACM 1-58113-463-0/02/0006…$5.00.

## 1. INTRODUCTION

Modern computer systems spend significant amount of time in processing memory references. For example, the Itanium systems consume nearly 40% of execution cycles [24] stalling on data cache and DTLB misses while running the SPECINT2000 benchmarks. Irregular programs such as the SPECINT2000 benchmarks contain many irregular data references caused by pointer-chasing code. Irregular data references are difficult to prefetch, as the future address of a memory location is hard to anticipate by a compiler or hardware [15][21]. However, recent studies suggest that some pointer chasing references exhibit stride patterns. Namely, the difference between two successive data addresses changes only infrequently at runtime. Stoutchinin et al. [26] and Collins et al. [5] notice that several important loads in 181.mcf benchmark of SPECINT2000 suite have stride reference patterns. Our experience indicates that many other SPECINT2000 programs, in addition to 181.mcf, contain important references with stride patterns. For example, the SPECINT2000 *197.parser* benchmark has code segments as shown in Figure 1. The first load at S1 chases a linked list and the second load at S2 references the string pointed to by the current list element. The program maintains its own memory allocation. The linked elements and the strings are allocated in the order that is referenced. Consequently, the address strides for both loads remain the same 94% of the time.

```
for (; string_list != NULL; string_list = sn) {
      S1:  sn = string_list->next;
      S2:  use string_list->string;
           other operations;
}
```

**Figure 1. Pointer-chasing code with stride patterns**

SPECINT2000 benchmark *254.gap* also contains pointer reference loads with stride patterns. An important loop in the benchmark performs garbage collection. A simplified version of the loop is shown in Figure 2. The variable ***s*** is a handle. The first load at the statement S1 accesses ****s*** and it has four dominant strides, which remain the same for 29%, 28%, 21%, and 5% of the time, respectively. One of the dominant stride occurs because the increment at S4. The other three stride values depend on the values in (**s&~3)->size* added to s at S3. The second load at the statement S2 accesses ***(*s & ~3)->ptr***. This access has two dominant strides, which remain constant for 48% and 47% of the time, respectively. These strides are mostly affected by the values in (**s&~3)->size* and by the allocation of the memory pointed to by *s.

```
while ( s < bound ) {
      S1:   if ((*s & 3 == 0) {                 // 71% time is true
      S2:         access (*s & ~3)->ptr
      S3:         s = s + ((*s & ~3)->size)+values;
                  other operations;
            } else if ((*s & 3 == 2) {          // 29% time is true
      S4:         s = s + *s;
            } else {
                  // never come here
            }
}
```

**Figure 2. Irregular code with multiple stride patterns**

Static compiler techniques [3][14][15][18][23][26], however, cannot easily discover the stride patterns in irregular programs. Pointer references make it hard for a compiler to understand the stride patterns of the load addresses. Also, the stride patterns in many cases are the results of memory allocation and compiler has limited ability to analyze memory allocation patterns. Without knowing that a load has stride patterns, it would be futile to insert stride prefetching, as doing so will penalize the references not exhibiting the prescribed strides.

Profiling has been used successfully to discover regularity among irregular behaviors. For example, edge profiling may identify highly biased branches to form large traces in control-intensive-programs [4]. Value profiling can reveal invariance in data variables to guide specialization optimizations [19]. Similarly, we will use profiling to identify regular stride patterns in irregular programs. We will call this profiling the *stride profiling*. Stride profiling is a potentially very slow process when every reference is profiled. An important observation to reduce the profiling overhead is that stride patterns often exhibits in loads executed inside loops with relatively high trip counts. Thus selectively profiling these loads leads to significant reduction in profiling overhead. Furthermore, stride patterns of load is often statistically stable and can be discovered with sampling of the references.

In this paper, we develop an efficient profiling method to discover loads with stride patterns. We integrate the stride profiling into the traditional frequency-profiling pass and the combined profiling pass is only slightly slower than the frequency profiling alone. The resulting stride profile is used to guide compiler prefetching during the profile feedback pass.

The example in Figure *3* illustrates our profiling and prefetching techniques. Figure *3* (a) shows a typical pointer-chasing loop. For simplicity, we assume that the data address of the load reference P->data at L is **P**. The compiler instruments the loop for both block frequency and stride profiling as shown in Figure *3* (b). The operations freq[b1]++ and freq[b2]++ collect block frequency information for the loop pre-head block b1 and the loop entry block b2. The conditional assignment "pr = (r2/r1) > 128" sets the predicate pr to true when the loop trip count is greater than 128, and false otherwise. The profiling runtime routine *strideProf* is guarded by the predicate *pr* and it is actually invoked only when the predicate pr is true. After the instrumented program is run, the stride profile is fed back and analyzed. The profile could indicate that the load at L has the same stride, e.g. 60 bytes, 80% of the time. In this case, the compiler can insert prefetching instructions as shown in Figure *3* (c), where the inserted instruction prefetches the load two strides ahead (120=2*60). The compiler decides the number of iterations ahead using heuristics to be described in this paper. In case the profile indicates that the load has multiple dominant strides, e.g. 30 bytes 40% of the time and 120 bytes 50% of the time, the compiler may insert prefetching instructions as shown in Figure *3* (d) to compute the strides before prefetching. Furthermore, the profile may suggest that a load has a constant stride, e.g. 60, sometimes and no stride behavior in the rest of the execution, the compiler may insert a conditional prefetch as shown in Figure *3* (e). The conditional prefetch can be implemented on Itanium using predication [11].

```
While (P)          // b1
L:  D= P->data   // b2
    Use (D)
    P = P->next
```

(a) A pointer-chasing loop

```
r1=freq[b1]++
r2=freq[b2]
pr =( r2/r1) > 128
While (P)               // b1
    freq[b2]++
    pr? strideProf(P)
L:  D= P->data       // b2
    Use (D)
    P = P->next
```

(b) Stride profiling code

```
While (P)
    prefetch(P+120)
L:  D= P->data
    Use (D)
    P = P->next
```

(c) Single stride prefetching

```
prev_P = P
While (P)
    stride = (P-prev_P);
    prev_P=P;
    prefetch(P+2*stride)
L:  D= P->data
    Use (D)
    P = P->next
```

(d) Multi-stride prefetching

```
prev_P = P
While (P)
    stride = (P-prev_P);
    prev_P=P;
    pr = ( stride == 60)
    pr? prefetch(P+120)
L:  D= P->data
    Use (D)
    P = P->next
```

(e) Conditional single-stride prefetching

**Figure 3. Example of stride profile guided prefetching**

This paper makes the following contributions.

- We develop an efficient method to collect both stride profile and the traditional frequency profile in the same profiling pass. The integrated profiling pass runs only 17% slower than the frequency profiling alone, and significantly faster than the naïve stride profiling methods.
- Using the stride profile to guide compiler prefetching, we obtain significant performance improvement for the SPECINT2000 programs running on the Itanium machines. For example, we achieve a 1.59x speedup for "181.mcf", 1.14x for "254.gap", 1.08x for "197.parser". These performance improvements, with an average of 7% for the entire benchmark suite, are significant for highly optimized SPECINT2000 programs running on real machines.
- We show that the performance gain is stable across input data sets. We collect the stride profiles with both the reference-input data set and the train-input data set. The performance difference between the binaries compiled using the two profiles running with the reference input data set is small.

The rest of the paper is organized as follows. Section 2 overviews the prefetching algorithm. Section 3 presents the methods for efficient stride profiling. Section 4 provides the experimental results. Section 5 discusses the related work. Section 6 concludes the paper and points out the future research directions.

