Zou Q, Li XF, Zhang LB. Runtime engine for dynamic profile guided stride prefetching. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 23(4): 633–643 July 2008

Runtime Engine for Dynamic Profile Guided Stride Prefetching

Qiong Zou[1,2] (邹　琼), Xiao-Feng Li[3] (李晓峰), and Long-Bing Zhang[2] (章隆兵)

[1]*Department of Computer Science, University of Science and Technology of China, Hefei 230027, China*

[2]*Key Laboratory of Computer System and Architecture, Institute of Computing Technology, Chinese Academy of Sciences Beijing 100190, China*

[3]*Intel China Research Center, Beijing 100190, China*

E-mail: {zouqiong, lbzhang}@ict.ac.cn; xiaofeng.li@intel.com

Received December 10, 2007; revised May 9, 2008.

Abstract Stride prefetching is recognized as an important technique to improve memory access performance. The prior work usually profiles and/or analyzes the program behavior offline, and uses the identified stride patterns to guide the compilation process by injecting the prefetch instructions at appropriate places. There are some researches trying to enable stride prefetching in runtime systems with online profiling, but they either cannot discover cross-procedural prefetch opportunity, or require special supports in hardware or garbage collection. In this paper, we present a prefetch engine for JVM (Java Virtual Machine). It firstly identifies the candidate load operations during just-in-time (JIT) compilation, and then instruments the compiled code to profile the addresses of those loads. The runtime profile is collected in a trace buffer, which triggers a prefetch controller upon a protection fault. The prefetch controller analyzes the trace to discover any stride patterns, then modifies the compiled code to inject the prefetch instructions in place of the instrumentations. One of the major advantages of this engine is that, it can detect striding loads in any virtual code places for both regular and irregular code, not being limited with plain loop or procedure scopes. Actually we found the cross-procedural patterns take about 30% of all the prefetchings in the representative Java benchmarks. Another major advantage of the engine is that it has runtime overhead much smaller (the maximal is less than 4.0%) than the benefits it brings. Our evaluation with Apache Harmony JVM shows that the engine can achieve an average 6.2% speed-up with SPECJVM98 and DaCapo on Intel Pentium 4 platform, in spite of the runtime overhead.

Keywords stride prefetching, dynamic profiling, runtime system

1 Introduction

Software prefetching is widely used for improving memory access performance. A couple of typical prefetching techniques were developed to guide the insertion of prefetch instructions into programs[1−6]. At the same time, researchers in compilation area proposed various algorithms to automatically identify the prefetch opportunities and insert prefetch instructions into compiled code[3,7,8]. It is relatively easy to prefetch array-based data references in numerical applications, while it is challenging to effectively prefetch pointer-chasing data structures because there exist difficulties in discovering prefetch patterns in irregular code. Wu *et al.*[1] developed a value profiling method of discovering stride patterns for irregular programs. Their work yielded significant performance improvement with SPEC2000; however, the proposed methodology is only applicable to static compiler and offline profiling. Robert *et al.*[2] proposed a technique that supports post-link stride prefetching, which does not require source code availability or recompilation. Their work still needs offline profiling with training input set.

Dynamic online profiling is more desirable sometimes than the offline profiling, because it provides the same context (runtime environment and data input) for both profiling run and execution run[4,5]. It is especially interesting with the popularity of management runtime systems, such as Java, C#, Ruby, etc., due to the nature of JIT compilation. Since program analysis and compilation are conducted right before a method is going to be executed, it is usually infeasible to have pre-execution instrumentation or offline profiling.

The major challenge to dynamic profiling is the run-

Regular Paper

Supported by the National Natural Science Foundation of China under Grant Nos. 60673146, 60603049, 60736012, and 60703017, the National High Technology Development 863 Program of China under Grant No. 2006AA010201 and No. 2007AA01Z114, the National Basic Research Program of China under Grant No. 2005CB321601.

time overhead control. Since the profiling overhead is considered a part of the overall execution time, it should be kept minimal in order to be amortized by the improvement memory accesses. The commonly used online profiling techniques include execution sampling[9] and program instrumentation[10], or both [11]. Execution sampling can be done in pure software (e.g., stack frame sampling) or with hardware assistance (e.g., cache miss sampling). Although execution sampling incurs low runtime overhead, it normally is incapable of getting the fine-grained profiles based on program semantics, such as basic block frequencies or value profile. Program instrumentation is perceived to have "opposite" characteristics to execution sampling. That is why sometimes a hybrid method is used, but the method proposed by Arnold *et al.*[11] is not very useful when collecting the necessary information for stride prefetching, i.e., the address values of successive load operations. Inagaki *el al.*[4] suggested a light-weighted runtime profiling method of stride prefetching, where the loops in a method are interpreted for a few iterations before JIT compilation. This method can discover the stride patterns in loops; however, it does not support cross-procedural prefetching.

In this paper, we propose a runtime prefetch engine. The engine uses program instrumentation to trace load addresses; it can detect stride patterns in any virtual code places. Once a stride pattern is discovered, the engine modifies the compiled code to inject the prefetching instructions and remove the instrumentations. In order to reduce the profiling overhead, the engine avoids instrumenting the same data loads more than once. The engine also eliminates useless instrumentations adaptively at runtime. We implemented the engine in Apache Harmony, a product-grade JVM[12], and evaluated the performance with SPECJVM98[13] and DaCapo[14] benchmarks on an Intel Pentium 4 machine by using its prefetch instruction[15]. The result shows that an average 6.2% speed-up can be achieved in spite of the 4.0% profiling overhead.

