Stride Prefetching by Dynamically Inspecting Objects

Tatsushi Inagaki Tamiya Onodera Hideaki Komatsu Toshio Nakatani

IBM Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan
{e29253,tonodera,komatsu,nakatani}@jp.ibm.com

ABSTRACT

Software prefetching is a promising technique to hide cache miss latencies, but it remains challenging to effectively prefetch pointer-based data structures because obtaining the memory address to be prefetched requires pointer dereferences. The recently proposed stride prefetching overcomes this problem, but it only exploits *inter-iteration* stride patterns and relies on an off-line profiling method.

We propose a new algorithm for stride prefetching which is intended for use in a dynamic compiler. We exploit both *inter-* and *intra-iteration* stride patterns, which we discover using an ultra-lightweight profiling technique, called *object inspection*. This is a kind of partial interpretation that only a dynamic compiler can perform. During the compilation of a method, the dynamic compiler gathers the profile information by partially interpreting the method using the actual values of parameters and causing no side effects.

We evaluated an implementation of our prefetching algorithm in a production-level Java just-in-time compiler. The results show that the algorithm achieved up to an 18.9% and 25.1% speedup in industry-standard benchmarks on the Pentium 4 and the Athlon MP, respectively, while it increased the compilation time by less than 3.0%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, memory management, optimization*

General Terms

Algorithm, Design, Experimentation, Performance

Keywords

Java just-in-time compiler, object inspection, stride prefetching

1. INTRODUCTION

The performance gap between the processor and memory continues to widen with no indication of yet reaching a limit. Cache memories at several levels attempt to ameliorate the problem, exploiting temporal and spatial locality of program execution. They are quite effective in many programs for reducing memory access latencies and thus improving performance.

However, cache memories do not help very much for programs which access a large amount of data and suffer from a significant number of cache misses. While classical instances of such programs are numerical applications accessing large data structures such as vectors and matrices, modern object-oriented programming also tends to result in such programs by creating a large number of objects in the heap and chasing references among the objects. It is not uncommon for Java [7] programs to create millions of objects in a heap that might be hundreds of megabytes in size [5].

Software prefetching is one of the promising techniques to address the issue [3]. Assuming a special prefetch instruction exists for moving data into a higher-level cache, it attempts to hide cache miss latencies by issuing a prefetch instruction for the data well before the data is accessed. However, improving performance with software prefetching is not a trivial task. First, the timing of issuing the prefetch instruction is tricky. It must not be issued too late, or the prefetched data may still not have become available when the processor executes the memory operations using that data. On the other hand, it must not be issued too early, or the cache may no longer contain the prefetched data when the processor executes the operations. Second, the prefetch instruction must be issued only when memory bandwidth is not being fully used, since executing a prefetch instruction is not free. Finally, the overhead of computing the address of the data to be prefetched should be small. In particular, the number of memory operations executed for obtaining the address must be minimized.

While many algorithms have been successfully developed to prefetch *array-based* data references in numerical applications [16], it is more challenging to effectively prefetch *pointer-based* data references because of the much larger overhead to obtain the target address. Consider a loop that iterates over the elements of a list. If we prefetch in the i-th iteration the element accessed in the $(i+c)$-th iteration, we must make c pointer dereferences to obtain the address of the element for the prefetch instruction.

Recently, Wu [23] and Wu et al. [24] presented a new prefetching algorithm that can uniformly handle both array-based and pointer-based references. It is based on the observation that important load instructions could exhibit stride patterns even when they reference pointer-based data structures. Consider again a loop traversing a list. If the program constructs the list by allocating and appending equal-sized elements without other intervening allocations, the load instruction for retrieving the next element in the loop probably has constant strides.

Wu developed an off-line profiling method to efficiently discover load instructions with stride patterns, and used the obtained stride profiles to guide compiler prefetching. Wu's stride prefetching

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, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'03, June 9–11, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-662-5/03/0006 ...$5.00.

yields a significant speedup because stride patterns allow prefetching without dereferences.

While Wu handles both in-loop and out-of-loop loads, his experimental results show that the performance gain from prefetching out-of-loop loads was insignificant. Moreover, Wu attempted to discover only what we call *inter-iteration* stride patterns for in-loop loads. That is, given a load instruction in a loop, Wu attempted to determine whether or not the sequence of the addresses accessed by the instruction over iterations exhibits a stride pattern.

We propose a new approach to stride prefetching which is intended for use in a dynamic compiler. Focusing on in-loop loads, we attempt to discover and exploit not only inter-iteration patterns but also *intra-iteration* patterns. Given a pair of load instructions in a loop, we define the stride between them as the difference between the addresses accessed by the two instructions within one iteration, and two load instructions are said to show an intra-iteration stride pattern when the sequence of the strides shows a pattern between iterations. As we show later, pairs of important loads without inter-iteration stride patterns could exhibit intra-iteration stride patterns, allowing our approach to yield more performance gains.

We discover inter-iteration and intra-iteration stride patterns with an ultra-lightweight technique for dynamic profiling called *object inspection*. This is a kind of partial interpretation that only a just-in-time (JIT) compiler is able to perform. When invoked for a method containing one or more loops, the JIT compiler partially interprets the method with the *actual* values of the method's parameters and without generating any side effects, executing each loop a small number of times to discover the stride patterns.

We build a dependence graph of load instructions to limit the number of candidate pairs of load instructions that must be considered for intra-iteration stride patterns. Load instructions for chasing references are likely to show intra-iteration stride patterns, since constructors in object-oriented languages tend to allocate a bunch of related objects. We build the dependence graph according to reference-chasing sequences within a loop, resulting in candidate pairs of load instructions being represented as adjacent nodes in the graph.

We evaluated an implementation of our prefetching algorithm in the JIT compiler [22] of IBM Developer Kit for Windows, Java Technology Edition. We ran the SPECjvm98 benchmark [20] and Section 3 of the JavaGrande v2.0 benchmark [12] on two machines with different IA-32 architectures, an Intel Pentium 4 [9] and an AMD Athlon MP [1]. The results show that our stride prefetching achieved up to an 18.9% and 25.1% speedup on the Pentium 4 and the Athlon MP, respectively.