## 2. OVERVIEW OF PREFETCHING ALGORITHM

For easy of discussions, we will refer a load that is selected for stride profiling as a *profiled load*, and a load that is selected for prefetching as a *prefetched load*. Also, we will refer to a load inside a loop as an *in-loop load* and a load not inside any loop an *out-loop* load. For a load inside an irreducible loop, we will treat it as an out-loop load.

In the following subsections, we outline the prefetching algorithm using the generation and feedback of the stride profiles. We first focus on in-loop loads, and then explain how to handle out-loop loads.

### 2.1. Stride Profile Generation

The compiler identifies the profiled loads for stride profiling with heuristics. The naive method is to select all loads as profiled loads. Another heuristic is to select loads inside loops with high trip counts, if the frequency profile is available.

The set of profiled loads can be refined to reduce profiling overhead. We call a set of loads equivalent if they are inside the same loop, in control equivalent blocks, and their addresses are different only by compile-time constants. They will have the same stride values or their strides can be derived from the stride for another load. The compiler selects only one of them as the representative to be profiled.

For each profiled load, the compiler inserts profiling code to collect its stride profile. When the instrumented program is run, the profiling runtime routine collects two types of information for the profiled load: stride value profile and stride difference profile.

The *stride value profile* collects the top N most frequently occurred stride values and their frequencies. An example for N = 2 is shown in **Figure 4** (a). For the 10 stride values from a profiled load, the profiling runtime routine identifies that the most frequently occurred stride is 2 with a frequency of 5, and the second mostly occurred stride is 100 with a frequency of 4.

The *stride difference profile* collects the top M most frequently occurred differences between successive strides and their frequencies. An example for M = 1 is shown in **Figure 4** (b). For the nine stride differences, the profiling runtime routine identifies that the most frequently occurred stride difference is 0 with a frequency of 7.

The stride difference profile is used to distinguish a phased stride sequence from an alternated stride sequence when they have the same stride value profile. The stride sequence shown in **Figure 4** (a) is a phased stride sequence. A phased stride sequence is characterized by the fact that its top stride difference is zero. An alternated stride sequence is shown in **Figure 4** (c), which has the same stride value profile as the phased stride sequence in **Figure 4** (a). However, its most frequently occurred stride difference is not zero. A phased stride sequence is better for prefetching than an alternated stride sequence as the stride values in phased stride sequence remain a constant over a longer period, while the strides in an alternated stride sequence frequently change.

We use the value-profiling algorithm proposed in [2] to collect the stride profile. The detailed profiling algorithm will be given in Section 3.

Stride sequence
2, 2,2,2,2,100,100,100, 100, 1

top[1] = 2, freq[1] = 5
top[2] = 100, freq[2] = 4
Total strides = 10

(a) Stride values and top strides identified

Stride differences
0, 0, 0, 0, 98, 0, 0, 0,-99

dtop[1] = 0, dfreq[1] = 7
Total differences = 9

(b) Stride differences and top difference identified

Stride sequence
2,100,2,100,2,100,2,100,2,1
Stride differences
98,-98,98,-98,98,-98,98,-98,-1

top[1] = 2, freq[1] = 5
top[2] = 100, freq[2] = 4
Total strides = 10

dtop[1] = 98, dfreq[1] = 4
Total differences = 9

(c) Same top strides but different top difference

**Figure 4. Stride value and stride difference profile example**

### 2.2. Stride Profile Feedback

During the profile feedback pass, the compiler reads both the stride profile and the frequency profile to guide its prefetching decision. It first uses the frequency profile to filter out loads that obviously will not benefit from stride prefetching, such as not inside a loop with high trip count.

It then identifies loads with stride patterns according to the following criteria and filters out all loads that do not satisfy at least one of the criteria:

- Strong single stride (SSST) load: A load with one non-zero stride that occurs with a very high probability (e.g. at least 70% of the time). We will refer to the threshold 70% as SSST_threshold.
- Phased multi-stride (PMST) load: A load with multiple non-zero strides that together occur frequently (e.g. at least 30% of the time) and the differences between the strides are frequently zeroes (e.g. at least 20% of the time). For example, a phased multi-stride load may have the stride values 32, 60, 1024 together occur more than 60% of time and 40% of the stride differences are zeros. We will refer to the threshold 60% as PMST_threshold, and the threshold 40% PMST_diff_threshold.
- Weak single stride (WSST) load: A load with only one of the non-zero stride values that occurs somewhat frequently (e.g. at least 20% of the time) and the stride differences are sometime zeros (e.g. at least 10% of the time). For example, a weak single stride load may have a stride 32 in 25% of time, and the stride differences are zeroes 10% of the time. We will refer to the threshold 25% as WSST_threshold, and the threshold 10% WSST_diff_threshold.

Any of the remaining loads may represent an equivalent set of loads with the same stride patterns. For each set of equivalent loads, although only one of them may be profiled, multiple of them may need to be prefetched. To decide which ones to prefetch, the compiler analyzes the range of cache area accessed by the loads in the set. Enough loads will be prefetched to cover the cache lines in that range. The selected loads are the *prefetched loads*.

The process to identify prefetched loads is outlined in Figure 5, where FT is the frequency threshold, e.g. 2000, and TT is the trip count threshold, e.g. 128. Each profiled load uses a data structure *prof_data* to store its stride profile information. The fields *freq*[1] to *freq*[4] stores the frequencies of the top 4 non-zero strides. The field *total_freq* counts the number of total strides profiled. The field *num_zero_diff* remembers the number of zero stride differences.

For each prefetched load, the compiler inserts prefetching instructions according to the type of the load. Assume a prefetched load has a load address P.

If this is a *strong single stride* load and the dominant stride value is S, the compiler inserts one prefetch instruction "*prefetch (P+K*S)*" in the same basic block before the load instruction, where K*S is a compile-time constant. The constant K is the prefetch distance determined by compiler analysis.

```
SSST_loads = {}
PMST_loads = {}
WSST_loads = {}

for each profiled load {
    if ((load-> freq <= FT)
        continue;           // load is filtered out
    if (load is an in-loop load && load->loop->trip_count<=TT)
        continue;           // load is filtered out
    pdata = load->prof_data
    top1freq = pdata->freq[1]
    top4freq = pdata->freq[1] + pdata->freq[2]
               + pdata->freq[3] + pdata->freq[4]
    num0diff = pdata->num_zero_diff
    totalfreq = pdata->total_freq
    cover_loads = loads that cover the cache lines accessed
                  by the equivalent set represented by load
    if (top1freq / totalfreq > SSST_threshold)
        SSST_loads ∪= { cover_loads }
    elseif (top4freq / totalfreq > PMST_threshold
         && num0diff / totalfreq > PMST_diff_threshold)
        PMST_loads ∪= { cover_loads }
    elseif (top1freq / totalfreq > WSST_threshold&&
         && num0diff / totalfreq > PMST_diff_threshold)
        WSST_loads ∪= { cover_loads }
}
```

**Figure 5. Identify prefetched loads during profile feedback**

For an in-loop load with a trip count *trip_count* and a dominant stride *stride*, the size of the referenced data area is (trip_count * stride). If this size is larger than the size of a cache level with L cycle miss latency, we can assume that it takes at least L cycles. If the loop body takes B cycles without taking the miss latency of prefetched loads into account, then K can be computed as $\min(\lceil L/B \rceil, C)$, where C is the maximum prefetch distance (e.g. 8). The value of K may also be determined using loop trip count value as follows:

$$K = \min(\lceil trip\_count / TT \rceil, C)$$

where TT is the trip count threshold (e.g. 128).