The major contributions of this paper are as follows.

First, we developed a runtime pattern detection mechanism, which can discover cross-procedural opportunities, besides those in plain loops for cross-iteration and intra-iteration prefetchings. To the best of our knowledge, this is the first attempt to discover all of them at runtime.

Second, in order to reduce the runtime profiling overhead, we only instrument one of multiple loads to the same data with program analysis. Moreover, we can remove the useless instrumentations at runtime.

Third, we put together the techniques we developed into a runtime prefetch engine, implemented it in product-grade JVM, and evaluated the results with representative workloads.

The remainder of the paper is organized as follows. In Section 2 we introduce the runtime prefetch engine, with focus on the pattern detection mechanism. In Section 3, we discuss the techniques to reduce profiling overhead. In Section 4, we present the evaluation results. We discuss related work in Section 5. In Section 6, we conclude the paper and suggest the work for next step.

2 Runtime Prefetch Engine

In this section, we will describe how our runtime prefetch engine works. Since the work was implemented in JVM, we will first give a quick introduction to some JVM specific concepts.

2.1 Load Operations

All the heap accesses in JVM can be classified into table accesses and object accesses. (The rest memory accesses are mainly to stack and code, which are usually not considered as parts of heap in JVM.) The majority of heap accesses are object accesses, e.g., they take 90% of total heap accesses in several benchmarks in SPECJVM98[16]. In this work, we only consider object accesses, which include the accesses to both object fields and array elements. For accessing an object in JVM, the object reference should be put on the stack first, and then a *getfield* or *xaload* (*x* is a wildcard for array element type) instruction dereferences the stack value to read the object field or array element in the heap. Basically the runtime prefetch engine instruments these load instructions to profile the target address values, and tries to prefetch the data once a stride pattern is discovered.

When two load instructions are executed closely in time, and their load address values show some striding difference patterns repeatedly, we call them a pair of related instructions (RI). Related instructions are the interesting targets for prefetching optimization. They have to be close enough in execution time so that the fetched data by one instruction can be useful to the other one; otherwise, the fetched data in cache could have been flushed before the second instruction uses it. Note related instructions can be the same one but executed repetitively, which is common for the loads in loops.

Fig.1 gives three representative code examples extracted from SPECJVM98 *jess* and Dacapo *pmd.* The first example in Fig.1(a) is a simple array access that causes lots of cache misses in a loop. It can be a good candidate for cross-iteration prefetch. Fig.1 (b) is an example that our compiler cannot identify that *pop*() is invoked in a loop due to the multiple-level call chain, so it is a potential candidate for cross-procedural prefetch. We found most of the load cache misses in *jess* are caused by similar codes. Fig.1(c) does not show the full code snippet for the load operation that causes cache misses. We show it because it is a typical programming pattern in Java with *Iterator*. The call chain of *get*() cannot be inlined by our compiler as the example in Fig.1(b). We will show that our runtime engine can profile the code dynamically, identify the stride patterns, insert prefetch instructions, and improve the applications performance successfully.

```
In one method:
        for (int j=RU.FIRST_SLOT; j< fact.size(); j++){
         if (fact.get(j).type()==RU.FUNCALL)}
              ValueVector subexp=
              fact.get(j).FuncallValue();
          Value r = Execute(subexp, context);
          fact.set(r,j);
         }
       }
Here is get():
        public final Value get(int i) {
             return v[i]; //frequent cache miss
         }
                              (a)
```

```
In one method:
        while (tok.ttypes==‘(’){
             . . .
             tok=jts.nextToke();
        }
Here is nextToken():
        JessToken nextToken(){
             . . .
             JessToken tok=(JessToken)
        m_stack.pip();
             m_string.append(tok.toString()+“  ”);
             m_lineno=tok.lineno;
          return tok;
        }
Here is pop():
        public synchronized E pop(){
             . . .
             int index = elementCount −1;
             //frequent cache miss
              E obj=(E)elementData[index];
              removeElementAt(index);
              return obj;
        }
                              (b)
```

```
      for(Iterator i=acus.iterator(); i.hasNext();){
          ASTCompliationUnit node=
                          (ASTCompilationUnit) i.next();
             visit(node,ctx);
        }
                              (c)
```

Fig.1. Code examples that cause frequent cache misses with load operations. (a) From SPECJVM98 jess. (b) From SPECJVM98 jess. (c) From DaCapopmd.

2.2 Framework of the Runtime Engine

The framework of the engine is illustrated in Fig.2. It consists of two components in a JIT compiler, a prefetch controller, and a trace buffer.

When a JIT compiler is invoked to compile a method, it processes the code to identify the candidate load operations; then the compiler instruments the candidate loads to collect execution profile.

The profile data is recorded in the trace buffer which is arranged as a Sequential Store Buffer (SSB)[17]. New trace information is written into an entry of the buffer sequentially. Each entry has three fields: the program counter (PC) value, the address value of the object beginning, and the offset value to the object beginning. The trace buffer length is a fixed value. The memory page after the trace buffer is write-protected. When the buffer is fully filled, the next write will trigger a protection fault. The fault handler implements the functionalities of the prefetch controller, which is the key component of the engine.

