Value-Profile Guided Stride Prefetching for Irregular Code

Youfeng Wu[1], Mauricio Serrano[1], Rakesh Krishnaiyer[2], Wei Li[2], and Jesse Fang[1]

[1] Intel® Programming Systems Research Lab
[2] Intel® Compiler Lab
2200 Mission College Blvd, Santa Clara, CA 95052
{youfeng.wu,mauricio.serrano,rakesh.krishnaiyer,wei.li,
jesse.fang}@intel.com

Abstract: Memory operations in irregular code are difficult to prefetch, as the future address of a memory location is hard to anticipate by a compiler. However, recent studies as well as our experience indicate that many irregular programs contain loads with near-constant strides. This paper presents a novel compiler technique to profile and prefetch for those loads. The profile captures not only the dominant stride values for each profiled load, but also the differences between the successive strides of the load. The profile information helps the compiler to classify load instructions into strongly or weakly strided and single-strided or phased multi-strided. The prefetching decisions guided by the load classifications are highly selective and beneficial. We obtain significant performance improvement for the CPU2000 integer programs running on Itanium™ machines. For example, we achieve a 1.55x speedup for "181.mcf", 1.15x for "254.gap", 1.08x for "197.parser" and smaller gains in other benchmarks. We also show that the performance gain is stable across profile data sets and that the profiling overhead is low. These benefits make the new technique suitable for a production compiler.

1 Introduction

Memory operations in irregular code are difficult to prefetch, as the future address of a memory location is hard to anticipate by a compiler. An example of irregular code is the "pointer-chasing" code that manipulates dynamic data structures. However, recent studies suggest that some pointer chasing references exhibit near-constant strides. Namely, the difference between two successive data addresses changes only infrequently at runtime. Stoutchinin et al [23] and Collins et al [4] notice that several important loads in 181.mcf of CPU2000 integer benchmark suite have near-constant strides. Our experience indicates that many irregular programs, in addition to 181.mcf, contain loads with near-constant strides. For example, the *197.parser* in CPU2000 integer benchmarks has code segments as shown in Figure 1 (a). The first load chases a linked list and the second load references the string pointed to by the

R. N. Horspool (Ed.): CC 2002, LNCS 2304, pp. 307-324, 2002.

current list element. The program maintains its own memory allocation. The linked elements and the strings are allocated in the order that are referenced. Consequently, the strides for both loads remain the same 94% of the times with reference input set.

CPU2000 integer benchmark *254.gap* also contains near-constant strides in irregular code. An important loop in the benchmark performs garbage collection. A simplified version of the loop is shown in Figure 1 (b). 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 times, respectively. One of the dominant stride occurs because of 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 & ~3L)->ptr***. This access has two dominant strides, which remain constant for 48% and 47% of the times, respectively. These strides are mostly affected by the values in (**s&~3)->size* and by the allocation of the memory pointed to by *s.

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

(a) Example from 197.parser

```
while ( s < bound ) {
S1:     if ((*s & 3 == 0) {
S2:           access (*s & ~3)->ptr
S3:           s = s + ((*s & ~3)->size)+values;
              other operations;
        } else if ((*s & 3 == 2) {
S4:           s = s + *s;
        } else {
        }
}
```

(b) Example from 254.gap

Figure 1. Irregular code with dominant strides

Many other CPU2000 integer benchmarks contain loads with near-constant strides as well. To illustrate the widespread occurrence of the near-constant strides, we examine loads in the CPU2000 integer benchmarks with the following properties (in the following sections, these loads are referred to as *candidate loads*):

- Execute at least 2000 times.
- Occur inside loops with minimum trip counts of 100.
- Memory addresses are not loop invariant.

For each candidate load, we collect the top five most frequently occurring strides (including the stride value of zero), and count the number of dynamic references that have one of these strides. Figure 2 shows that, on an average, about 73% of dynamic references issued at a candidate load have one of the top five strides (see the bars marked with *Top5*). For a few benchmarks, e.g. 255.vortex, almost all the references at a candidate load have one of the top five strides. Figure 2 also shows that the set of candidate loads accounts for about 10% of the total dynamic loads (see the bars marked with *Coverage*).

Traditional compiler techniques [23][3][17][13][15][20], however, cannot easily discover the near-constant strides in irregular code. Pointer references make it hard for a compiler to estimate the stride patterns of load addresses. Also, the near-constant strides in many cases are the results of memory allocation and compiler has limited ability to analyze memory patterns. Without knowing that a load has near-constant

strides, it would be futile to insert stride prefetch instructions, as doing so will penalize those references with no regularity in strides.