If this is a *phased multi-stride* in-loop load, the compiler inserts the prefetching instructions as follows:

1. Insert a move instruction before the load operation to save its address in a scratch register.
2. Insert a subtract instruction before the move instruction to subtract the value in the scratch register from the current address of the load. Place the difference in a register called *stride*.
3. Insert a "prefetch (P+K**stride*)" instruction before the load, where K is determined as described previously, but rounded to a power of two to avoid the multiplication operation.

If this is a *weak single stride* in-loop load, the compiler inserts prefetching instructions in the same way as the steps 1 and 2 described for a phased multi-stride load. Step 3 is modified as follows:

3'. Insert the following conditional prefetching before the load.

    p = (stride == profiled stride)

    p? prefetch (P+K*stride)

The reason for a conditional prefetching is to reduce the number of useless prefetches, when the load exhibits only occasional stride patterns.

## 2.3. Handling Out-Loop Loads

Although the subsections above consider only in-loop loads for profiling and prefetching, we can also profile and prefetch out-loop loads. Our experiments show that out-loop loads are much more expensive to profile and also more difficult to prefetch.

An out-loop load is hard to prefetch primarily because the prefetching code sequences for a phased multi-stride and a weak single stride load do not work for an out-loop load. The successive executions of the out-loop load will be from different invocations of the current function. The address value saved in the scratch register will be lost when the function returns. We would need a static memory location for each such load to store the address value for computing its stride value. The load and store to the static memory location increases prefetching overhead. For this reason, we will not prefetch out-loop loads when they are classified as phased multi-stride or weak single stride loads. Another reason why an out-loop load is not easy to prefetch is that the heuristic for determining the prefetching distance for an in-loop load no longer works for an out-loop load. The values used to estimate prefetching distance, such as trip counts and the loop body latencies, are unavailable for an out-loop load. As a result, we will prefetch only SSST out-loop loads and use a fixed constant, such as 4, as the prefetching distance.

## 3. EFFICIENT STRIDE PROFILING

Stride profile is collected by instrumenting each profiled load to invoke a profiling runtime routine. The profiling overhead can be reduced in two directions. First, the profiling routine should be made as efficient as possible. Second, the invocation to the runtime routine can be made less frequent. We develop an efficient profiling method along both directions.

## 3.1. Efficient Profiling Runtime Routine

The profiling runtime routine is outlined in Figure 6, where *address* is the address for the load being profiled, and *prof_data* is the data area allocated for the load to collect profiling result. The profiling runtime routine calls the Least Frequently Used (LFU) replacement algorithm described in [2] to collect stride profiles. LFU manages two buffers, a temp buffer and a final buffer, to keep track of the most frequently recurrent strides. For each stride passed to LFU, if the stride is already in an entry in the temp buffer, the frequency count of the entry is incremented. If no entry is found for the stride, the least frequently used entry in the temp buffer is replaced. The temp buffer is periodically merged with the final buffer by selecting the strides with the highest frequencies in both buffers into the final buffer. At that time, the temp buffer is cleared.

```
strideProf(address, prof_data) {
    stride = address – prof_data->prev_address
    if (stride == 0)
        prof_data->num_zero_stride ++
        return
    diff = stride – prof_data->prev_stride
    if (diff == 0)
        prof_data->num_zero_diff ++
    else
        prof_data->prev_stride = stride
    prof_data->prev_address = address
    LFU(stride, prof_data)
}
```

**Figure 6. Profiling runtime routine**

```
int is_same_value (a1, a2) {
    if ((a1>>4) == (a2>>4))
        return 1
    return 0
}
strideProf(address, prof_data) {
    if (is_same_value(address, prof_data->prev_address)
        prof_data->num_zero_stride ++
        return
    stride = address – prof_data->prev_address
    diff = stride – prof_data->prev_stride
    if (diff == 0)
        prof_data->num_zero_diff ++
    else
        prof_data->prev_stride = stride
    prof_data->prev_address = address
    LFU(stride, prof_data)
}
```

**Figure 7. Enhanced profiling runtime**

```
Address sequence:
    2,4,6,8,10,12,14,16,18,20,...

Addresses with fine sampling F = 4:
    2, , , ,10,  ,  ,  ,18, ,...

Stride with fine sampling: S1 = 8
Original stride: S2 = 8/F = 2
```

**Figure 8. Example of fine sampling**

We could also use the LFU algorithm to collect the stride difference profile. However, we can simply count the number of zero differences between successive strides. If the percentage of the zero differences is high, we know that the stride sequence is phased.

Since LFU is a heavy-duty operation, the number of zero strides are also checked and counted without going through the LFU operation.

The execution speed of the LFU algorithm depends on the number of different values it tracks in its buffers. The less number of different values it tracks, the faster it runs. Notice that stride prefetching often remains effective when the stride value changes slightly. For example, the prefetching at *address* and the prefetching at *address+8* should not have much performance difference, if the cache line is large enough to accommodate the data at both addresses. Thus, the profiling runtime routine may reduce the number of different strides it tracks by treating the strides that are different by a small value, such as half the size of a cache line or less, as the same. The enhanced profiling routine is shown in Figure 7. The subroutine *is_same_value* considers two values the same if they differ only in the last 4 bits. *Is_same_value* is used inside the LFU routine whenever it compares any two strides for equality.

```
strideProf(address, prof_data) {
    // chunk sampling
    static int number_skipped = 0
    static int number_profiled = 0
    if (number_skipped < N1)
        number_skipped ++;
        return;
    else if (number_profiled == N2)
        number_profiled = 0;
        number_skipped = 0;
        return;
    number_profiled ++;

    // fine sampling
    if (prof_data->number_to_skip > 0)
        prof_data->number_to_skip --;
        return;
    prof_data->number_to_skip = F-1;

    if (is_same_value(address, prof_data->prev_address))
        prof_data->num_zero_stride ++
        return
    stride = address – prof_data->prev_address
    diff = stride – prof_data->prev_stride
    if (diff == 0)
        prof_data->num_zero_diff ++
    else
        prof_data->prev_stride = stride
    prof_data->prev_address = address
    LFU(stride, prof_data)
}
```

**Figure 9. Profiling runtime routine with sampling**

We may further reduce profiling overhead by introducing sampling into the profiling runtime routine. The stride profile can be sampled in two ways. First, for a sequence of load addresses, we may take one sample after every F references, where F is constant such as 4. We call this the fine sampling method. A stride value S1 collected with fine sampling relates to the original stride S2 by the equation S1 = F*S2. Thus the compiler can derive the original stride value as S2 = S1/F. An example of fine sampling is shown in Figure 8.

The second sampling method is for chunks of references. In this method, after every N1 references are skipped, the profiling runtime routine profiles the next N2 references. Example values for N1, and N2 are 8 millions and 2 millions, respectively. The profiling runtime routine with sampling is shown in Figure 9.

## 3.2. Integrated Profiling

To reduce the frequency that the profiling runtime routine is called, we may select only the loads inside a loop with a high trip count, e.g. > 128, for profiling. This trip count condition indicates that the load is likely to touch a large range of memory and will benefit from stride prefetching. For example, the range [x, x+stride * trip_count] of memory area will be touched. Our experiment indeed shows that the majority of loads that can benefit from stride prefetching are inside loops with high trip counts (see Section 4). Therefore, restricting profiling and prefetching to in-loop loads with high trip counts does not impact the performance. Furthermore, the number of dynamic loads inside loops with high trip counts is small. For example, our data indicates that about 7.5% of dynamic loads are inside loops with trip counts > 128 for the SPECINT2000 benchmarks. As a result, restricting prefetching to in-loop loads with high trip counts can significantly reduce profiling overhead.