To use write-protected faulting mechanism is better than a counter increment and comparison, because the then instrumentation code is free from the bookkeeping; otherwise, the instructions for the counter increment and comparison will appear for every instrumentation site and spread over the entire compiled code base.

When the prefetch controller is invoked due to a protection fault, it analyzes the data in the trace buffer and checks if there is any stride pattern exposed. If a stride pattern exists, the prefetch controller will modify the compiled code to inject the prefetching instruction in place of the original instrumentation instructions; at the same time, it tries to eliminate the useless instrumentations as well. The useless instrumentations refer to those of the load operations which do not exhibit any stride pattern after long enough profiling time. We will discuss the details of the prefetch controller in following text.

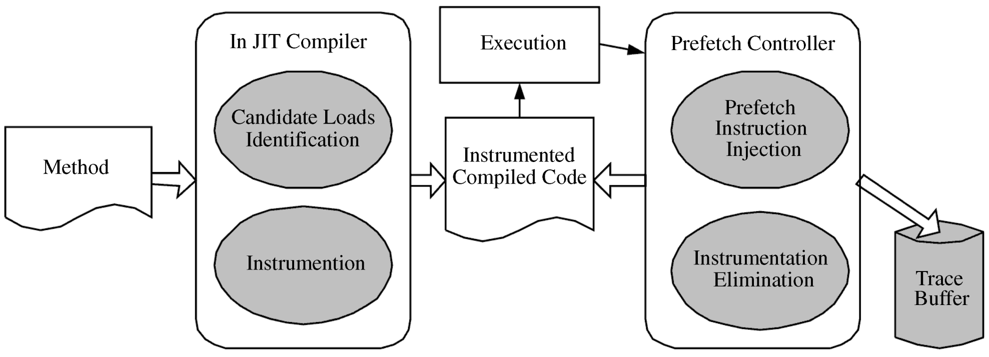

Method
In JIT Compiler
Candidate Loads Identification
Instrumention
Instrumented Compiled Code
Execution
Prefetch Controller
Prefetch Instruction Injection
Instrumentation Elimination
Trace Buffer

Fig.2. Framework of our runtime prefetch engine. (The grey boxes are the components of the engine.)

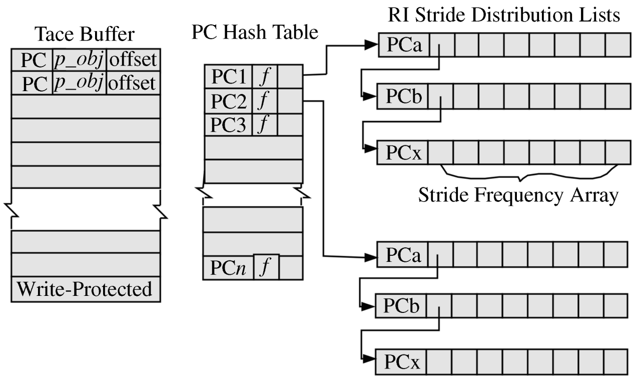


Fig.3. Core data structures for runtime detection of stride pattern.

2.3 Core Data Structures

Fig.3 is the core data structure for runtime detection of stride patterns. The trace buffer holds the address value profile of the instrumented loads. PC value in each buffer item is the key to index a PC value hash table. The hash table entry for the instruction at position of PC maintains the information of its related instructions in a list, called "RI stride distribution list". Each hash table entry also records the frequency (f) of the instruction executed up to the moment.

The RI stride distribution list for the instruction at PCi records its executed RIs and the distribution of the address differences between PCi and its RIs. One node in the list is for one RI of PCi encountered at runtime. For instance, in the figure, PC1 has three RIs: PCa, PCb and PCx. The address value differences between PC1 and PCa are kept in the node for PCa, which is called a stride frequency array. Each element of the array records the frequency of a certain stride value.

The trace buffer is full and the prefetch controller is triggered, the prefetch controller analyzes the trace data, and increments the execution frequency f for each instruction appearing in the buffer. The prefetch controller also updates the RI stride distribution list. It computes the value difference between the addresses of the load instruction and its RI, then increments the frequency of the stride value in the corresponding element of the stride frequency array. If the stride frequency in a certain element becomes bigger than a threshold, the prefetch controller believes a stride pattern is discovered. It will modify the compiled method to inject the prefetch instruction and cease the instrumentation's further effect. The prefetch controller may optionally add a new node to the list if a new RI is encountered during the trace analysis. Finally, the prefetch controller resets the trace buffer pointer to point to the first item for next cycle of trace collection.

2.4 Runtime Detection Implementation

The critical problem of our prefetch engine design is how to control the runtime overhead. We had tried a few approaches and finally chose the current one, which we believe is a good balance between the prefetching benefits and runtime overhead.

One tradeoff in our engine is about the trace buffer design. We chose sequential store buffer with write-protection as overflow checking. Although the fault handler processing requires careful programming, SSB saves the overflow checking overhead obviously. In our implementation, the trace buffer has 4K entries, which is large enough to catch stride patterns early while small enough to be processed quickly.

When the prefetch handler is analyzing the data in the trace buffer, it examines the instructions one by one. For each instruction, it considers a number of successive instructions in the buffer as its RIs. That is, the prefetch handler uses a sliding window for RI filtering. The window size needs not large, because RIs should be close enough to keep the prefetching valid. On the other hand, the size should be adequate so that no important RI is missed. We found that 12 entries in the window suffice our requirements. The prefetch controller takes one instruction in the trace buffer, and updates its RI stride distribution lists according to the following 11 buffer entries, then moves on to next instruction till the buffer end.