Our contributions in this paper are as follows.

- *Discovery and exploitation of intra-iteration stride patterns.* To the best of our knowledge, this is the first attempt to discover intra-iteration stride patterns and utilize them for software prefetching. Given a pair of load instructions in a loop, the two instructions are said to show an intra-iteration stride pattern, if the sequence of the strides between them shows a pattern over iterations. We first build a dependence graph of load instructions to limit the number of candidate pairs of load instructions, and attempt to discover intra-iteration stride patterns using object inspection.

- *Object inspection.* This is an ultra-lightweight technique for dynamic profiling which only a dynamic compiler is able to use. When dynamically compiling a method, it gathers an "execution profile" of the method by partially interpreting the method at compile time, using the *actual* values of the parameters and causing no side effects. To the best of our knowledge, this is the first application of such an ultra-lightweight profiling technique for dynamic optimizations. More concretely, we use the novel profiling technique for discovering both inter-iteration and intra-iteration stride patterns.

- *Evaluation on a production Java JIT compiler.* We implemented our stride prefetching algorithm on a production Java virtual machine and JIT compiler, and evaluated the effectiveness on two different IA-32 machines using industry standard benchmarks.

```
class TokenVector {
 Token[] v;
 int ptr;
 void addElement (Token val) {...}
 void removeElement (Token val) {...}
 ...
}

class Token {
 ValueVector[] facts;
 int size = 0;
 Token (ValueVector firstFact) {
  facts = new ValueVector[5];
  facts[size++] = firstFact;
 }
 ...
}

class Node2 {
 Token findInMemory (TokenVector tv, Token t) {
 TokenLoop:
  for (int i = 0; i < tv.ptr; i++) {
   Token tmp = tv.v[i];
   for (int j = 0; j < t.size; j++)
    if (!t.facts[j].equals (tmp.facts[j]))
     continue TokenLoop;
   return tmp;
  }
  return null;
 }
 ...
}
```

Figure 1: Simplified code fragments from _202_jess.

The rest of this paper is organized as follows. Section 2 shows a motivating example to discuss stride prefetching and give an overview of our approach. Section 3 describes our stride prefetching algorithm. Section 4 presents performance results, while Section 5 discusses the related work. Finally, Section 6 offers conclusions.

2. A MOTIVATING EXAMPLE

We discuss stride prefetching using the motivating example shown in Figure 1. The code is taken from the *_202_jess* benchmark in SPECjvm98. The `findInMemory()` method is one of the time-consuming methods in *_202_jess*, spending most of the time executing the doubly nested loop. Also, the execution profile shows that, while the outer loop has a large trip count, the inner loop has a quite small trip count. Figure 2 summarizes the data structures accessed in the method, representing as solid arrows the pointer references chased in the method.

In-loop loads are major targets of software prefetching. The doubly nested loop contains the eleven load instructions listed in Table 1. The load instructions for the `length` fields of the arrays are

Table 1: Load instructions in the `findInMemory()` method.

Load instructions	Memory addresses
L_1	`&tv.ptr`
L_2	`&tv.v`
L_3	`&tv.v.length`
L_4	`&tv.v[i]`
L_5	`&t.size`
L_6	`&t.facts`
L_7	`&t.facts.length`
L_8	`&t.facts[j]`
L_9	`&tmp.facts`
L_{10}	`&tmp.facts.length`
L_{11}	`&tmp.facts[j]`

not explicit in the Java source program, but are generated for array bound checks.

We now explain how Wu et al. [24] discover and exploit inter-iteration stride patterns for in-loop loads. Their approach is based on off-line profiling. They select candidate loads using the execution frequency profile and compiler's static analysis, instrument the code to collect stride profiles for the candidate loads, and generate prefetch instructions based on the stride profiles obtained.

They select a load in a loop as a candidate for stride profiling using the following criteria:

- The loop has a high trip count.
- The load is frequently executed.
- The memory address of the load is not a loop invariant.

A load in a loop with a high trip count is selected since the load is likely to touch a large range of memory and to miss the cache. When a nested loop does not have a high trip count but the parent loop does, each load in the nested loop is considered as if it were in the parent loop.

When applied to the `findInMemory()` method, their algorithm selects L_4, L_9, L_{10}, and L_{11} to collect stride profiles, and the rest of the loads are all loop-invariants. However, the resulting stride profiles show that only L_4 has a stride pattern. This is because the array of `Token` objects referenced by `tv.v` is not constructed in the initialization phase. The `Token` objects are appended to and removed from the array during the execution of the *_202_jess* benchmark. Furthermore, when the `removeElement()` method attempts to remove a `Token` object and finds it stored as the array's *i*-th element, it removes the object by moving to the index `i` the array's last element. As a result, by only discovering a stride pattern for L_4, their algorithm generates a prefetch instruction as in Figure 3, which is intended for prefetching data *c* iterations ahead, when L_4 has *d* bytes of an inter-iteration constant stride.

We extend their approach in two ways to capture more loads as targets for prefetching. First, although L_9 does not exhibit an inter-iteration stride pattern, we could prefetch the data which L_9 is likely to load after *c* iterations, since L_9 is data dependent upon L_4 and L_4 shows an inter-iteration stride pattern. That is, we could *speculatively* execute L_4 using the memory address predicted based on the stride pattern, and prefetch the data for L_9 using the result. This requires one pointer dereference, but the benefits may offset the cost.