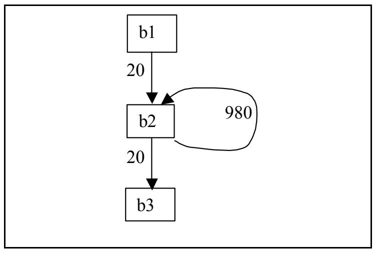

**Figure 10. Example for determining trip count**

The trip count of a loop is calculated as the ratio of the entry block frequency over the entering frequency from outside of the loop to the entry block of the loop. For the loop with edge frequencies as shown in Figure 10, the trip count is calculated as follows.

$$TC = \frac{freq(b_2 \to b_2) + freq(b_2 \to b_3)}{freq(b_1 \to b_2)} = \frac{980 + 20}{20} = 50.$$

Checking the trip count condition needs the frequency profile information. Although the frequency profile is usually available for compiler optimizations [4], we would need a separate pass to collect the stride profile by using the frequency profile generated in an early pass. This *two-pass* method poses a usability problem. The additional pass to collect stride profiles could place a significant burden on software development. This is especially painful for a cross compilation environment, in which the compilation and execution are on different machines, and a lot of manual work is involved to execute the instrumented program to obtain the profiles.

In the following, we develop a number of one-pass methods to collect both frequency and stride profiles in the same profiling pass. All the methods select profiled loads without using frequency profile information and rely on the frequency profile during the profile feedback pass to filter out unwanted loads (such as using the routine shown in Figure 5).

The first method simply collects the stride profile for every load while collecting the frequency profile. The method can be used to profile in-loop loads as well as out-loop loads. If it is used only to profile in-loop loads, we will call it the *naïve-loop* method. When it is used to collect stride profiles for both in-loop and out-loop loads, we will call it the *naïve-all* method. Notice that after the unwanted loads are filtered out, the stride profile collected by the naïve-loop method is the same as that collected by the two-pass method.

The naïve-loop method can be improved in two ways. First, loads with loop-invariant memory addresses should not be profiled, as their stride values are usually zero and cannot benefit from stride prefetching. Second, the partially collected frequency profile information may be used to check for the trip count condition, and only when the trip count condition is satisfied, should the calls to the profiling runtime routine inside the loop be activated. This process is efficient as the check code is inserted outside the loop and the profiling runtime routine is not invoked inside the loop unless the condition is satisfied. The following methods include both of the improvements.

The stride profiling can be integrated with either the block frequency profiling or the edge frequency profiling, depending on which frequency profiling the compiler uses. If the compiler uses the block frequency profiling already, the stride profiling can be integrated to use the partially collected block frequency information to check for the trip count condition. We will call this method the *block-check* method. In Figure 11 (a), two blocks b1 and b2 are instrumented for block frequency profiling. For the load in block b2, the naive method would instrument block b2 as shown in Figure 11 (b). If the loop turns out to have a low trip count, the stride-profiling overhead could be wasted. The block-check method instrument blocks b1 and b2 as shown in Figure 11 (c), where the register r1 stores the frequency of the loop pre-head block and r2 contains the frequency of the loop entry block. It computes the trip count by r2/r1 and checks the trip count condition in block b1, and if this condition is satisfied it sets a predicate register to true. Block b2 uses the predicate to guard the invocation to the profiling runtime routine. To avoid division or multiplication overhead, we use r1<(r2 >>W) to compute r2/r1 > TT, where TT is the trip count threshold and W = $\lfloor \log_2 TT \rfloor$ is a compile-time constant.

Similarly, if the compiler uses the edge frequency profiling already, the stride profiling can be integrated to use the partially collected edge frequency information to check for the trip count condition. We will call this method the *edge-check* method. For the example in Figure 12 (a), the edges e1=(b1→b2), e2=(b2→b2), and e3=(b2→b3) are instrumented for edge frequency profiling. Although there is no block frequency information for the loop entry block b2, the frequency of block b2 can be computed as the sum of frequencies of its outgoing edges (freq[e2] + freq[e3]). The instrumentation for both edge frequency and stride profiling is shown in Figure 12 (b).

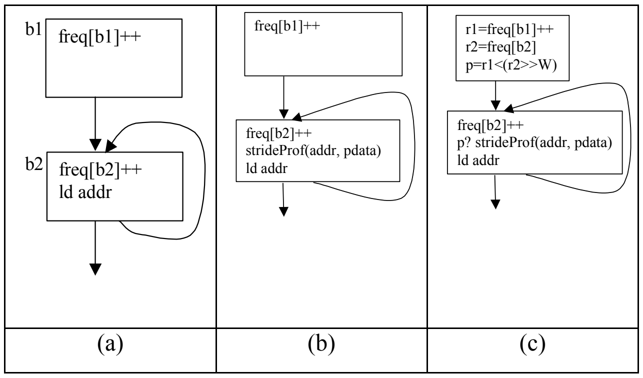

**Figure 11. Example of block-check method**

In general, the loop entry block may have any number of outgoing edges. To find the entry block frequency, we need to sum up all frequencies on the outgoing edges. Similarly, the entry block may have any number of the incoming edges from outside of the loop. The pre-head frequency is the sum of all the frequencies on the incoming edges. Figure 13 shows a general loop and the instrumentation for collecting both the edge frequency and the stride profiles. The instrumentation process for the edge-check method is shown in Figure 14.

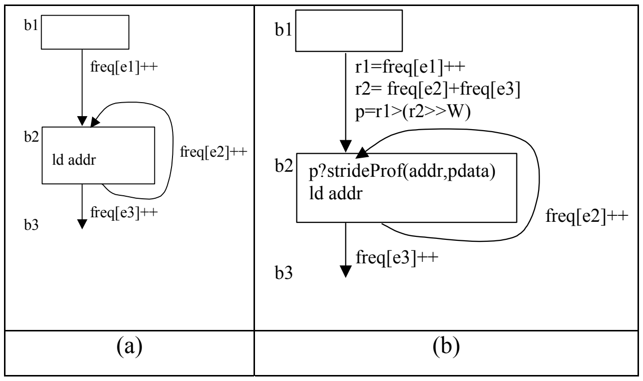

**Figure 12. Example of edge-check method**

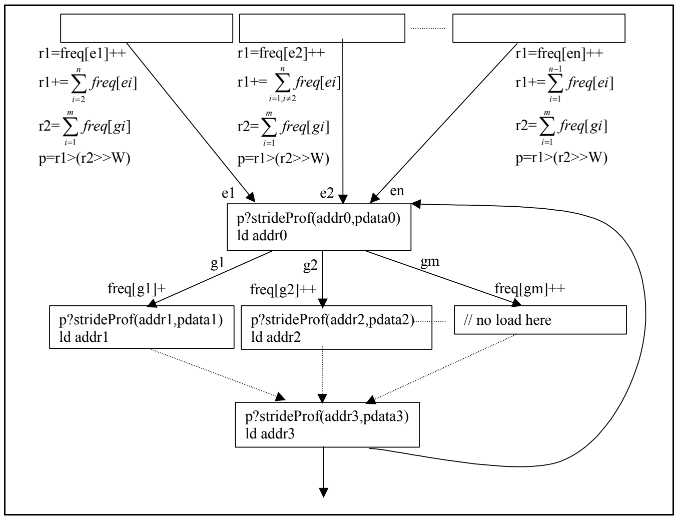
r1=freq[e1]++
r1+= $\sum_{i=2}^{n} freq[ei]$
r2= $\sum_{i=1}^{m} freq[gi]$
p=r1>(r2>>W)
r1=freq[e2]++
r1+= $\sum_{i=1,i\neq 2}^{n} freq[ei]$
r2= $\sum_{i=1}^{m} freq[gi]$
p=r1>(r2>>W)
r1=freq[en]++
r1+= $\sum_{i=1}^{n-1} freq[ei]$
r2= $\sum_{i=1}^{m} freq[gi]$
p=r1>(r2>>W)
e1
e2
en
p?strideProf(addr0,pdata0)
ld addr0
g1
g2
gm
freq[g1]+
freq[g2]++
freq[gm]++
p?strideProf(addr1,pdata1)
ld addr1
p?strideProf(addr2,pdata2)
ld addr2
// no load here
p?strideProf(addr3,pdata3)
ld addr3