The stride frequency array for each RI maintains the frequencies of different stride values. In our implementation, we only maintained the frequencies for stride values between -40 through 40 with interval of 4. The stride frequency array has 21 elements. Element i keeps the frequency of stride value $(i - 10) \times 4$. This makes the frequency update operation very fast. This is essential for the lightweight engine design. We tried the approach mentioned in Robert *et al.*[2]; its overhead was substantially higher than our approach. More importantly, our experiments with the benchmarks showed that most of the stride values were within $-40 \sim 40$. More discussions are given in Section 4, Experimental Evaluations.

When a stride frequency is updated, the prefetch controller checks if it reaches the threshold to be a valid stride to trigger prefetch instruction injection. The threshold is set to be half of the trace buffer length, i.e., 2K in our implementation.

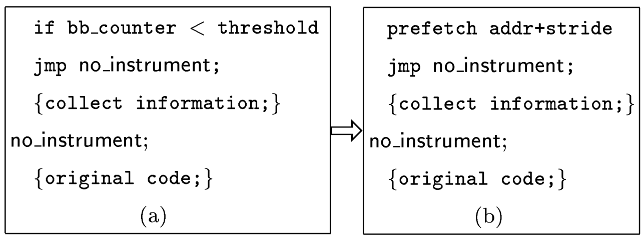
2.5 Injection of Prefetch Instructions

When the stride value S is identified for an RI of current load instruction, the prefetch controller injects the prefetch instruction into the compiled code. Because of the memory subsystem constraints, the real prefetch distance is $psd \times S$, where psd is an architecture dependent value, called *prefetch scheduling distance*. psd ensures that the data be prefetched into cache before it is used by the related instruction. For example, a cross-iteration stride pattern for a load instruction in a loop may require to prefetch the data two or more iterations ahead of its use, because the latency to fetch data into cache can be multiple times of the execution time of an iteration. A heuristic formula for psd computation is as below[10]:

$$psd = \left\lceil \frac{N_{\text{lookup}} + N_{\text{xfer}} \times (N_{\text{pref}} + N_{\text{st}})}{CPI \times N_{\text{inst}}} \right\rceil;$$

where N_{lookup} is the number of cycles for cache lookup latency; N_{xfer} is the number of cycles to fetch a cache line; N_{pref} and N_{st} are the numbers of cache lines to be prefetched and stored; CPI is the number of clocks per instruction; N_{inst} is the number of instructions between RI. For a cross-iteration prefetch in a loop, N_{inst} is the loop body size.

When the prefetch controller modifies the compiled code, it only needs to replace an original instruction with the prefetch instruction, as shown in Fig.4. In Fig.4(a), the code after instrumentation actually still keeps the original code in a branch. A counter (bb_counter) is incremented every time when the basic block is executed. Initially, the original code is executed. When the code is hot and the counter exceeds a threshold, the execution path will follow the instrumentation branch. In Fig.4(b), the injected prefetch instruction replaces the counter comparison instruction, and the execution path always takes the original code branch. This trick makes the compiled code modification simple. It does not require recompilation, and avoids race condition for multithreaded applications.



```
  if bb_counter < threshold
  jmp no_instrument;
  {collect information;}
no_instrument;
  {original code;}
```

(a)

```
  prefetch addr+stride
  jmp no_instrument;
  {collect information;}
no_instrument;
  {original code;}
```

(b)

Fig.4. Compiled code for instrumentation and prefetching. (a) Code after instrumentation. (b) Code after prefetch instruction injection.

3 Runtime Overhead Reductions

The key to the design of an effective prefetch engine is the runtime overhead control. We tried to reduce the runtime overhead throughout the engine, including trace buffer arrangement, stride pattern detection, prefetch instruction injection, the loads instrumentation and instrumentation elimination. We have discussed the implementation of stride pattern detection; in this section we will discuss more on the runtime overhead control in the components related to instrumentation.

3.1 Eliminate Redundant Instrumentations

Prefetch instructions are not free in terms of resource consumption. Excessive prefetchings not only take processor cycles but also put pressure on cache hierarchy. The adverse effect of excessive prefetchings becomes serious when they are used in loops that have small loop bodies or the loops performance is already bounded by the memory subsystem. In our investigations, we found Java programs have a common characteristic that the data of same object is loaded repeatedly before a store is conducted on it. The data can be different fields of the same object or adjacent elements of the same array. In this case, only one of the loads needs to be prefetched, and we only instrument that load. We discuss how we achieve that for object and array accesses separately.

3.1.1 Object Instrumentations Control

It is not straightforward for JIT compiler to analyze all the addresses dereferenced by load operations, since they are mostly runtime values. We developed a clever trick that can reduce the repetitive instrumentations largely. We leverage the JVM property that a reference can only be dereferenced when it is on the stack. We do not analyze the loads instructions directly; instead, we can find the repetitive loads by analyzing the instructions that put the references onto the stack.

In JVM, there are only two sources to get object references. One is from local variables; the other is from an object's field. We studied the ratios of the two sources in typical applications from SPECJVM98 and DaCapo, we found most references on stack are got from local variables: in 7 of the 9 applications, local variables take more than 95% as reference source (see Table 1).