Second, while none of L_9, L_{10}, and L_{11} is likely to show an inter-iteration stride pattern, (L_9, L_{10}) is likely to exhibit an intra-iteration stride pattern. This is because the constructor for the `Token` class allocates an array and stores a reference to the array into the `facts` field, not reassigning a new reference into the field after that. When

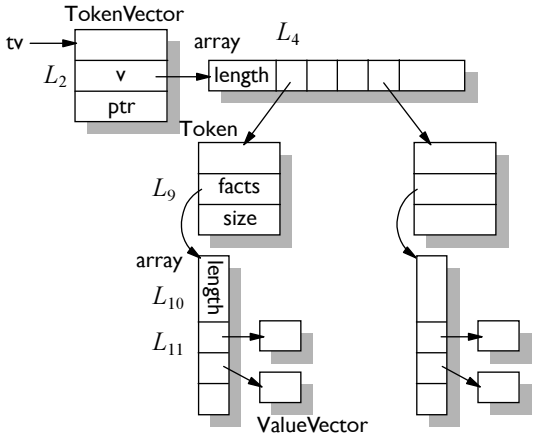

Figure 2: Data structures accessed and pointer references chased in the `findInMemory()` method.

```
TokenLoop:
 for (int i = 0; i < tv.ptr; i++) {
  Token tmp = tv.v[i];
  prefetch (&tv.v[i] + c*d);
  for (int j = 0; j < t.size; j++)
   if (!t.facts[j].equals (tmp.facts[j]))
    continue TokenLoop;
  return tmp;
 }
```

Figure 3: Stride prefetching by Wu et al.'s approach.

the data loaded by the two instructions with an intra-iteration stride are farther apart than the size of a cache line, the stride pattern is *exploitable* for software prefetching. Assuming that we have already obtained the memory address for prefetching the data loaded by L_9, we could prefetch the data loaded by L_{10} based on the same memory address.

Figure 4 shows the resulting optimized code using our approach, where *o* denotes the offset of the `facts` field in the `Token` object and *s* denotes the stride between L_9 and L_{10}. We also assume that the stride is longer than the cache line. We perform three types of prefetching in the optimized code, inter-iteration stride prefetching, dereference-based prefetching, and intra-iteration stride prefetching. Notice that we do not necessarily assume hardware support for the speculative load, and we could obtain an equivalent effect with a sequence of ordinary instructions.

```
TokenLoop:
 for (int i = 0; i < tv.ptr; i++) {
  Token tmp = tv.v[i];
  tmp_pref = spec_load (&tv.v[i] + c*d);
  prefetch (tmp_pref + o);
  prefetch (tmp_pref + o + s);
  for (int j = 0; j < t.size; j++)
   if (!t.facts[j].equals (tmp.facts[j]))
    continue TokenLoop;
  return tmp;
 }
```

Figure 4: Stride prefetching by our approach.

3. OUR PREFETCHING ALGORITHM

We describe our algorithm for stride prefetching intended for use in a JIT compiler. The JIT compiler is invoked for a method when the method is about to be executed. It may be compiling the method for the first time, or recompiling the method with more aggressive optimizations. Thus, actual values for the parameters are available at compile time, and our algorithm fully exploits this information.

As one of the optimization phases, our algorithm transforms the input code in the intermediate representation of a method into code augmented with prefetch instructions. Given a method, it first attempts to identify loops, constructing a loop nesting forest. The algorithm then traverses the loops in each tree in a postorder traversal, walking the trees in the program order.

For each loop, the algorithm performs the following three steps. First, it constructs a dependence graph of the load instructions in the loop. As explained below, the reference-chasing sequences of load instructions are connected in the graph, limiting the number of pairs of load instructions we must check for intra-iteration stride patterns. Second, the algorithm performs object inspection to detect both inter-iteration and intra-iteration stride patterns. It attempts to partially interpret the loop body a certain number of times, annotating the dependence graph with the stride patterns discovered. Finally, we generate prefetching code based on stride patterns recorded in the dependence graph. We prefetch data for a load instruction only when it is effective. The more instructions there are which are data dependent upon a particular load instruction, the more effective prefetching is estimated to be for that load instruction.

Before we describe each step in detail in the subsequent subsections, we note that a nested loop with a small trip count is handled in a manner similar to [24]. When we process the parent loop, all the load instructions in the nested loop are considered again as if they were in the parent loop. Our algorithm detects that a loop has a small trip count when it is performing object inspection. Alternatively, we could rely on execution profiles if the system supports online profiling.

3.1 Construction of a Load Dependence Graph

We utilize a directed graph, called a *load dependence graph*, to capture reference-chasing sequences of load instructions. Each node of the graph is a load instruction using a reference as an operand. A directed edge exists from node L_1 to node L_2 if and only if L_2 is directly data dependent upon L_1. That is, L_2 loads data using the value loaded by L_1, which must thus also be a reference.

When the Java bytecode is used as an intermediate representation, the instructions that can be a node of a load dependence graph include `getfield`, `getstatic`, `aaload`, `iaload`, `daload`, `arraylength`, and others. Only three instructions can be non-leaf nodes in the graph: `getfield` and `getstatic` instructions yielding reference values, and `aaload`.

For a given loop, we construct a load dependence graph of the load instructions in the loop. When it has a nested loop, the load instructions in the nested loop are also considered only if it has a small trip count. We can construct the graph, for instance, by utilizing the use-def chains built for the method containing the loop, but many other ways are possible.

Figure 5 shows a part of the load dependence graph constructed for the doubly nested loop in the `findInMemory()` method. The load instructions in the nested loop also appear in the graph since it has a small trip count.

3.2 Object Inspection

After constructing a load dependence graph for the target loop, we perform object inspection to detect stride patterns. We attempt to partially interpret the loop body a certain number of times (for example, 20 times) using the actual values of the parameters, and record the memory addresses used by the load instructions in the graph. After this partial interpretation, we analyze the trace of the memory addresses. While we check each of the load instructions for inter-iteration patterns, we also check each of the adjacent pairs in the graph for intra-iteration patterns. If the majority (for example, over 75%) of the strides of a load or a pair of loads are the same, we recognize that they have stride patterns, and annotate the corresponding node or edge with the constant stride value.

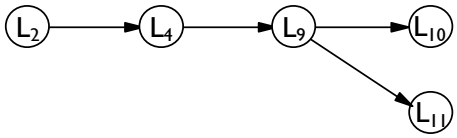

Figure 5: A load dependence graph for the candidate load instructions in the `findInMemory()` method.