**Figure 13. Example of general edge-check method**

After the instrumented program is executed, both frequency and stride profiles are generated. The frequency profile is exactly the same as that would be collected in a separate pass. However, the stride profile collected by the edge-check or block-check method, after the unwanted loads are filtered, may be different from what it would be collected by the two-pass method. This is because the invocation to profiling runtime routine may be turned on and off during the execution. In particular, the two-pass method may profile loads inside a high-trip count loop whose loop nest is executed only once. The block-check and edge-check methods, on the other hand, will not have stride profile information for loads inside loop whose loop nest is executed only once.

```
Instrument_edge_check() {
    // don't profile loads whose addresses are loop invariant
    for each loop
        for each load
            if (load->addr is not a loop invariant)
                mark load as a profiled load
        identify equivalent loads and reduce profiled loads
    for each loop that contains a profiled load
        // reserve a predicate register for the loop
        loop->predicate = create a new predicate
    for each edge
        // collect edge frequency
        insert the following instructions on the edge
            "r1 = load edge's counter"
            "r1++"
            "store r1 to edge's counter"
        // compute trip count predicate
        if the edge is an incoming edge from outside to the loop
            for every other incoming edge e of the loop
                insert on the edge
                    "r1 += load e's counter"
            insert "r2 = 0" on the edge
            for every outgoing edge e of the loop entry block
                insert on the edge
                    "r2 += load e's counter"
            insert on the edge
                "r2 = r2 >> W"
                "loop->predicate = r2 > r1"
    // guard strideProf with trip count predicate
    for each loop
        for each profiled load inside the loop
            pr = loop->predicate
            address = load's data address
            pdata = load->prof_data
            if the load is a predicated instruction
                insert the following instruction before load
                    "pr1 = pr && load->predicate"
                    "pr1 ? strideProf(address, pdata)"
            else
                insert the following instruction before load
                    "pr ? strideProf(address, pdata)"
}
```

**Figure 14. Instrumentation for edge-check method**

## 4. EXPERIMENTAL RESULTS

We implemented the new profiling and prefetching algorithms in a research compiler for the Itanium Processor Family (IPF). The compiler is based on a production compiler with additional components to make compiler and architectural exploration easier. The generated codes are highly optimized with all of the inter-procedural, intra-procedural, architectural specific, and profile-guided optimizations, assisted with aggressive memory disambiguation and whole program knowledge [8][10][13]. Our compiler automatically performs the profiling and stride-profile guided prefetching transformation without any hand coding involvement. The prefetching for weak single strided load (WSST) is not enabled for this paper, as it does not show noticeable performance contribution.

All our experiments are performed on a 733 MHz Itanium machine [11] running the SPECINT2000 benchmarks ([9] see Figure 15). The Itanium machine used in our experiment have a 16KB 4-way set associative L1 data cache, a 96KB 6-way set associative unified L2 cache, 2MB 4-way set associative unified L3 cache, and 1 GB memory.

| Programs | Lang | Description |
|---|---|---|
| 164.gzip | C | Compression/Decompression |
| 175.vpr | C | FPGA circuit placement and routing |
| 176.gcc | C | C programming language compiler |
| 181.mcf | C | Combinatorial Optimization |
| 186.crafty | C | Game Playing: Chess |
| 197.parser | C | Word Processing |
| 252.eon | C++ | Computer Visualization |
| 253.perlbmk | C | PERL programming language |
| 254.gap | C | Group theory, interpreter |
| 255.vortex | C | Object-oriented database |
| 256.bzip2 | C | Compression |
| 300.twolf | C | Place and route simulator |

**Figure 15. SPECINT2000 benchmarks**

We measured the performance gains and profiling overhead for the edge-check, naïve-loop, and naïve-all profiling methods without sampling, as well as these methods with sampling. The sampling versions will be called sample-edge-check, sample-naïve-loop, and sample-naïve-all. These methods profile only in-loop loads, except the naïve-all and sample-naïve-all methods, which profile also out-loop loads. Notice that the block-check and the edge-check method generate the same stride profile. Since our compiler only supports edge frequency profiling, we will not evaluate the block-check method.

The performance gain with the stride profile guided prefetching is measured in terms of speedup, namely the ratio of the execution time of the binaries built with feedback of only the edge frequency profile over the execution time of the binaries built with feedback of both the edge frequency and stride profiles which is used to guide stride prefetching. For example, a speedup of 1.2 indicates that the stride prefetching improves the performance by 20%.

The profiling overhead is reported as the following ratio.

$$\frac{\text{execution time with edge and stride profiling - execution time with edge profiling}}{\text{execution time with edge profiling}}$$

For example, a ratio of 0.2 indicates that the new integrated profiling method is about 20% slower than the edge profiling alone.

We also measured the sensitivity of the performance gains with respect to the input data sets.

### 4.1. Performance Gains

For this experiment, the profile generation runs use the train input data set and the performance runs with profile feedback and prefetching use the reference input data set. Figure 16 shows the speedup from prefetching guided by the stride profiles collected using the six different profiling methods.

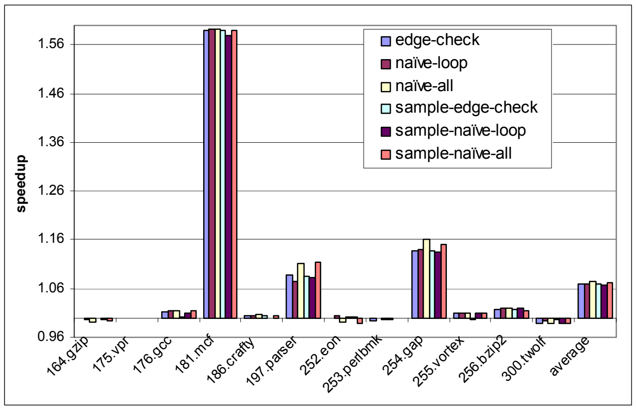

**Figure 16. Speedup of stride prefetching**

Overall, all the profiling methods result in very similar performance gain as the edge-check method: about 1.59x speedup for "181.mcf", 1.14x for "254.gap", 1.08x for "197.parser" and smaller gains in more benchmarks. These performance improvements, with an average of 7%, are significant for highly optimized SPECINT2000 programs running on real machines.

There are some minor performance differences for the naïve methods. The naïve-loop (as well as sample-naïve-loop) method performs slightly worse than the edge-check method for 197.parser (and 181.mcf). The differences are due to the slight differences in the stride profiles they collected. Specifically, the naïve-loop method will generate stride profiles for loads inside a loop whose loop nest is executed only once, while the edge-check method will not. It seems that prefetching the loads inside a loop whose loop nest is executed once will degrade performance.