Based on the data above, we design an ultra lightweight program analysis technique to detect repetitive loads. The conditions for repetitive loads are:

1) the addresses that two load instructions dereference are from the same local variable;

2) there is no store operation to the local variable between the two loads;

3) the repetitive relation is transitive. That is, if LoadA and LoadB are repetitive, LoadB and LoadC are repetitive, then LoadA and LoadC are repetitive as well.

To implement the design, JIT compiler maintains a set of load operations appearing in the code. The load operations are represented by the local variables where the references are got from. The data structure is a state array indexed by local variable indices: *load_state*[*num_of_localvar*]. The elements in the state array have only two possible states: {NOT_INSTRUMENTED, INSTRUMENTED}. The state transition is dictated by a state machine shown below in Fig.5. The compiler decides which load operations to be instrumented during its analysis on control flow graph (CFG), with following state transition rules.

Rule 1. At first all the elements are set NOT_INSTRUMENTED.

Rule 2. When code pattern (*aload, getfield*) is encountered by the compiler for local variable i, it checks *load_state*[i]. If *load_state*[i] is NOT_INSTRUMENTED, the compiler marks the *getfield* instruction as "to be instrumented". Otherwise, apply Rule 3.

Rule 3. State INSTRUMENTED does not change for pattern (*aload, getfield*), and the *getfield* instruction is a repetitive load, so no instrumentation is needed.

Rule 4. When code pattern (*astore*) is met by the compiler for local variable i, it checks *load_state*[i]. If *load_state*[i] is INSTRUMENTED, the compiler will change it to be NOT_INSTRUMENTED. It means that next time a load from local variable i requires instrumentation.

Rule 5. When two paths merge at a basic block, the state arrays from both paths are merged as the input array of current block. If NOT_INSTRUMENTED is represented by 0, and INSTRUMENTED by 1, the values of the state array is computed as $load\_state[i] = load\_state[i]_{path1} | load\_state[i]_{path2}$; if there are more entries to the block, apply the OR computation to the state arrays of all paths.

Although our algorithm for repetitive is simple, we found it is surprisingly effective in reducing excessive instrumentations.

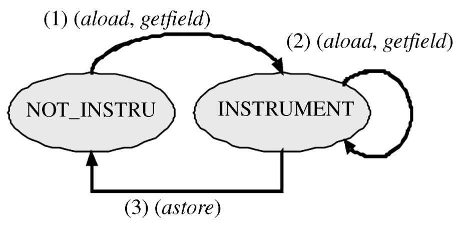


Fig.5. State machine for repetitive loads detection.

3.1.2 Array Instrumentations Control

For the loads in loops, similar analysis as above can help as well (by introducing array element index tracking), but we found loop unrolling is a simpler while effective method. Fig.6 presents an actual code example in a benchmark application. After unrolling the original loop four times, six instrumentations in every eight can be saved. Loop unrolling not only helps to reduce redundant instrumentations, but also helps to reduce the useless prefetch instructions when a stride pattern

Table 1. Source Ratios of Object References Put onto Stack

%	compress	jess	raytrace	javac	mpegaudio	bloat	fp	ps	pmd
Local Vars	97.3	96.1	96.6	96.5	95.7	95.7	95.4	78.2	76.3
Object Fields	2.7	3.9	3.4	3.5	4.3	4.3	4.6	21.8	23.7

```
for (i = 0; i < n; i + +) {
    OutBuff[OutCnt++]=buf[i];
}
```

(a)

```
for (i = 0; i < n;) {
    profile(OutBuff[OutCnt]);
    profile(buf[i]);
    OutBuff[OutCnt++]=buf[i + +];
    OutBuff[OutCnt++]=buf[i + +];
    OutBuff[OutCnt++]=buf[i + +];
    OutBuff[OutCnt++]=buf[i + +];
}
```

(b)

Fig.6. Loop unrolling to reduce excessive instrumentations. (a) Original code. (b) Code with instrumentations after unrolling.

is discovered. Here, we want to point out that loop unrolling is supported by JIT in JVM. We discuss more on the prefetch instruction injections in next subsection.

3.2 Eliminate Redundant Prefetchings

When a prefetch instruction is going to be injected into the compiled code, the prefetch controller decides whether it is redundant. When a prefetching fetches data that is not distant enough from a preceding prefetched data, it is considered redundant. The minimal distance between the data addresses of two prefetchings is the length of the cache line of the underlying processor, which is available at runtime.

There are two cases that redundant prefetchings can exist. One is that, the prefetch controller wants to inject two prefetch instructions at the same code place for two different RIs of the same load operation. The other case is that, the two prefetch instructions are for different loads. In order to avoid redundant prefetchings, the prefetch controller maintains a list of prefetch instructions injected for each method. Every time when a new stride pattern is discovered, the prefetch controller checks if the stride is distant enough from existing prefetchings. Only the distant enough prefetchings are injected.

3.3 Remove Useless Instrumentations

An instrumentation in a hot method can be replaced by a prefetching when a stride pattern is detected. But there are cases that no stride pattern exists for the instrumented hot methods. The prefetch controller tries to remove those instrumentations once identified useless.

It is easy to know a useful instrumentation, while it is not so easy to determine a useless one. This is one of the key differences between runtime profiling and offline profiling. With offline profiling, the usefulness is decided after the profiling execution passes. But this is infeasible for online profiling because there is no separate profiling execution pass.