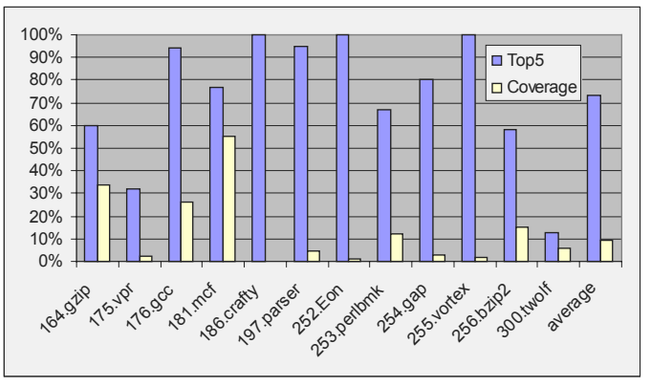

Figure 2. Near-constant strides of candidate loads

This paper presents a novel compiler technique called **V**alue-**P**rofile-**G**uided **S**tride-**P**refetching (or VPGSP for short), which uses profile feedback to determine stride values for memory references and to insert prefetching instructions for those loads that can be effectively prefetched. The compiler first identifies a set of loads and instruments the loads to collect the **S**tride **V**alue and **S**tride **D**ifference profile (or SVSD profile for short). A stride value of a load is the difference between the load addresses in adjacent iterations of the loop containing the load. The stride difference is the differences between successive stride values. The compiler then performs program analysis using the SVSD profile to determine which loads have near-constant stride and inserts prefetching instructions for these loads. The compiler may also employ code analysis whenever possible to a) determine the best prefetching distance; b) reduce the profiling cost and c) reduce the prefetching overhead.

The example in Figure 3 illustrates the new prefetching techniques. Figure 3 (a) shows a typical pointer-chasing loop. For simplicity, we assume that the load address of the reference P->data at L is **P**. The compiler instruments the load at L as shown in Figure 3 (b). The load address is passed to the profile routine to collect stride value and stride difference profile. The profile could indicate that the load at L frequently has the same stride, e.g. 60 bytes. 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, the compiler may insert prefetching instructions as shown in Figure 3 (d) to compute the runtime strides before the prefetching. The variable **prev_P** stores the load address in the previous iteration. The variable **stride** stores the difference between the **prev_P** and current load address **P**. 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 efficiently using predication [323].

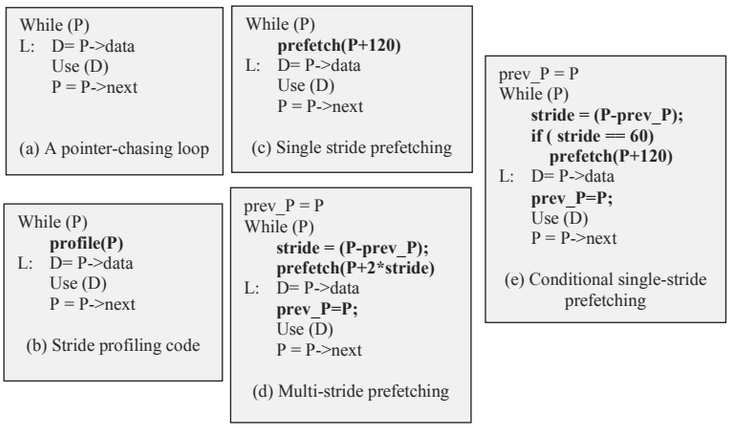

Figure 3. Example of value profiling guided stride prefetching

This paper makes the following contributions.

- A new profiling and selective stride prefetching method is presented. The profile not only captures the dominant stride values for the profiled loads but also the changing characteristics between successive strides of the loads. The profile information helps the compiler to classify load instructions into strongly or weakly strided and single-strided or phased multi-strided. The prefetching decisions guided by the load classifications are highly selective and beneficial.
- Experiments with the new prefetching method show significant performance improvement for the CPU2000 integer programs running on Itanium machines. For example, the results show a 1.55x speedup for "181.mcf", 1.15x for "254.gap", 1.08x for "197.parser" and smaller gains in other benchmarks.
- The overhead to collect the profile for stride prefetching is low. For example, there is only about 1.3x slowdown using the reference input set with instrumentation for stride profiling. This overhead is smaller than that for collecting the block/edge profiles in many production compilers [1].
- Performance comparisons show that the stride prefetching technique is competitive to a hardware-prefetching scheme. This technique seems a viable alternative to the hardware prefetch for saving the processor cost/power and at the same time achieving comparable or higher performance.
- The SVSD profile is shown to be stable across input data sets. Experiments were performed to collect the SVSD profiles using the reference-input set and the train-input set. The performance difference between binaries compiled using these two profiles running with the reference input set is small.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 describes the prefetching algorithm. Section 4 provides the experimental results. Section 5 concludes the paper and points out some future research directions.

2 Related Work

2.1 Static Compiler Prefetching

There is extensive research on static compiler prefetching. The earlier work focuses on compiler inserted prefetching instructions for array references [3][17][20]. Data reuse analysis is done to reduce the amount of redundant prefetching to the same cache line. Architecture features such as rotating registers and predication have been incorporated to data prefetching to reduce the overhead of the prefetching and branch misprediction penalty [6]. Huang et al [8] use a configurable memory controller to perform runtime data re-mapping and prefetching.