Object inspection interprets the method's instructions in the intermediate representation, starting from the method entry. Although the actual values for the parameters are available, there will still be some cases where the operand is not available, and when those cases happen we use a special value, `unknown`, for the operand. Any instruction that involves an unknown operand will have an unknown result at interpretation.

Object inspection must be free of side effects. In particular, we must prevent the interpretation of store instructions from causing any visible effects. To do this, we make a copy of the stack frame, and interpret each store instruction into a local variable within the copied stack frame. Also, we interpret each store instructions into an object by recording the updated address and the value in a hash table, and accordingly interpret a load instruction from an object by first looking in the hash table. For a similar reason, we prepare a private heap, and interpret object-creating instructions using the private heap.

We may encounter one or more loops before reaching the entry point of the target loop. We interpret the body of such a loop only once. This is because the induction variable or the recurrent reference variable of a loop is often initialized without depending on the results of the preceding loops. For instance, it is often the case that the induction variable is initialized to zero, and that the recurrent reference variable is initialized by chasing one of the parameters. Also, since object inspection must be lightweight, we cannot afford to interpret other loops until we really exit from them at any rate.

Finally, we interpret a method invocation by simply skipping it and assuming that the return value, if any, is `unknown`. Alternatively, we could step into the callee method for a non-virtual invocation or all the methods potentially invoked for a virtual invocation. Making object inspection inter-procedural might improve the accuracy of our analysis, but it would increase the compilation time, requiring the trade-off to be carefully assessed.

3.3 Generation of Prefetching Code

After object inspection, we generate the prefetching code based on the stride patterns recorded in the dependence graph. We first explain the code sequences we generate to exploit stride patterns. As shown below, the prefetching code for a load instruction varies depending on the types of stride patterns of the instruction and the adjacent nodes. We then explain a simple profitability analysis to remove ineffective and redundant prefetching codes. Fi-

nally, we discuss mapping of two prefetch instructions we assume, `prefetch` and `spec_load`, to hardware instructions.

Code Sequences

Consider a node L_x with an inter-iteration stride d. If L_x has no adjacent node, or, if all of the adjacent nodes have inter-iteration stride patterns, we generate the following code for prefetching data accessed by L_x in c iterations after the current iteration, where we denote as $A(L)$ the memory address of data loaded by L in the current iteration.

```
prefetch (A(Lx) + d*c);
```

Otherwise, there exists a node, L_y, which is adjacent to L_x and does not exhibit an inter-iteration stride pattern. We then generate the following code for prefetching data accessed by L_x and L_y in c iterations after the current iteration,

```
a = spec_load (A(Lx) + d*c);
prefetch (F[Lx,Ly](a));
```

where $F[L_x, L_y]$ denotes a function which maps the memory address produced by L_x to the memory address used by L_y. Typically, the function simply adds a constant offset to the input address. The code sequence performs both (inter-iteration) stride prefetching and dereference-based prefetching. Notice that L_x and L_y can never have an intra-iteration stride pattern. The existence of the intra-iteration pattern between the two implies that L_y has an inter-iteration pattern, which contradicts that L_y does not exhibit such a stride pattern.

The case that L_y does not have an inter-iteration stride pattern opens opportunities for exploiting intra-iteration stride patterns. For each node L_z which has an intra-iteration stride pattern with L_y directly or transitively, we generate the following prefetching code,

```
prefetch (F[Lx,Ly](a) + S[Ly,Lz]);
```

where $S[L_y, L_z]$ denotes the stride between L_y and L_z.

The actual value for the *scheduling distance* c depends on the processor's cache parameters and the amount of computation and number of memory accesses in the loop body. While we cannot change the cache parameters, we can increase the amount of computation by unrolling the loop.

Profitability Analysis

Since the prefetch instructions consume processor resources and memory bandwidth, we must be selective in issuing them. Ideally, we should generate prefetching codes for those load instructions that frequently cause cache misses. However, it is quite difficult to predict the frequency of cache misses by a load instruction at compile-time, because it subtly depends on many factors, including cache parameters and static and dynamic instruction streams surrounding the load instruction.

Instead, we perform a simple but effective profitability analysis. We generate the prefetching code for a load instruction L only when it satisfies the following three conditions. First, one or more instructions must be data dependent on L. Second, data accessed by L must not apparently share the same cache line with data for which the prefetch code is already issued. Finally, when L has an inter-iteration stride pattern, the stride must be larger than half of the cache line. Prefetching for such a load instruction will not be profitable, especially on processors with hardware prefetching [13].

Table 2: Parameters related to prefetching on the Pentium 4 and the Athlon MP.

Processor	L1 size (KB)	L1 line size (B)	L2 size (KB)	L2 line size (B)	#DTLB entries
Pentium 4	8	64	256	128	64
Athlon MP	64	64	256	64	256

Table 3: Description of the SPECjvm98 and the JavaGrande v2.0 Section 3.

Programs	Description	Compiled code (%)
mtrt	Two threaded ray tracing	75.1
jess	Java expert shell system	70.3
compress	Modified Lempel-Ziv method	93.6
db	Memory resident database	92.3
mpegaudio	MPEG Layer-3 audio decompression	87.0
jack	Java parser generator	36.2
javac	Java compiler from JDK1.0.2	51.9
Euler	Computational fluid dynamics	79.5
MolDyn	Molecular dynamics simulation	85.4
MonteCarlo	Monte Carlo simulation	48.0
RayTracer	3D ray tracer	79.8
Search	Alpha-beta pruned search	73.4

Mapping to Hardware Instructions

We can realize each of the `prefetch` and `spec_load` instructions in two ways, the hardware instruction and a load instruction guarded by a software exception check. If the underlying processor provides the hardware support, we should obviously use the hardware instruction. It takes less processor resources, and imposes less impact on bandwidth since the processor cancels the execution of the instruction when a data translation lookaside buffer (DTLB) miss will occur. Currently, the prefetch instructions are supported by most of the modern commercial processors, while the speculative load instruction only receives support in the Intel IA-64 [8] and the SPARC-V9 [19] architectures.