We adopt a simple heuristic to decide useless instrumentations. As we introduced in Section 2, the prefetch controller maintains the execution frequency of an instrumented load instruction. When the prefetch controller is invoked, it processes the profile for each instruction in the trace buffer. When it finds the frequency of an instruction is higher than a threshold without a pattern discovered, the prefetch controller considers there is no pattern existing for the instruction, and will remove its instrumentation. The removal can be simply done by replacing the branch instruction in the instrumentation code to a no-op.

4 Experimental Evaluations

We evaluated the runtime prefetch engine about its effectiveness with Java programs. We present the evaluation results in this section.

We used Apache Harmony[12] for the study. The compiler part of the prefetch engine is implemented in the Jitrino component, the optimizing JIT compiler of the JVM. The prefetch controller is implemented in a separate component. We collected the data with JVM benchmarks SPECJVM98 and DaCapo[①]. The machine we used for the evaluation is a 2.6GHz Pentium 4 with 1GB RAM. The L1 cache in the processor has 8KB size, 4-way set-associativity and 64 bytes per line. The L2 cache has 256KB, 8-way and 128 bytes per line. The OS is Fedora Core 4 Linux. The parameters for "*prefetch scheduling distance*" are set to be $N_{\text{lookup}} = 60$, $N_{\text{xfer}} = 25$, $N_{\text{pref}} = N_{\text{st}} = 1$, $CPI = 1.5$.

According to the *psd* computation formula, the example codes in Fig.1 have different *psd* values. Fig.1(a) has 4, Fig.1(b) has 3, and Fig.1(c) has 1. All of their most frequently profiled stride values are 4 bytes, so their prefetch distances are $D = 16$, 12, 4 bytes. Since they are smaller than the cache line size, we can unroll the loop n times, where n is the cache line size divided by D; and then prefetch next cache line in every iteration.

[①]At the time the work was done, we could not run all the DaCapo applications, so we have only the data for the working applications.

4.1 Performance Speedups

Fig.7 shows the speed-ups we achieved with the runtime prefetch engine. The "average" bar has the arithmetic mean values. The baseline is the default execution without the prefetch engine. The bars of "direct" are the speed-ups without method inlining and loop unrolling. The "inline" bars introduce method inlining, and the "inline + unroll" bars introduce loop unrolling in addition to method inlining. In Fig.7, three applications have negative speed-up for "direct" and "inline" bars, which were caused by redundant prefetch and instrumentation. Loop unrolling can eliminate them. We can see that, with loop unrolling, the negative speed-ups become positive, and the prefetch engine can speed up the benchmarks up to 13.2%, with an average of 6.23%. Even the minimal speedup with *raytrace* is 2.1%. Without inlining or unrolling optimization, the straightforward prefetching can still get an average of 4.91% speed-up.

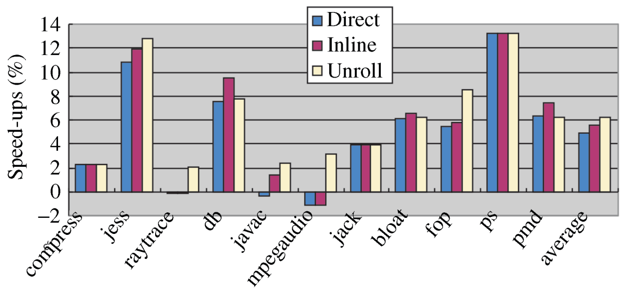


Fig.7. Performance speed-ups with runtime prefetch engine.

Loop unrolling is more effective in improving the speedups than method inlining. As we discussed in last section, loop unrolling can reduce redundant instrumentations and also help to reduce excessive prefetchings. Although it does not help to expose more prefetching opportunities, loop unrolling helps to reduce the runtime overhead. The overhead reduction contributes to the overall performance of the runtime systems.

We observed some applications get lower performance when loop unrolling is introduced. It is due to our compiler. Loop unrolling increases the method size, which leads the compiler not to inline some small methods.

During the experiment, hardware prefetch is on. Hardware prefetch does not improve Java application's performance much, there are mainly two reasons. Firstly, hardware prefetch requires two successive cache misses in the last level cache to trigger the mechanism; these two cache misses must satisfy the condition that strides of the cache misses are less than 256 bytes. Secondly, it can prefetch up to 8 simultaneously, which is not adequate for our benchmarks.

4.2 Distribution of Related Instructions (RIs)

We collected the counts of RIs in the applications as shown in Fig.8. We classified the related instructions into three categories: cross-iteration, intra-iteration, and cross-procedural. We found the counts for the three categories are similar. Interestingly, the cross-procedural part takes about 28% of total RI count. This can partly explain why Inagaki *et al.*[4] achieved only 2.0% speed-up for *jess* on Pentium 4, whereas we got 12.8%. *Jess* has 33% RIs which are cross-procedural.

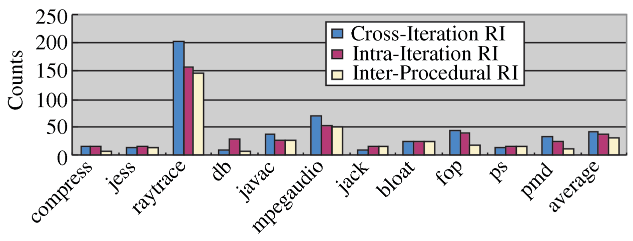


Fig.8. Counts of related instructions in three categories.