Several recent studies focus on prefetching for recursive data structures. Lipasi et al [13] use heuristics to prefetch for pointers passed into procedures. 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 [19]. Karlsson et al [12] extend the jump-pointer prefetching with prefetch arrays, or arrays of jump pointers. 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 compile-time stride prefetching method is proposed by Stoutchinin et al [23]. 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 a near-constant stride 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 stride prefetching is applied to loads without near-constant strides. Although they showed 20% performance gain for 181.mcf, they reported either very small (< 1%) or negative performance gain for the remaining CPU2000 integer benchmarks.

In Section 4.2, we will compare VPGSP with static prefetching that implements the above technique.

2.2 Hardware Prefetch

One of the well-known hardware prefetching scheme is the stream buffer based prefetching [11][7][21]. These stream buffers can be viewed as additional caches with different allocation and replacement policies from the normal caches [18]. A stream buffer is allocated when a load misses both in the data cache and in the stream buffers. The stream buffers use stride [7] or history [21] information to predict the addresses to be prefetched. When free bus cycles become available, the stream buffers prefetch cache blocks. When a load accesses the data cache, it also searches the stream buffer entries in parallel. If the data requested by the load is not in cache but is in the stream buffer, that block in the stream buffer is transferred to the cache.

Two other hardware-prefetching schemes are stride prefetching and sequential prefetching [5]. The stride-prefetching scheme works as follows. The first time a load instruction misses in a cache, the corresponding instruction address I (used as a tag)

and data address *D1* are inserted in a reference prediction table, RPT. At that time, the state is set to 'no prefetch'. Subsequently, when a new read miss is encountered with the same instruction address *I* and data address *D2,* there will be a hit in RPT, if the corresponding record has not been displaced. The stride is calculated as S1=D2-D1 and inserted in RPT, with the state set to 'prefetch'. The next time the same instruction *I* is seen with an address *D3*, a prediction of a reference to D3+S1 is done, while monitoring the current stride S2=D3-D2. If the stride S2 differs from S1, the state downgrades to 'no prefetch'.

The hardware stride prefetching has the following limitations compared to VPGSP:

- The prefetching distance is the difference of the data addresses at two misses, and it is somewhat arbitrary. This may cause either cache pollution by unnecessarily prefetching too far ahead or wasted memory traffic by prefetching too short.
- The hardware table is limited in size. For a program with many loads that miss cache, the table may overflow and cause some of the useful strides to be thrown away, and thus reduce the effectiveness of the prefetching.
- The hardware monitors cache misses at a particular cache level, e.g. L1, to determine prefetch strides. VPGSP is more flexible as it can prefetch for different cache levels by using different prefetching distances.

We will compare the performance of VPGSP with that of the hardware stride prefetching in Section 4.3.

2.3 Pre-computation

Recently, a number of studies propose to use speculative threads to run portions of the program in a separate thread to prefetch for the main program. For example, Collins et al [4] identifies delinquent loads and the backward slices for the loads during a previous simulation run. The profile is then used in the later run of the program to issue the slices speculatively before the delinquent loads are executed. C. Zilles and G. Sohi [25] use a runtime mechanism to predict which backward slices to run-ahead. C. Luk [14] uses compiler to insert code in user program to fork speculative threads.

With multithreaded resource, pre-computation can perform more complex address computation than stride calculation. However, all the pre-computation approaches so far resort to hand coding in program transformation and the performance gains are measured on simulators. Some of them report performance gain only for 181.mcf out of all the CPU2000 integer benchmarks. Our approach does not require speculative multithreaded hardware and the compiler performs stride prefetching automatically for the entire CPU2000 integer benchmark suite, with significant performance improvement on a real machine.

2.4 Value Profiling

A value profiling technique was proposed in [2]. It instruments user programs to collect the frequencies of the recurrent values for each profiling candidate. It uses a Least Frequently Used (LFU) replacement algorithm to manage a buffer to keep track

of the most frequently recurrent values. We extend it to profile both stride value and stride difference. We also device a few techniques to improve the speed of the value profiler for collecting stride value and stride difference profiles.

3 Prefetching Algorithm

Here are some terms that are used in the description of the prefetching algorithm:

- *Candidate load*: a load that may miss cache and should be considered for profiling or prefetching.
- *Profiled load*: a candidate load that is selected for profiling.
- *Prefetched load*: a candidate load that is selected for prefetching.

The prefetching algorithm is described in the following subsections.

3.1 Identify Candidate Loads

The compiler identifies a candidate load for stride profiling with the following criteria using control flow profile information.

- The load is frequently executed.
- The load is inside a loop.
- The loop has a high trip count. For a loop with a very low trip count (e.g. 1), the compiler will consider the trip count of its parent loop, and the loads inside the loop will be prefetched as if they are in the parent loop.
- The memory address of the load is not a loop invariant.