However, a guarded load instruction is sometimes preferable even in the presence of hardware support, since we can use the guarded load instruction to fill a missing DTLB entry in advance (called *TLB priming* in [9]). Thus, when the stride is larger than half of the page size, the guarded load instruction might be better than the hardware instruction. A more important case is to prefetch the address obtained by a dereference. In the above example, it is not surprising if the difference between $A(L_x)$ and $A(L_y)$ is often larger than half of the page size. Thus, it might be better to use the guarded instruction for prefetching $A(L_y)$.

4. EXPERIMENTAL RESULTS

We implemented our prefetching algorithm and evaluated the following two algorithms:

INTER This option enables only inter-iteration stride prefetching. Note that this configuration is a limited emulation of Wu's stride prefetching using our prefetching algorithm. The limitations are that 1) we use object inspection instead of off-line profiling, and 2) we apply prefetching only to in-loop loads.

INTER+INTRA This option enables both inter- and intra-iteration stride prefetching as described in Section 3.

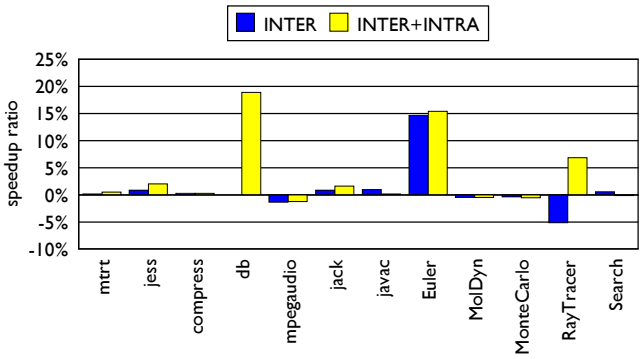

Figure 6: Speedup ratios on the Pentium 4.

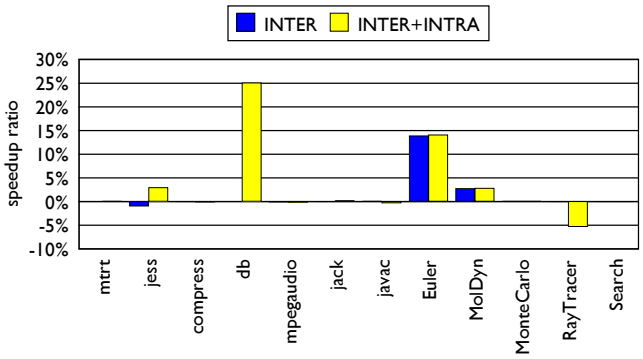

Figure 7: Speedup ratios on the Athlon MP.

We investigated the first 20 iterations of a given loop to collect the constant strides. We recognize that a constant stride is dominant when it matches 75% of the all collected strides. We fixed the scheduling distance as one iteration for both inter- and intra-iteration stride prefetching because our primary concern was not to optimally tune up both kinds of stride prefetching, but to examine the effectiveness of intra-iteration stride prefetching.

We prototyped our prefetching algorithm as an extension to the JIT compiler [11] of the IBM Developer Kit for Windows, Java Technology Edition, Version 1.3.1. The JVM runs in a mixed-mode, meaning it selectively compiles methods that are executed frequently [22]. The garbage collector [6] of the JVM uses a traditional mark-and-sweep algorithm. Live objects are packed by *sliding* compaction, which does not change their internal order on the heap. Thus, the garbage collector usually preserves constant strides among the live objects. We set the initial and the maximum heap sizes to 128 MB.

The measurements were done on two workstations, one with a 2 GHz Intel Pentium 4 [9] processor and 1 GB of memory, and the other with a 1.2 GHz AMD Athlon MP [1] processor and 512 MB of memory. Both of these processors provide out-of-order superscalar execution, and software and hardware prefetching mechanisms. Table 2 shows the parameters of the Pentium 4 and the Athlon MP related to prefetching [9, 1]. The major differences between the two processors that affect this research are that 1) the Pentium 4 provides a smaller number of DTLB entries, and 2) the target cache levels for software prefetching are the L2 cache on the Pentium 4 and the L1 cache on the Athlon MP. We used a load instruction guarded by a software exception check for intra-iteration stride prefetching on the Pentium 4 in order to fill a missing DTLB entry. Otherwise, we used a prefetch instruction provided by the processor. We used the Microsoft Windows 2000 Professional operating system on both workstations.

We used two benchmark suites, the SPECjvm98 benchmark [20] and the JavaGrande v2.0 [12] benchmark Section 3. To evaluate the performance improvement, we report the best run times reported by the benchmarks themselves. The scores of the SPECjvm98 are usually measured by their best run times under automatic continuous execution. This means it tends to exclude the JIT compilation time because after several runs, the benchmark is in the steady state where the JIT compilation rarely occurs. We iterated each benchmark ten times in its auto run mode. We set the problem size to 100. For the JavaGrande benchmarks, we ran each benchmark once and therefore that run includes the JIT compilation time in the best run time. We set the problem size to "Size A". To evaluate the overhead of compilation time, we report the total execution time and the total JIT compilation time of each benchmark. Table 3 shows the description of the benchmarks in the SPECjvm98 and the JavaGrande v2.0 Section 3. The last column shows the ratios of the execution time of the compiled code against the total execution times on the Pentium 4. The benchmarks are suitable for evaluating the optimizations by a JIT compiler because the compiled code consumes over 70% of the total execution time except for *jack*, *javac*, and *MonteCarlo*.

4.1 Performance

Figure 6 and Figure 7 show the speedup ratios on the Pentium 4 and on the Athlon MP, respectively. The baseline is the execution time without stride prefetching.

Overall, the combination of inter- and intra-iteration stride prefetching gives a performance improvement better than only using inter-iteration stride prefetching. The algorithm INTER was effective for *Euler* on both processors, and was effective for *MolDyn* on the Athlon MP. The algorithm INTER+INTRA improved those programs, as well as *jess* and *db* on both processors, and also improved *RayTracer* on the Pentium 4.