The RI counts also explain why *raytrace* can only get speed-up after loop unrolling. The reason is, *raytrace* has too many RIs, which incurs lots of runtime overhead in both profiling and prefetching. Loop unrolling can effectively reduce those overheads in *raytrace*.

4.3 Cache Misses and DTLB Misses

The benefits of prefetching come from the memory access improvement, which can be reflected by the changes in cache misses and DTLB misses. We collected the data using Intel VTune Performance Analyzer[18]. The prefetch version used in the measurement did not have the optimizations of either method inlining or loop unrolling. The metric used is "*misses per instruction*" (MPI), which is the number of misses divided by the number of retired instructions. Fig.9 shows the reduction of L2 cache misses and DTLB misses with the

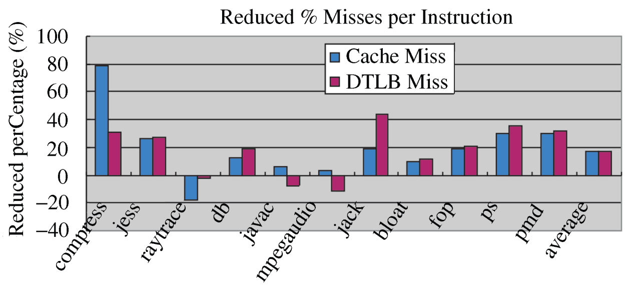


Fig.9. Cache misses and DTLB misses reduction with direct prefetching.

Table 2. Memory Dynamic Overhead of the Runtime Prefetch Engine

(KB)	compress	jess	raytrace	db	javac	mpegaudio	jack	bloat	fop	ps	pmd
Dynamic memory overhead	6	3	20	9	8	16	3	11	18	3	9

direct prefetching version, compared to the baseline no-prefetching version. We can see the average reductions for both cache misses and DTLB misses are almost the same (17%).

It is interesting to notice that, the data in Fig.9 correspond to the direct prefetching performance data in Fig.7 very well: the three applications that have negative miss reductions are the same as those with negative speed-ups in direct prefetching version. The three applications are *raytrace, javac* and *mpegaudio*. We asserted their miss reductions would become positive when loop unrolling was applied, which was confirmed in our experiments, but the data are not shown here.

4.4 Overhead of the Prefetch Engine

The runtime overhead of the prefetch engine is an important factor deciding if the engine is effective. We use total retired instruction count to represent the runtime overhead, and collect the data with two versions. One does not eliminate the redundant instrumentations, and the other does. The data are given in Fig.10.

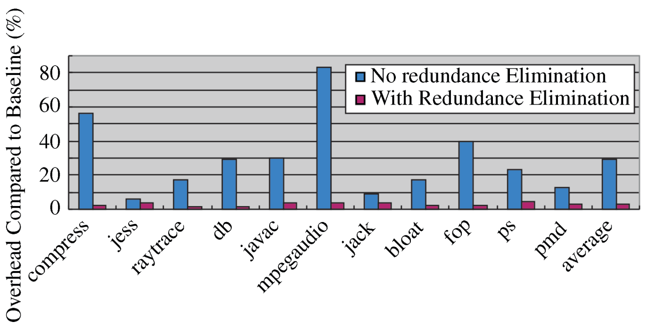


Fig.10. Runtime overhead of the runtime prefetch engine.

Without redundant instrumentation elimination, the runtime overhead is unacceptably high, with an average of 29% and minimum of 6%. As a comparison, the version with redundant instrumentation elimination has much less overhead, ranging from 1% to 4%, with an average of 3%. This means the instrumentation elimination technique is highly effective and it is critical for the prefetch engine to get performance improvement.

We also collected the memory overhead of the prefetch engine shown in Table 2. It shows the dynamic memory overhead in kilo bytes, including the memory for PC hash table and RI stride distribution lists. The static memory overhead of 48KB trace buffer is not counted. The maximal dynamic memory overhead is only 20KB, therefore we believe the memory overhead of the engine is actually negligible.

4.5 Distributions of Stride Values

Since the stride detection mechanism in the engine can only discover the stride values in range of -40 through 40 with interval of 4. It is interesting to know if the range will miss some stride pattern opportunities. We collected all the stride values in the applications. We found the occurrences of the values out of the range $[-40, 40]$ are almost negligible. This suggests we did not miss any important opportunities. Fig.11 shows the distributions of the stride values in the range of $[-80, 80]$. It clearly shows that stride values between -12 and 12 take most (i.e., 98.3%) of all the stride occurrences.

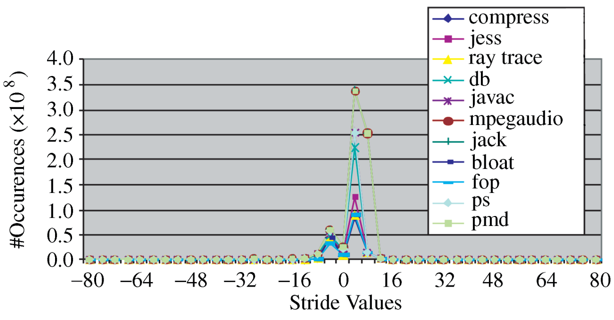


Fig.11. Distributions of stride values.