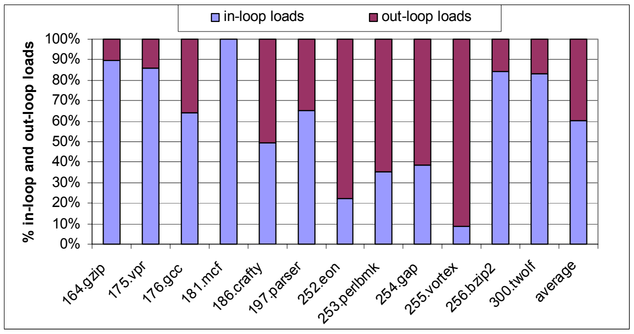

**Figure 17. Percentage of in-loop and out-loop load references**

The performance of the naïve-all (as well as sample-naïve-all) method is slightly better than the other methods mainly because it profiles and prefetches additionally the out-loop loads. It improves the speedup for 197.parser from 1.08x to 1.10x and that for 254.gap from 1.14x to 1.16x. However, the difference is small considering the large number of out-loop loads profiled. Figure 17 shows about 40% of load references (weighted by frequencies) are from out-loop loads and are profiled by the naïve-all method.

The reasons for the insignificant performance gain from prefetching out-loop loads are two folds. First, only a small portion of out-loop-loads is prefetched. Figure 18 shows the distribution of out-loop loads by stride properties, collected with the *naïve-all* method. Majority of the out-loop loads with stride properties are classified into PMST and WSST, which cannot be prefetched. Only 1.7% of the load references are from out-loop loads and is prefetched as SSST loads. In contrast, Figure 19 shows that nearly all of in-loop loads with stride patterns are prefetched as SSST and PMST. Second, even when an out-loop load is prefetched, the prefetching and using of the prefetched data may be separated by many other memory references, unlike an in-loop load where the loop body is usually small. The prefetched value could be evicted before it is used.

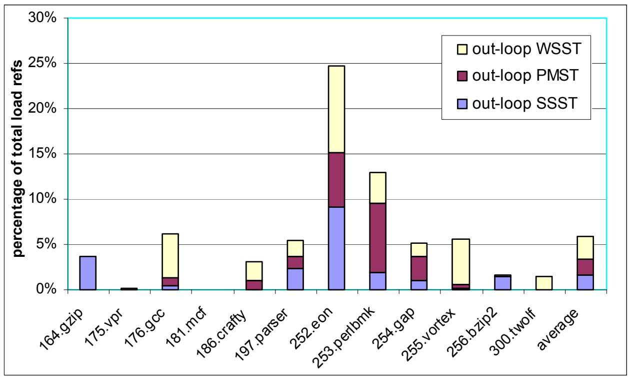

**Figure 18. Distribution of out-loop loads by stride properties**

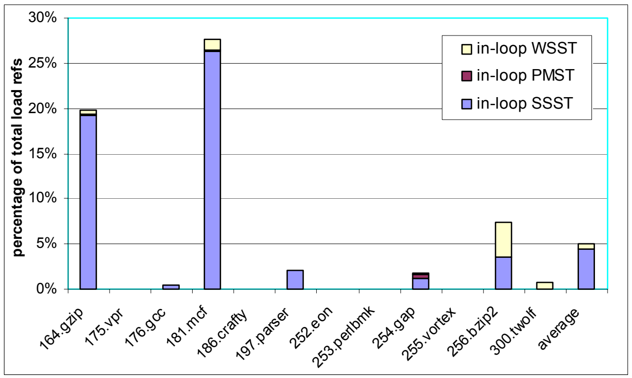

**Figure 19. Distribution of in-loop loads by stride properties**

Since all the profiling methods result in similar performance gain, we should select the profiling method basing on their easy of use and profiling overhead for production compilers. In particular, we do not need to use the two-pass profiling method that complicates the program development process, as the two-pass method prefetches the same set of loads as the naïve-loop method.

## 4.2. Profiling Overhead

For this experiment, all profile generation run uses the train input data set. Figure 20 shows the overhead of the integrated profiling methods over the edge frequency profiling alone. On the average, the edge-check method is about 58% slower than the edge profiling alone, while the naïve-loop method is 272% slower and the naïve-all method is 436% slower. The edge-check method achieves the low overhead by guarding the calls to the profiling runtime routine with the trip count condition. Figure 21 shows the percentages of load references processed by the profiling runtime routine strideProf for all the six profiling methods. Although there is about 60% of load references inside loops (see the bar marked with naive-loop in the average area), after checking the trip count condition, only about 11% load references is processed by the strideProf routine (see the bars marked with edge-check). The naïve-loop method on the other hand has to profile all in-loop loads (60%). The naïve-all method has to profile all in-loop and out-loop loads (100%).

Figure 20 also shows that sampling further reduces the overhead: the sample-edge-check method is only 17% slower than the edge profiling alone, while the sample-naïve-loop method is 67% slower and the sample-naïve-all method is 122% slower. This is because that only a portion of the total number of load references is sampled. Figure 21 shows that less than 1% load references is processed in the strideProf routine (after the sampling code) with the sample-edge-check method. The sample-naïve-loop method profiles 3% of the load references, and the sample-naïve-all method profiles about 5% of load references.

The strideProf routine is capable of collecting the number of zero strides directly and bypassing the LFU routine. This reduces the profiling overhead for all the profiling routines. Figure 22 shows that the percentage of load references that are processed by the LFU routine is significantly lower than those processed by the strideProf as shown in Figure 21. The difference between the corresponding bars on the two graphs shows the percentage of zero strides directly handled by strideProf without going through the LFU routine. For example, for the naïve-all method, 100% load references are processed by strideProf while only 68% of them are processed by the LFU routine. This indicates that about 32% of load references have zero strides.

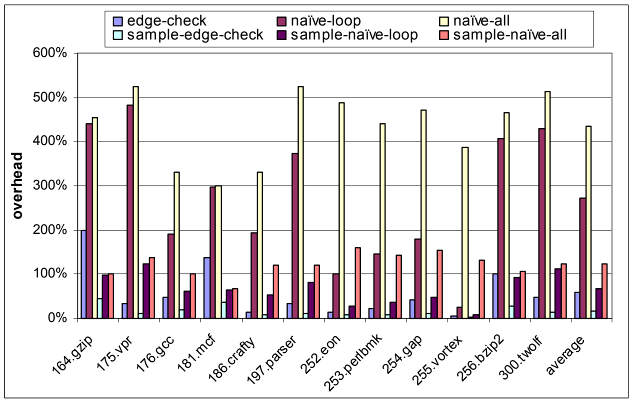

**Figure 20. Profiling overhead**

Considering both the performance gain and profiling overhead, we suggest using the sample-edge-check method in production compilers, since it has the lowest profiling overhead and similar performance to the other methods.