The above trip count condition indicates that the load is likely to touch a large range of memory. For example, the range [x, x+stride * trip_count] of memory area will be touched. Therefore, the candidate loads selected by the above criteria are likely to miss cache, especially when SVSD profile shows that the stride value is large.

3.2 Select Profiled Loads and Collect Profile

A set of loads is equivalent if 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. Examples of equivalent loads are:

- Loads that access different fields of the same data structure.
- Loads that access different elements of the same array.

For each profiled load, the compiler inserts profiling instructions to collect SVSD profile. When the instrumented program is run, profile runtime routine collects two types of information for the given series of addresses from a profiled load: stride value profile and stride difference profile.

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

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

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

The value-profiling algorithm reported in [2] is used to collect the stride value profile. The same algorithm could also be used to collect the stride difference profile. However, we can simply count the number of zero differences between successive strides to obtain the stride difference profile. If the percentage of the zero differences is high, we know that the stride sequence is phased.

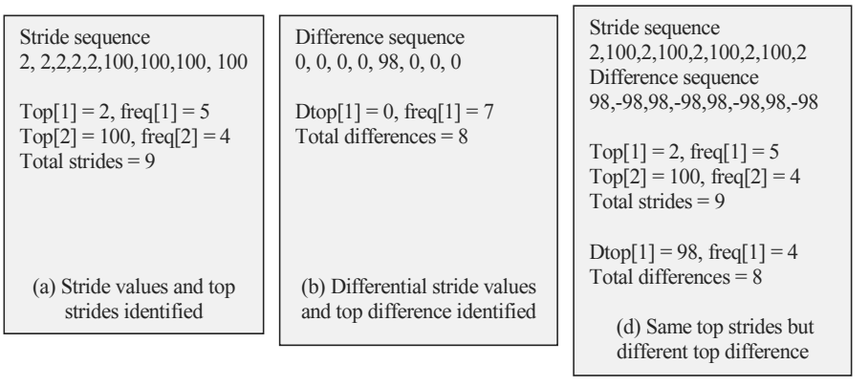

Figure 4. Stride value profiling and stride difference profiling example

Stride prefetching often remains effective when the stride value changes slightly. For example, prefetching at address+24 and the prefetch at address+30 should not have much performance difference, if the cache line is large enough to accommodate the data at both addresses. To consider this effect, the "profile (address)" routine considers the strides that are different by less than half the size of a cache line to be identical.

3.3 Analyze Profile

The compiler reads the SVSD profiles to guide its prefetching decision. It identifies the loads for stride prefetching by classifying the profiled loads into the following categories:

- Strong single stride load: A load with one non-zero stride that occurs with a very high probability (e.g. at least 70% of the time).
- Phased multi-stride load: A load with multiple non-zero strides that together occur the majority of the times and the stride differences are mostly zeroes. For example, the profile may find out the stride values 32, 60, 1024 together occur more than 60% of the time and 50% of the stride differences are zeros.
- Weak single stride load: A load where only one of the non-zero strides is frequent (e.g. > 30% of the time) and where stride differences are often zeros. For example, the profile may find out that the stride for a load has a non-zero stride 35% of time and the stride differences are zeroes 10% of the time.

In the first case, the compiler simply uses the most likely stride obtained from profile in the prefetching instructions. In the second case, it uses run-time calculation to determine the strides. In the third case, it uses conditional prefetching.

3.4 Insert Stride Prefetching Instructions

For each set of equivalent candidate loads, although only one of them is profiled, more than one may need to be prefetched. To decide which ones to prefetch, the compiler analyzes the range of cache area accessed by the loads. Enough loads will be prefetched to cover the cache lines in that range. The loads selected for prefetching are called the *prefetched loads*.

Assume a prefetched load has a load address P in the current loop iteration.

If this is a *strong single stride* load and the dominant stride value is S, the compiler inserts the prefetch instruction "*prefetch (P+K*S)*" right before the load instruction, where K*S is a compile-time constant. The constant K is the prefetch distance determined by compiler analysis. If the load may have a miss latency of W cycles, and the loop body takes about B cycles without taking the miss latency of prefetched loads into account, then K = !W/B∀

The cache miss latency can be estimated based on the analysis of the working set size of the loop. For example, if the estimated data working set size (trip_count * stride) of the load is larger than the size of L3 cache (the highest cache level), the L3 cache miss latency will be used as the value for W. The value of K may also be determined using loop trip count value as follows:

K = min (!trip_count / T∀, C)

where T is the trip count threshold (e.g. 100), and C is the maximum prefetch distance (e.g. 8).

If this is a *phased multi-stride* load, the compiler inserts the following instructions:

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 "prefetch (P+K**stride*)" 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* load, the compiler inserts prefetching instructions the same as the steps 1 and 2 described for a phased multi-stride load. Step 3 is modified as follows:

3' Insert a conditional prefetching "if (stride == profiled stride) prefetch (P+K*stride)" before the load. The conditional prefetch instruction can be implemented in Itanium using predication. For example, the compiler can compute a predicate "p = stride == profiled stride" and insert a predicated prefetch instruction "p? prefetch (P+K*stride)". The reason for a conditional instruction is to reduce the number of useless prefetches for a weak single stride load.