In particular, INTER+INTRA achieved an 18.9% speedup of *db* on the Pentium 4, and also achieved a 25.1% speedup on the Athlon MP, while INTER was ineffective on both processors. This program spends more than 85% of its execution time in a shell sort loop that reorders a number of large records and frequently causes cache misses and DTLB misses [18]. Each record contains a number of `Vector` and `String` objects, and they only have intra-iteration constant strides between the containing records in the sorting loop.

And also, INTER+INTRA achieved a 2.0% speedup for *jess* on the Pentium 4, and achieved a 2.9% speedup on the Athlon MP, while INTER had very small or negative speedups. The improvements of the *jess* benchmark by INTER+INTRA were rather small for two reasons. First, the method `findInMemory()` is hot, but not dominant. The hottest method, which the `findInMemory()` method is inlined into, uses only about 25% of the compiled code execution time, while the compiled code takes about 70% of the total execution time. Second, the cache line size is sufficiently large to contain both the `Token` object and the array object pointed to by the `facts` field. The speedup on the Athlon MP was slightly larger than the Pentium 4 because the Athlon MP has a larger number of DTLB entries.

Since the benchmark *Euler* has inter-iteration constant strides in its main data structures, large two-dimensional arrays of vectors, both algorithms achieved similar speedups on the Pentium 4 and the Athlon MP. The algorithm INTER+INTRA achieved a 15.4% speedup on the Pentium 4, and 14.0% speedup on the Athlon MP.

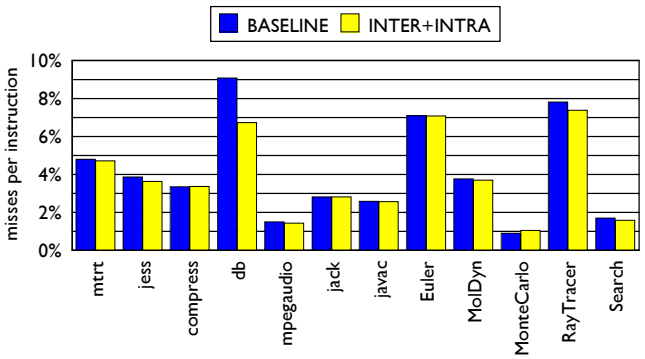

Figure 8: L1 cache load MPIs on the Pentium 4.

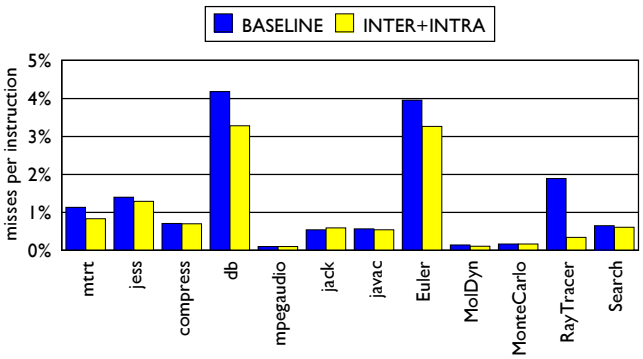

Figure 9: L2 cache load MPIs on the Pentium 4.

While the *MolDyn* benchmark also has inter-iteration constant strides, neither algorithm improved it on the Pentium 4. This is because the main data structure of *MolDyn* is a one-dimensional array of molecule objects that fits in the L2 cache given the problem size in this experiment. In contrast, both algorithms achieved small speedups on the Athlon MP, since the molecule objects are prefetched into the L1 cache.

The *RayTracer* benchmark showed an anomaly in that the algorithm INTER+INTRA improves the performance on the Pentium 4 and degrades it on the Athlon MP. One of the target loops of *RayTracer* contains an invocation of a recursive method. On the Pentium 4, stride prefetching in that target loop also reduces the cache misses in the other methods where prefetches are not inserted. We need further investigation of stride prefetching for this method invocation.

Both algorithms slightly degraded the *mpegaudio* benchmark on the Pentium 4. This is because the cache miss ratios and the DTLB miss ratio were quite small, as we will see in Section 4.2. The benchmarks *compress*, *javac*, and *Search* do not contain code fragments where either intra- or inter-iteration stride prefetching are applicable.

4.2 Cache Misses and DTLB Misses

In this section, we investigate the effect on cache misses and DTLB misses, comparing the execution without stride prefetching (labeled as "BASELINE") and using our prefetching algorithm, INTER+INTRA. We measured these results on the Pentium 4 using the Intel VTune Performance Analyzer [10] Version 5.0. We use a metric, *misses per instruction* (MPI), which is the number of dynamic miss events divided by the number of retired instructions. Note that in the out-of-order superscalar execution, cache misses or DTLB misses are observed by all of the concurrently executed load instructions that access the same cache line or the same TLB entry. Thus, prefetching and TLB priming do not reduce the number of fills, but can reduce the number of miss events.

Inserting prefetch instructions increases the number of retired instructions, but they are relatively few compared to the reduction of the number of miss events. Our prefetching algorithm increases the number of retired instructions for *db* by 9.7%, for *RayTracer* by 6.9%, for *jess* by 2.2%, and for the other benchmarks by less than 2%.

Figure 8 shows the L1 cache load MPIs. Our prefetching algorithm greatly decreased the L1 cache load MPI of *db*, and slightly decreased the L1 cache load MPIs of *jess* and *RayTracer*. This is because intra-iteration constant stride prefetching is applicable to these benchmarks, and an additional load instruction for prefetching brings the target cache line into the L1 cache. The L1 cache MPIs of *mpegaudio* and *MonteCarlo* are quite small, and thus prefetching is not profitable for these benchmarks.

Figure 9 shows the L2 cache load MPIs. Our prefetching algorithm greatly decreased the L2 cache load MPI of *RayTracer*, but this benefit was not fully predicted at compile-time, because it was caused by the effects across multiple methods, as described above. Our prefetching algorithm also decreased the L2 cache load MPIs of *db*, *Euler*, and *mtrt*. Since inter-iteration stride prefetching is applicable to the *Euler* benchmark, we can reduce its L2 cache load MPI more by a longer scheduling distance.