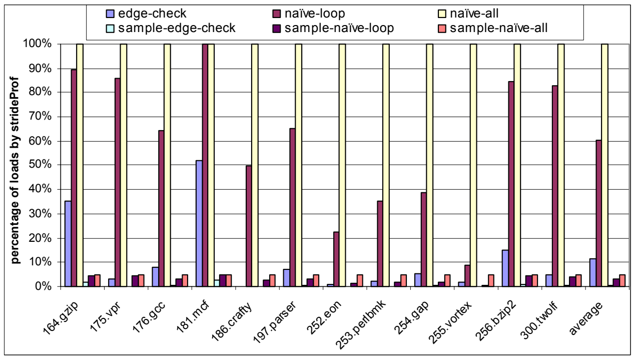

**Figure 21. Percentage of load references processed in strideProf routine (after sampling)**

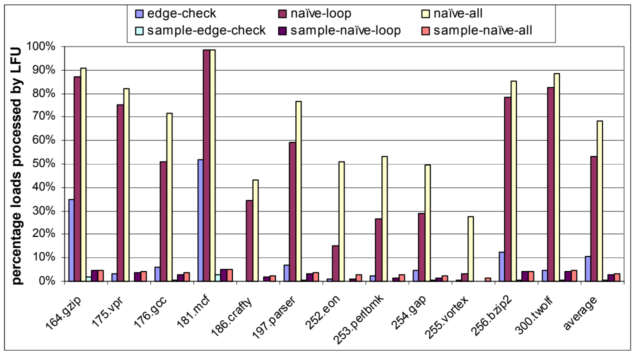

**Figure 22. Percentage of load references processed by LFU**

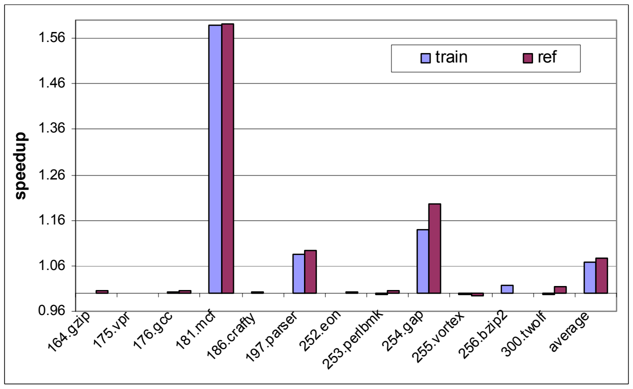

**Figure 23. Performance of train and ref**

## 4.3. Sensitivity to Input Data Sets

In this experiment, we examine the sensitivity of the performance gain to input data sets with sample-edge-check profiling and prefetching. We collect two sets of profiles, one collected from running with the train-input data set and the other with the reference-input data set. We compare the performance gains for the binaries compiled with the two profiles, both running with the reference input data set. We will refer to the binaries compiled with the profiles collected with the train input and running on the reference input as "train". Similarly, we will refer to the binaries compiled with the profiles collected with the reference input and running on the reference input as "ref".

Figure 23 shows that the performance gain of train is lower than that of the *ref*. For example, the *ref* improves over the train for 197.parser from a speedup of 1.08x to 1.09x, and for 254.gap from 1.14x to 1.20x.

Notice that the sample-edge-check profiling method collects both edge profile and stride profile. The ref not only uses stride profile collected with the reference input data set, it also uses the edge profile collected with the reference input data set. The performance improvement of ref over train could be the result of the better edge profile. To isolate the effects of stride profile and edge profile with the reference input data sets, we further introduce the following two sets of binaries. These binaries are collected with two profiling passes.

- edge.ref-stride.train: the binaries compiled with the edge profile collected with the reference input data set and the stride profile collected with the train input data set.
- edge.train-stride.ref: the binaries compiled with the edge profile collected with the train input data set and stride profile collected with reference input data set.

Figure 24 shows that the improvement of edge.ref-stride.train over the train. This improvement is similar to the improvement of ref to train. In other words, the performance improvement of ref over train is the result of the better edge profile.

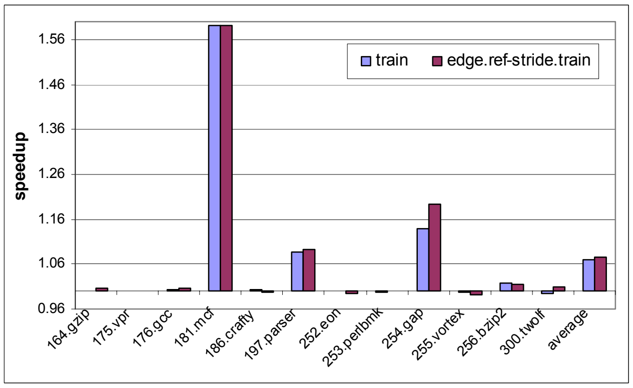

**Figure 24. Performance of train and edge.ref-stride.train**

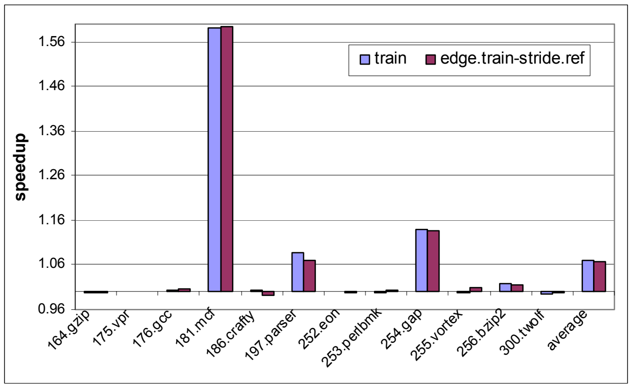

**Figure 25. Performance of train and edge.train-stride.ref**

Figure 25 further shows that the performance gain of edge.train-stride.ref is very similar to that of the train, although 197.parser shows slightly lower performance gain with edge.train-stride.ref. The reason for the lower performance gain of edge.train-stride.ref for 197.parser is that stride profile-guided prefetching uses edge profile to filter out unwanted loads. When the edge profile and the stride profile do not match (with edge profile collected from train input data set and stride profile collected from reference input data set), a few loads good for stride prefetching is unfortunately filtered out.

Comparing the results in Figure 24 and Figure 25, we can see that the sensitivity of performance gain to stride profiles collected with different input data sets is less than that to edge profiles. Since it is well accepted that edge profile is reasonably stable across input data sets [7], we conclude that stride profile is stable across input data sets as well.

## 5. RELATED WORK

There is extensive research on static compiler prefetching. The earlier work focused on prefetching array references [3][18][23]. Data reuse analysis is done to reduce the amount of redundant prefetching to the same cache line. These techniques have been incorporated in the production compiler with significant performance gains for regular programs (i.e. SPECFP2000 [10]). However, we have yet to see them bringing positive overall performance gain for the SPECINT2000 benchmarks. For example, in [27] we established that the static prefetching method results in a negative overall performance improvement. Enhancing the static prefetching with frequency profile information improve the performance noticeably, but still much lower than the stride profile guided prefetching collected with a .

Several recent studies focus on prefetching for recursive data structures. Lipasti et al [14] use heuristics to de-reference pointers passed into procedures. They analyze execution trace and inserting prefetching instructions during cache simulation. An overall improvement of 1.7% is reported. Luk and Mowry [15] examine several software techniques, including compiler-direct greedy prefetching, software full jumping and data linearization. Roth and Sohi describe a framework for jump-pointer prefetching [22]. Although these techniques have shown promising results in small benchmarks using hand-optimized code, designing compiler algorithms to automatically and beneficially perform the transformations could pose a serious challenge.

The most relevant work to ours is the compile-time stride prefetching method proposed by Stoutchinin et al [26]. It uses compiler analysis to detect induction pointers and insert instructions into user programs to compute strides and perform stride prefetching for the induction pointers. However, the compiler analysis cannot determine whether an induction pointer has stride patterns and the prefetching instructions have to be inserted conservatively, e.g. only when machine resource allows the prefetching instructions. Still, this technique can slow a program down when the stride prefetching is applied to loads without stride patterns. They showed 20% performance gain for 181.mcf, and either very small (< 1%) or negative performance gain for the remaining SPECINT2000 benchmarks.