4 Experimentation

VPGSP is implemented in a research compiler for the Itanium Processor Family (IPF). The compiler automatically performs the profiling and prefetching without any hand coding involved. The compiler is based on a production compiler with additional components to make compiler and architectural exploration easier. The code it produced has a similar base performance as reported in [9].

The experiments use the integer benchmark suite in the CPU2000 (see Figure 5) running on the Itanium machines [10]. The Itanium machines used in these experiments have a 16K 4-way set associative split L1 data cache, a 96K 6-way set associative unified L2 cache, and 2M 4-way set associative unified L3 cache. Some of the experiments also use an Itanium machine with 4M L3 cache.

In the experiments, the compiler uses the following criteria to classify loads for prefetching.

Strong single stride loads: loads with one of the non-zero strides occurring at least 70% of the time.

Phased multi-stride loads: loads with the top 4 non-zero strides occurring at least 30% of the time and at least 30% of the differences between non-zero strides are zeroes.

Weak single stride loads: loads with one of the non-zero strides occurring at least 20% of the time and at least 10% of the differences between strides are zeroes.

In this section, we first measure the speedup of VPGSP compared to 1) no prefetching, 2) static compiler prefetching and 3) hardware prefetching. Then, we report experimental results on the profiling overhead and the sensitivity of the profile to input data sets.

ID	Programs	Language	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 5. CPU2000 integer benchmarks

4.1 Comparison with No Prefetching

Figure 6 shows the performance comparison of VPGSP vs. no prefetching. For a 2M L3 machine, the 181.mcf benchmark is sped up by 1.55x, the 254.gap benchmark is sped up by 1.15x. For the 4M L3 machine, the 181.mcf benchmark is sped up by 1.43x, the 254.gap benchmark is sped up by 1.15x. Several benchmarks (e.g. 181.mcf, 197.parser, and 300.vortex) miss L3 frequently and increasing L3 size from 2M to 4M reduces the effectiveness of the prefetching.

Figure 6 also shows that a few benchmarks, such as 164.gzip, have slight performance loss with VPGSP. The prefetching instructions inserted by the compiler may increase the schedule length if the instruction scheduler cannot find enough available slots to place the instructions. The schedule length measures the time to execute a program without considering the cache miss and other microarchitecture stalls. Figure 7 shows that, on the average, about 3.6% more instructions are executed for prefetching, and the prefetching instructions increase the schedule length by about 0.6%. If the prefetching instructions inserted by the compiler in a program do not reduce cache misses enough to compensate for the prefetching overhead (i.e. the increase in schedule length and additional memory traffic), the program will show a performance loss.

4.2 Comparison with Static Prefetching

This experiment compares VPGSP with a start-of-the-art static prefetching implemented in a production compiler for IPF. The production compiler includes implementation of a number of prefetching techniques for array and irregular code [23][3][17][6][13][15][20]. Specifically, it includes the technique proposed by Stoutchinin et al [23] to detect induction pointers for stride prefetching. The compiler prefetching is very effective for numeric code [9][22]. For irregular code, however, its performance is mixed. Figure 8 shows the results on an Itanium machine with 4M L3 cache. The bars marked with "*static prefetchingî* show the performance of the static prefetching. Some benchmarks have significant performance gains (181.mcf 9%, 254.gap 7%) and others have noticeable losses (197.parser 10%, 300.twolf 8%). The geomean is slightly negative.