Figure 10 shows the DTLB load MPIs. Our algorithm greatly decreased the DTLB load MPIs of *RayTracer* and *db*, and slightly decreased the DTLB load MPI of *jess*. The reduction of the L2 cache misses for *mtrt* is slightly better than that for *jess*, but the speedup ratio of *jess* was better than that of *mtrt*. It suggests the importance of reducing the DTLB misses on the Pentium 4.

4.3 Compilation Time

The left-hand bars in Figure 11 show additional compilation time for our prefetching algorithm (INTER+INTRA) normalized by the total JIT compilation time (indexed by the left axis). The baseline is compilation time without stride prefetching. The additional compilation time for our prefetching algorithm were less than 3.0% of the total JIT compilation time, and exceeded 2.0% for *mpegaudio*, *Euler*, and *MolDyn*, since these benchmarks contain a number of loops that refer to arrays of objects. For the other benchmarks, the overheads were smaller than 1.5%.

The right-hand bars in Figure 11 show total JIT compilation time normalized by the total execution time of each benchmark (indexed by the right axis). The total JIT compilation time was less than 13% of the total execution time. Thus, by multiplying these numbers, we observed that the additional JIT compilation time required for our prefetching algorithm, in other words the runtime overhead, was less than 0.4% of the total execution time.

5. RELATED WORK

Luk and Mowry [14, 15] studied software prefetching for recursive data structures (RDSs) such as linked lists, trees, and graphs. At a given RDS node n_i, they wish to prefetch the node n_{i+d} that will be visited d nodes after n_i, and they propose three schemes for computing the address of n_{i+d}, A_{i+d}, at node n_i. While *greedy prefetching* approximates A_{i+d} as one of the pointers from n_i, *history-pointer prefetching* adds a *jump-pointer* at n_i for this purpose that contains the observed value of A_{i+d} during a recent traversal. *Data lineariza-*

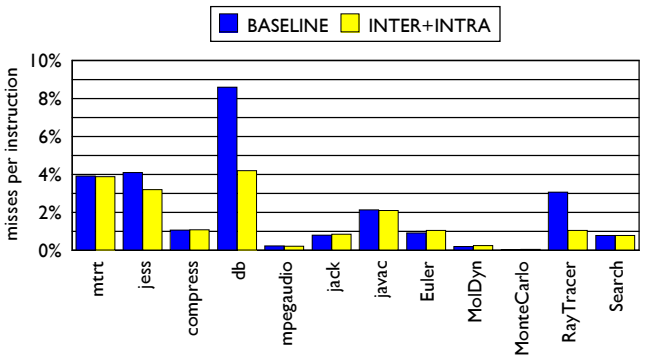

Figure 10: DTLB load MPIs on the Pentium 4.

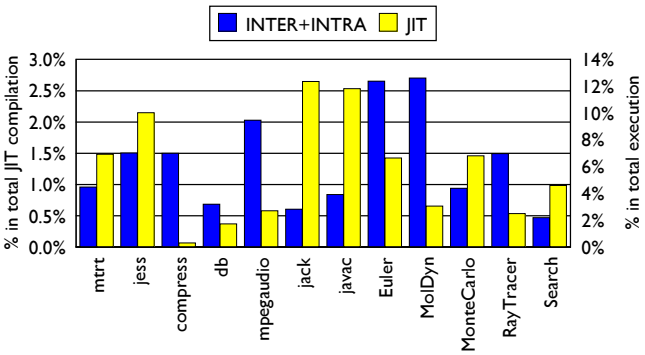

Figure 11: Compilation time for prefetching and total JIT compilation time.

tion attempts to map the nodes of a RDS onto an array, so that one can easily predict A_{i+d} without any pointer dereference. Data linearization exploits the same stride pattern information as (inter-iteration) stride prefetching. In contrast to the attempts of stride prefetching to *discover* patterns, data linearization attempts to *create* them, which is almost impossible for a compiler to do automatically. Indeed, they only simulate data linearization by hand for two benchmarks, where the RDSs already exhibit stride patterns, thus requiring no mapping for linearization.

Stoutchinin et al. [21] proposed an automatic approach for prefetching data for RDSs, which relies solely on compiler analysis. They first identify pointer-chasing recurrences in loops with a low complexity algorithm. They then perform a profitability analysis to investigate whether there are enough processor resources and available memory bandwidth for profitable prefetching. Only if there are, they issue prefetch instructions for the data accessed through *induction pointers*, which are the addresses used by the loads involved in the pointer-chasing recurrences. In other words, they assume that the loads exhibit constant stride patterns, and rely on the profitability analysis to avoid performance degradation when the assumption does not hold. However, their experimental results show that performance is actually degraded in six of the eleven programs they tested.

Wu [23] and Wu et al. [24] attempted to discover and exploit the inter-iteration stride patterns of load instructions through efficient off-line profiling. They exploit three stride patterns, strong single stride, phased multiple-stride, and weak single stride, by issuing different prefetching code sequences. As expected, the majority of loads prefetched have strong single stride patterns. Also, they handle both in-loop and out-of-loop loads, but the performance gain from prefetching out-of-loop loads was insignificant. We extend their approach in two significant ways. First, we attempt to discover and exploit intra-iteration patterns as well as inter-iteration patterns. Second, we provide stride prefetching in a dynamic compiler. Since the overhead of off-line stride profiling is still too high for such a dynamic environment, we have invented an ultra-lightweight profiling technique called object inspection. Mainly because we must use such a lightweight profiler, we focus on discovering single stride patterns in in-loop loads, but we believe that we are capturing most of the opportunities for performance gain that stride prefetching can provide.

Chilimbi and Hirzel [4] developed a dynamic prefetching scheme for general-purpose programs that involve extensive pointer dereferencing. They gather a temporal data reference profile and extract the *hot data streams*, which are data reference sequences that are frequently repeated in the same order. The system then dynamically injects code at appropriate program points to detect and prefetch these hot data streams. Prefetching hot data streams does not result in stride prefetching. In other words, loads with stride patterns are not captured as hot data streams. Thus, the two approaches can work effectively together.