Dedicated hardware is also proposed to prefetch for irregular references. The stream buffer based prefetching [12][20][25] uses history information to prefetch data into a stream buffer. When a load accesses the data cache, it also searches the stream buffer entries in parallel. If the data requested by the load is in the stream buffer, that data is transferred to the cache. The stride prefetching hardware [6] uses a *reference prediction table, RPT,* to monitor cache misses and record the difference between successive misses as the stride for prefetching. For a program with many loads that miss cache, the hardware tables may overflow and cause useful strides to be thrown away, and thus reduce the effectiveness of the prefetching. We believe that our software techniques can be a viable alternative to reduce hardware budget and power consumption.

Pre-computation [5] [16] [28] uses speculative threads to run portions of the program in a separate thread to prefetch for the main program. The major challenge is to identify small and precise backward slices that lead to the loads missing caches. Pointer references make the analysis difficult. All of the studies so far use hand coding to identify the backward slices.

## 6. CONCLUSIONS AND FUTURE WORK

In this paper, we present novel profiling and prefetching techniques for guiding the compiler to perform stride prefetching. We integrate the stride profiling into the traditional frequency-profiling pass and the integrated profiling pass can be only 17% slower than the frequency profiling alone. The compiler prefetching decision guided by the profile is highly selective and beneficial. We show significant performance improvement for the SPECINT2000 programs running on the Itanium machines. For example, we achieve a 1.59x speedup for "181.mcf", 1.14x for "254.gap", and 1.08x for "197.parser". These performance improvements, with an average of 7% for the entire benchmark suite, are significant for highly optimized SPECINT2000 programs running on real machines. We also demonstrate that the performance gain is stable across profiling data sets. These benefits make the new techniques suitable for production compilers.

We are currently pursuing the study in the following directions.

- The current prefetching algorithm is ineffective for out-loop loads, because the prefetch operations and the use of the prefetched data are sometimes separated by a large number of memory references. We may profile the number of memory references between the successive references at a load site. If this number is large, we should not prefetch for the load. This information can also be beneficial for in-loop loads as some loops can be very large or contain function calls.
- Extend our method to prefetch for loads without stride patterns. There are cases where a load itself does not have stride patterns, but its address depends on another load with stride patterns. We may extend our method to prefetch loads that depend on the results of the prefetching instructions.
- Investigate the possibility of using customized memory allocation to produce more strides that can be prefetched.

## 7. ACKNOWLEDGEMENTS

We would like to thank Jesse Fang, John Shen, Sun Chan, Dong-Yuan Chen, Li-Ling Chen, Roy Ju, Rakesh Krishnaiyer, Yong-Fong Lee, Hsien-Hsin Lee, Wei Li, Tin-fook Ngai, and Mauricio Serrano for their support and valuable comments. We appreciate the comments from the anonymous reviewers that helped improve the quality of the paper.

## REFERENCES

[1] Ball, T. and J. Larus, "Optimally profiling and tracing programs," ACM Transactions on Programming Languages and Systems, 16(3): 1319-1360, July 1994.

[2] Calder, B., P. Feller, and A. Eustance, "Value Profiling," MICRO30, Dec. 1997.

[3] Callahan, D., K. Kennedy, and A. Porterfield, "Software Prefetching", in Proceedings of the Fourth International Conference on Architecture Support for Programming Languages and Operating Systems, 1991, 40-52.

[4] Chang, P. P, S. Mahlke, and W.M. Hwu, "Using profile information to assist classic code optimizations," Software-practice and Experience, 1991.

[5] Collins, J., H. Wang, H. Christopher, D. Tullsen, C. J. Hughes, Y. F. Lee, D. Lavery and J. Shen, "Speculative Pre-computation: Long-range Prefetching of Delinquent Loads," ISCA28, 2001.

[6] Dahlgren, F., Stenstrom, P., "Evaluation of Hardware-Based Stride and Sequential Prefetching in Shared-Memory Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 7, No. 4, April 1996.

[7] Fisher, J. and S. Freudenberger, "Predicting Conditional Branch Directions From Previous Runs of a Program," Proc. 5th Annual Intl. Conf. on Architecture Support for Prog. Lang. and Operating Systems, Oct. 1992.

[8] Ghiya, R., D. Lavery, D. Sehr, "On the Importance of Points-to Analysis and Other Memory Disambiguation Methods for C Programs," PLDI2001, May 2001.

[9] Henning, J.L., "SPEC CPU2000: Measuring CPU Performance in the New Millennium," IEEE Computer, July 2000.

[10] Intel Corp, "Benchmarks: Intel® Itanium™ based systems," http://www.intel.com/eBusiness/products/ia64/overview/bm012101.htm.

[11] Intel Corp, Intel® Itanium™ Processor Hardware Developer's Manual, 2000. http://developer.intel.com/design/ia-64/manuals.htm.

[12] Jouppi, N., "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers," ISCA17, May 1990

[13] Krishnaiyer, R., D. Kulkarni, D. Lavery, W. Li, C. Lim, J. Ng, and D. Sehr, "An Advanced Optimizer for the IA64 Architecture, IEEE Micro, Vol 20, No 6, Nov 2000, 60-68

[14] Lipasti, M.H., W.J. Schmidt, S.R. Kunkel, and R.R. Roediger, "SPAID: Software Prefetching in Pointer and Call Intensive Environments", MICRO28, Nov 1995, 231-236.

[15] Luk, C.K. and T.C. Mowry, "Compiler-Based Prefetching for Recursive Data Structures," in Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, September 1996, 222-233.

[16] Luk, C.K., "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," ISCA28, 2001.

[17] Mahlke, S.A., D.C. Lin, W.Y. Chen, R.E. Hank, and R.A. Bringmann, "Effective Compiler Support for Predicated Execution Using Hyperblock," MICRO25, Dec. 1992.

[18] Mowry, T.C., M.S. Lam, and A. Gupta, "Design and Evaluation of a Compiler Algorithm for Prefetching," in Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, October 1992, 62-73.

[19] Muth, R., S. Watterson, S. Debray, "Code Specialization based on Value Profiling," SAS2000.

[20] Palacharla, S. and R. Kessler, "Evaluating stream buffers as secondary cache replacement," ISCA21, April 1994.

[21] Roth, A., A. Moshovos, and G. Sohi, "Dependence Based Prefetching for Linked Data Structures," Proc. 8th ASPLOS, pages 115-126. Oct. 1998.

[22] Roth, A., and G. Sohi. "Effective Jump-Pointer Prefetching for linked data structures," ISCA26, June 1999, 111-121.

[23] Santhanam, V., E. Gornish, and W. Hsu, "Data Prefetching on the HP PA-8000," ISCA24, June 1997, 264 – 273.

[24] Serrano, M. J. and Y. Wu, "Memory Performance Analysis of SPEC2000C for the Intel ItaniumTM Processor," IEEE 4th Annual Workshop on Workload Characterization, in Conjunction with MICRO34, Austin, Texas, December 2, 2001.

[25] Sherwood T., S. Sair, B. Calder, "Predictor-Directed Stream Buffers," MICRO33, Dec. 2000.

[26] Stoutchinin, A., J. N. Amaral, G. Gao, J. Dehnert, S. Jain, and A. Douillet "Speculative Prefetching of Induction Pointers," in Proceedings of CC 2001, Geneva, Italy, 2 - 6 April, 2001. Also in LNCS 2207, pp 289-303, 2001.

[27] Wu, Y., M. Serrano, R. Krishnaiyer, W. Li, J. Fang, "Value-Profile Guided Stride Prefetching for Irregular Code," in Proceedings of CC 2002, April 6 - 14, 2002, Grenoble, France.

[28] Zilles, C. and G. Sohi, "Execution-based Prediction Using Speculative Slices," ISCA28, 2001.