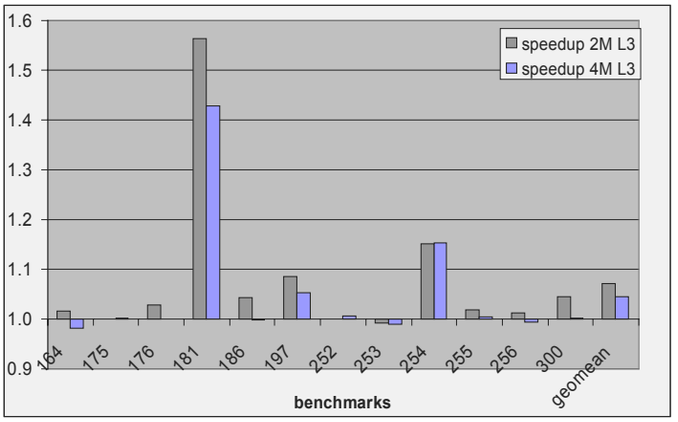

Figure 6. Performance comparison with no prefetching

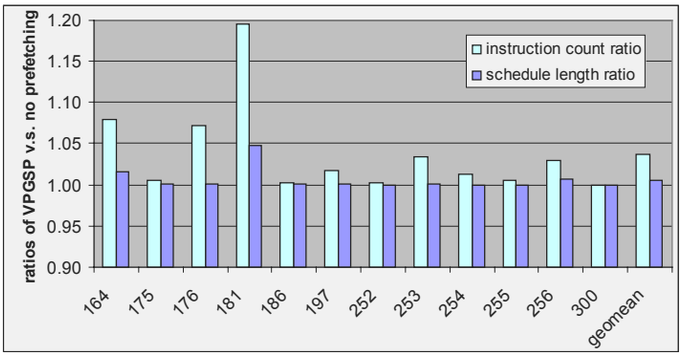

Figure 7. VPGSP overhead in terms of instruction count and schedule length

Notice that VPGSP uses both control flow profile (block frequency and trip count) and SVSD profile to make prefetching decisions. The control flow profile could also benefit the static prefetching. This experiment further enhances the static prefetching with control flow profile information, namely to prefetch only loads that are frequently executed (e.g. at least 2000 times) and inside a loop with high trip count (e.g. the trip count is at least 100). This leads to a significantly higher performance. The result for this experiment is shown with the bars marked by "*static+cflow prefetching*î in Figure 8. Still, the performance of the enhanced prefetching lags behind that of VPGSP (see the bar marked with VPGSP).

4.3 Comparison with Hardware Prefetching

To compare with hardware prefetching, a cache simulator is used to measure the miss rate reduction by VGPSP and by the hardware stride prefetching mechanism [5], on the same memory configuration as an Itanium machine with 4M L3 cache.

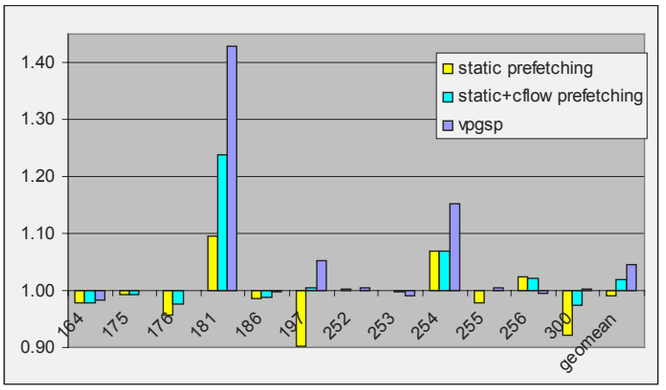

Figure 8. Performance comparison with static prefetching

In this experiment, two sets of binaries are generated, one with and another without VPGSP prefetching. For the binaries with prefetching, the simulator treats the prefetch instructions inserted by the compiler as load references. For the binaries without prefetching, the simulator performs the hardware stride prefetch function. The hardware stride prefetch monitors L1 misses and uses a 256-entry direct-mapped reference prediction table (as suggested in [5]).

We compare the miss rates for L1/L2/L3 data caches. Figure 9 shows the miss rate reductions for the CPU2000 integer benchmarks with VPGSP and the hardware scheme. VPGSP reduces the cache miss rates much more significantly than the hardware scheme. For example, VPGSP reduces the L1, L2, and L3 miss rates for 181.mcf by 59%, 57%, 23%, respectively, while the hardware scheme reduces miss rates by 23%, 24%, and 3%. On the average, VPGSP reduces the L1, L2, and L3 miss rates for CPU2000 integer benchmarks by 16%, 16%, and 15% respectively, while the hardware scheme reduces L1 miss rates by 12%, 10%, and 10%.

There are a few cases for which the hardware stride prefetching reduces miss rates slightly more than VPGSP does, e.g. 164.gzip (L2), 176.gcc (L3), and 255.vortex (L1 and L2). We suspect that VPGSP and the hardware scheme might be prefetching slightly different sets of loads. VPGSP may support more aggressive program-specific prefetching using profile information. The hardware dynamic logic may capture transient behavior for code not inside a loop or in a loop with low trip counts. We will investigate a possible integration of the two in the future. For example, we may pass hints about the loads that cannot be effectively prefetched by compiler to the hardware.

4.4 Sensitivity to Profiling Data Sets

This experiment shows that the performance improvement of VPGSP is stable across input data sets. In this experiment, SVSD profiles with both the reference-input set and the train-input set are collected. The performance difference is measured between the binaries compiled using the two profiles running with the reference input set. Figure 10 shows that the performance gain with the profile obtained using the train

input set (train-ref) is only slightly lower than that obtained using the reference input set (ref-ref).