Cahoon and McKinley [2] proposed an effective data flow analysis technique for identifying RDS traversals in Java. The analysis contains intra- and inter-procedural components and finds recurrent pointer variables that occur both in loops and in recursive function calls. They use the analysis to drive greedy prefetching and history-pointer prefetching. They do not include stride prefetching, although it is possible as in Stoutchinin et al. [21]. Their approach is based on whole-program analysis, and thus cannot be used for a dynamic compilation system.

6. CONCLUDING REMARKS

We have proposed a new algorithm for stride prefetching which is intended for use in a dynamic compiler. Our algorithm attempts to discover and exploit both inter- and intra-iteration stride patterns. Intra-iteration patterns often result since the constructors in an object-oriented language tend to allocate a bunch of related objects and store references to them into the objects being constructed.

We discover stride patterns using object inspection, an ultra-lightweight technique for dynamic profiling which only a dynamic compiler is able to use. During the compilation of a method, the dynamic compiler gathers an execution profile of the method by partially interpreting the method using the actual values of parameters and while causing no side effects. Also, in order to discover intra-iteration patterns efficiently, we build a load dependence graph which represents as adjacent nodes the load instructions which chase references and thus limits the number of pairs we must check for intra-iteration patterns.

We evaluated an implementation of our prefetching algorithm in a production-level Java JIT compiler. We measured the SPECjvm98 benchmark and Section 3 of the JavaGrande v2.0 benchmark on two different IA-32 processors, an Intel Pentium 4 and an AMD Athlon MP. The results show that our prefetching algorithm achieved up to an 18.9% and 25.1% speedup on the Pentium 4 and the Athlon MP, respectively, while it increased the compilation time by less than 3.0%.

One of the future goals is to extend the scope of load instructions for prefetching in our algorithm. In particular, handling out-of-loop loads in recursive methods is important and expected to be as re-

warding as in-loop loads, but discovering exploitable stride patterns for out-of-loop loads still remains as an open problem. Also, it might be interesting to combine stride prefetching with other types of prefetching, such as history-pointer prefetching and prefetching hot data streams.

Acknowledgments

We thank the members of the Network Computing Platform group of Tokyo Research Laboratory for their support and valuable comments. We also thank the anonymous reviewers for their helpful comments and suggestions.

7. REFERENCES

[1] Advanced Micro Devices, Inc. *AMD Athlon Processor x86 Code Optimization Guide*, Aug. 2001. Document Number 22007J.

[2] B. Cahoon and K. S. McKinley. Data Flow Analysis for Software Prefetching Linked Data Structures in Java. In *Proc. of the International Conference on Parallel Architectures and Compiler Techniques*, Sept. 2001.

[3] D. Callahan, K. Kennedy, and A. Porterfield. Software Prefetching. In *Proc. of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 40–52, Apr. 1991.

[4] T. M. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In PLDI '02 [17], pages 199–209.

[5] S. Dieckmann and U. Hölzle. A Study of the Allocation Behavior of the SPECjvm98 Java Benchmarks. In *Proc. of the 13th European Conference of Object Oriented Programming*, pages 92–115, 1999. LNCS 1628.

[6] R. Dimpsey, R. Arora, and K. Kuiper. Java Server Performance: A Case Study of Building Efficient, Scalable JVMs. *IBM Systems Journal*, 39(1):151–174, 2000.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Publishing Co., Reading, MA, 1996.

[8] Intel Corporation. *Intel Itanium Architecture Software Developer's Manual Volume 3: Instruction Set Reference*, 2001. Revision 2.0, Document Number 245319-003.

[9] Intel Corporation. *Intel Pentium 4 Processor Optimization Reference Manual*, 2001. Document Number 248966.

[10] Intel Corporation. VTune Performance Analyzer. http://www.intel.com/software/products/vtune, 2002.

[11] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, Implementation, and Evaluation of Optimizations in a Just-In-Time Compiler. In *Proc. of the ACM JavaGrande Conference*, pages 119–128, June 1999.

[12] Java Grande Benchmarking Project. Java Grande Forum Benchmark Suite, Version 2.0. http://www.epcc.ed.ac.uk/javagrande, 1999.

[13] N. P. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *Proc. of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990.

[14] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching for Recursive Data Structures. In *Proc. of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 222–233, Oct. 1996.

[15] C.-K. Luk and T. C. Mowry. Automatic Compiler-Inserted Prefetching for Pointer-Based Applications. *IEEE Transactions on Computers*, 48(2), 1999.

[16] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proc. of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Oct. 1992.

[17] *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2002.

[18] Y. Shuf, M. J. Serrano, M. Gupta, and J. P. Singh. Characterizing the Memory Behavior of Java Workloads: A Structured View and Opportunities for Optimizations. In *Proc. of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 194–205, June 2001.

[19] SPARC International, Inc. *The SPARC Architecture Manual Version 9*, 2000. Document Number SAV09R1459912.

[20] Standard Performance Evaluation Corporation (SPEC). JVM Client98 (SPECjvm98). http://www.spec.org/osg/jvm98, 1998.

[21] A. Stoutchinin, J. N. Amaral, G. R. Gao, J. Dehnert, S. Jain, and A. Douillet. Speculative Prefetching of Induction Pointers. In *Proc. of the 10th International Conference on Compiler Construction*, pages 289–303, Apr. 2001. LNCS 2027.

[22] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-In-Time Compiler. *IBM Systems Journal*, 39(1):175–193, Feb. 2000.

[23] Y. Wu. Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching. In PLDI '02 [17], pages 210–221.

[24] Y. Wu, M. Serrano, R. Krishnaiyer, W. Li, and J. Fang. Value-Profile Guided Stride Prefetching for Irregular Code. In *Proc. of the 11th International Conference on Compiler Construction*, pages 307–324, Apr. 2002. LNCS 2304.