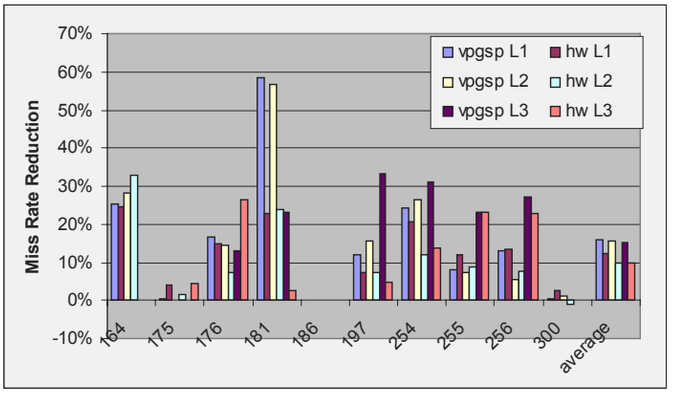

Figure 9. Miss rates reduction by hardware stride prefetching and by VPGSP

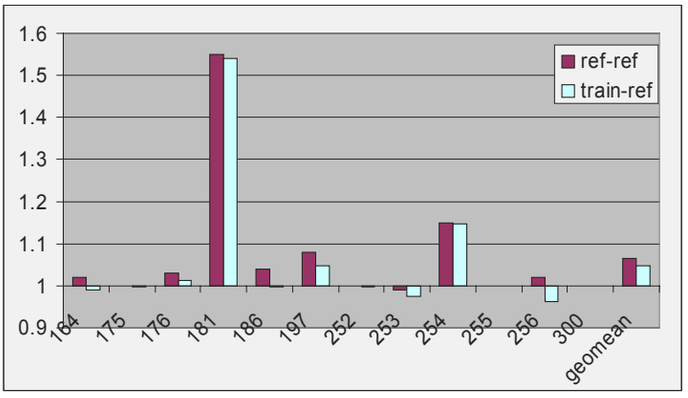

Figure 10. Speedup with profiles from reference and train input sets

4.5 Classifications of Profiled Loads

This experiment provides the distribution of profiled loads into the following categories:

- SSST - Strong single stride loads
- PMST - Phased multi-stride loads
- WSST - Weak single stride loads

Figure 11 shows that on the average, about 59% of profiled loads are “strong single stride”, about 5% are “phased multi-stride” and another 2% are “weak single stride”. Individual benchmarks show significantly different distributions. For example, the 254.gap benchmark has 45% profiled loads in the “phased multi-stride” category,

36% in the "weak single stride" category and only 17% in the "strong single stride" category.

As discussed in Section 3.4, each class of loads has a different prefetching code sequence. Benchmark like 254.gap needs all three types of prefetching code sequences to achieve the highest performance improvement.

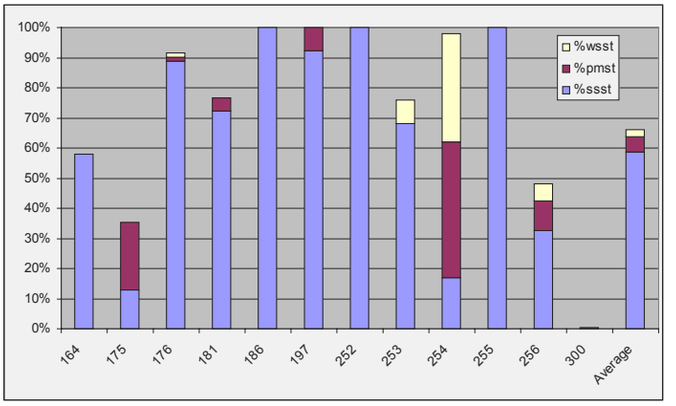

Figure 11. Distribution of profiled loads

4.6 Profiling Overhead

Finally, the execution time of the programs with instrumentation for collecting SVSD profile (profile_run) is compared with those of the programs without the instrumentation (base_run). Figure 12 shows the ratios of the execution time of the profile_run over the base_run with the same reference input set. The profiling_run is only about 1.3x slower than the base_run.

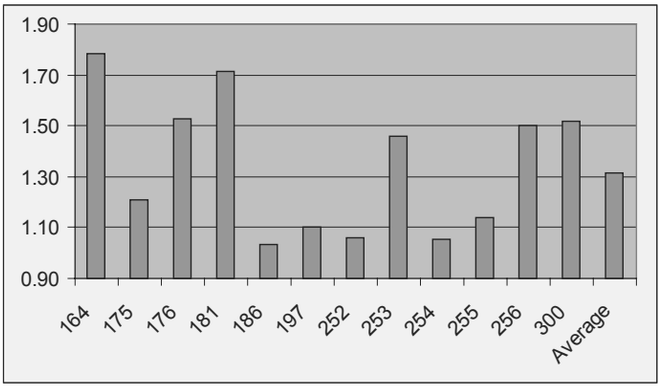

Figure 12. Profiling overhead

We achieve this low profiling overhead by selecting only a small percentage of the loads for profiling (about 10% of dynamic loads are profiled). We have also

employed a few techniques in the SVSD profiler to reduce the profiling overhead. Firstly, the value-profiling algorithm [2] invokes the heavy-duty LFU operation (Least Frequently Used replacement) whenever the profile value changes. We treat strides that are different by half a cache line size or less as the same and this reduces profiling overhead. Secondly, the value-profiling overhead is in proportion to the number of top values tracked by the LFU operation. In our implementation, only four non-zero top strides are tracked by the LFU operation. The zero strides are checked and counted without going through the LFU operation.

5 Conclusions and Future Work

In this paper, we have presented a novel profiling and prefetching method for guiding the compiler to perform selective stride prefetching. The profile not only captures the dominant stride values for the profiled loads, but also the changing rate between successive strides. The profile information helps the compiler to classify load instructions into strongly or weakly strided and single-strided or phased multi-strided. The prefetching decisions guided by the load classifications are highly selective and beneficial. We show significant performance improvement from VPGSP for the CPU2000 integer programs running on the Itanium machines. For example, we observe a 1.55x speedup for "181.mcf", 1.15x for "254.gap", and 1.08x for "197.parser". The performance gain from VPGSP is much higher than that obtained by the static compiler prefetching as well as the hardware prefetching methods. Furthermore, the performance gain is stable across profiling data sets and the profile overhead is quite low. These benefits make the new technique suitable for a production compiler.

We are currently pursuing the study in the following directions.

- The SVSD profile is collected in a separate pass from the control flow profiling pass. We are investigating ways to implement the SVSD profiling algorithm so it can be run in the same pass as the control flow profiling to simplify the software development cycle.
- We have observed cases where a load itself does not have a dominant stride, but its address depends on another load with constant strides. We may extend VPGSP to prefetch loads that depend on the results of the stride prefetching.
- Many applications maintain their own memory allocation and we need to investigate techniques to teach the customized memory allocation to produce more strides that are constant.
- A hybrid approach of software and hardware prefetching may lead to better performance. As described in Section 2.2, we may try to pass SVSD profile information to hardware to so the hardware will only prefetch for the loads that are not prefetched by software.
- The effectiveness of compiler prefetching can be improved through feedback of cache simulation information.

Acknowledgements

We would like to thank John Shen, Sun Chan, Dong-Yuan Chen, Yong-Fong Lee, and Hsien-Hsin Lee for their 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”, ASPLOS4, 1991, 40-52.
[4] 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.
[5] 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.
[6] Doshi, G., R. Krishnaiyer, and K. Muthukumar, “Optimizing Software Data Prefetches with Rotating Registers”, PACT 2001.
[7] Farkas, K., P. Chow, N. Jouppi, and Z. Vranesic, "Memory-system design considerations for dynamically-scheduled processors," ISCA24, June 1997.
[8] Huang, Xianglong, Zhenlin Wang, and K.S. McKinley, “Compiling for the Impulse memory controller,” PACT2001. Pages: 141 –150
[9] Intel Corp, "Benchmarks: Intel® Itanium™ based systems," http://www.intel.com/eBusiness/products/ia64/overview/bm012101.htm.
[10] Intel Corp, Intel® Itanium™ Processor Hardware Developer’s Manual, 2000. http://developer.intel.com/design/ia-64/manuals.htm.
[11] Jouppi, N., "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers," ISCA17, May 1990
[12] Karlsson, M., F. Dahlgren, and P. Stenstrom, “A Prefetching Technique for Irregular Accesses to Linked Data Structures,” HPCA6, January. 2000
[13] 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.
[14] Luk, C., "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors,” ISCA28, 2001.
[15] Luk, C.K. and T.C. Mowry, “Compiler-Based Prefetching for Recursive Data Structures,” ASPLOS7, September 1996, 222-233.
[16] 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, pp 45-54.

[17] Mowry, T.C., M.S. Lam, and A. Gupta, “Design and Evaluation of a Compiler Algorithm for Prefetching,” ASPLOS5, October 1992, 62-73.
[18] Palacharla, S. and R. Kessler, "Evaluating stream buffers as secondary cache replacement," ISCA21, April 1994.
[19] Roth, A., and G. Sohi. “Effective Jump-Pointer Prefetching for linked data structures,” ISCA26, June 1999, 111-121.
[20] Santhanam, V., E. Gornish, and W. Hsu, “Data Prefetching on the HP PA-8000," ISCA24, June 1997, 264 – 273.
[21] Sherwood, T., S. Sair, B. Calder, "Predictor-Directed Stream Buffers," MICRO33, Dec. 2000.
[22] Standard Performance Evaluation Corporation, “All SPEC CFP2000 Results Published by SPEC,” http://www.spec.org/osg/cpu2000/results/res2001q2/cpu2000-20010522-00663.html, 2001.
[23] Stoutchinin, A., J. N. Amaral, G. Gao, J. Dehnert, S. Jain, and A. Douillet “Speculative Prefetching of Induction Pointers,” CC 2001, April, 2001. Also in LNCS 2207, pp 289-303, 2001.
[24] Wiel, V., S.P., Lilja, D.J. “When caches aren't enough: data prefetching techniques,” Computer, Volume: 30 Issue: 7, July 1997, Page(s): 23 -30
[25] Zilles, C. and G. Sohi, "Execution-based Prediction Using Speculative Slices," ISCA28, 2001.