5 Related Work

Stride prefetching has been used widely in modern software systems to hide cache miss latencies. We classify the existing profile-guided techniques into two categories: offline profiling and online profiling.

Based on offline profiling, Robert *et al.*[2] proposed a technique which that supports post-link stride prefetching. Their technique does not require source code availability, can instrument the binary and profile with training input set, and then uses the strides discovered to insert prefetch instructions into the executable directly. Wu *et al.*[1] developed a compiler technique for efficient offline profiling and stride prefetching for irregular programs. The compiler can inject different prefetching code sequence for different striding scenarios. As a comparison, our work conducts online profiling and can discover striding patterns for arbitrary kinds of load operations. At the same time, our work does not require recompilation for prefetch injection and instrumentation removal.

Based on online profiling, Adl-Tabatabai *et al.*[5] developed a software prefetch engine for managed runtime system. Their work depends on hardware monitoring support to collect cache miss information, leverages garbage collector to maintain the stride distances between target objects, and then lets JIT compiler inject prefetch instructions. Inagaki *et al.*[4] proposed an ultra-lightweight profiling technique called object inspection. During the compilation of a method, the dynamic compiler gathers the profile information by partially interpreting the method using the actual values of parameters. Their technique can discover cross-iteration and intra-iteration RIs. As a comparison, our work is a pure software approach, and conducts profiling during real execution. We do not recompile the code, but we can eliminate the useless instrumentations at runtime.

Besides prefetching techniques, object reordering or data rearrangement can also improve the memory access performance[19−21]. We believe software prefetching and object reordering can be used together, but the striding patterns might be changed after object reordering, which needs further study.

6 Conclusions and Future Work

In this paper, we propose a runtime prefetch engine for managed runtime system. The engine instruments the program into JIT compiler for load address profiling, detects the stride patterns periodically at runtime. When a stride pattern is discovered, the engine injects prefetch instruction and removes the instrumentation effect. The key points in the engine design are the tradeoffs between prefetching accuracy and runtime overhead. In order to reduce the runtime overhead, we developed techniques to remove the redundant instrumentations, control the prefetch instruction injections, and disable the useless instrumentation. Our pattern detection engine is light-weighted that we use a sliding window to filter the trace information for runtime analysis, and we use a stride frequency array that covers stride range between -40 and 40. Finally, the experimental evaluations show that the prefetch engine can speed up SPECJVM98 and DaCapo benchmarks up to 13.2%, with an average of 6.23%. At the same time, the maximal runtime overhead is less than 4%, and the memory overhead is negligible.

Our next step is to combine the object reordering technique with stride prefetching, which needs close investigations into garbage collection algorithms. Another area we are looking at is to use hardware performance monitors to collection trace information, so as to reduce the runtime overhead even more. Compiler analysis would be important in this case to relate the hardware event with program semantics.

References

[1] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. *SIGPLAN Not.*, 2002, 37(5): 210–221.

[2] Robert Muth, Harish Patil, Richard Weiss, P Geoffrey Lowney, Robert Cohn. Profile-guided post-link stride prefetching. In *Proc. 16th Int. Supercomputing*, New York, USA, June 22–26, 2002, pp.167–178.

[3] Brendon D Cahoon. Effective compile-time analysis for data prefetching in Java [Dissertation]. University of Massachusetts, Amherst, 2002.

[4] Inagaki T, Onodera T, Komatsu H, Nakatani T. Stride prefetching by dynamically inspecting objects. In *Proc. the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, California, USA, June 9–11, 2003, pp.269–277.

[5] Adl-Tabatabai A, Hudson R L, Serrano M J, Subramoney S. Prefetch injection based on hardware monitoring and object metadata. In *Proc. the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, Washington DC, USA, June 9–11, 2004, pp.267–276.

[6] Vanderwiel S P, Lilja D J. Data prefetch mechanisms. *ACM Comput. Surv.*, Jun. 2000, 32(2): 174–199.

[7] Bernstein D, Cohen D, Freund A. Compiler techniques for data prefetching on the PowerPC. In *Proc. the IFIP Wg10.3 Working Conference on Parallel Architectures and Compilation Techniques*, Limassol, Cyprus, June 27–29, 1995, pp.19–26.

[8] Santhanam V, Gornish E H, Hsu W. Data prefetching on the HP PA-8000. In *Proc. the 24th Annual International Symposium on Computer Architecture*, Denver, Colorado, United States, June 1–4, 1997, pp.264–273.

[9] Hölzle U, Ungar D. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.*, July 1996, 18(4): 355–400.

[10] Suganuma T, Yasue T, Kawahito M, Komatsu H, Nakatani T. A dynamic optimization framework for a Java just-in-time compiler. In *Proc. the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, Tampa Bay, FL, USA, October 14–18, 2001, pp.180–195.

[11] Arnold M, Ryder B G. A framework for reducing the cost of instrumented code. In *Proc. the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, Snowbird, Utah, United States, New York, pp.168–179.

[12] Apache Harmony project. http://harmony.apache.org/.

[13] Standand Performance Evaluation Corporation (SPEC). JVM client98 (SPECjvm98).

[14] The DaCapo benchmark suite. http://dacapobench.org/.

[15] Intel Corporation. Intel(R) architecture software developer's manual, Volume 2: Instruction set reference manual. http://download.intel.com/design/intarch/manuals.

[16] Shuf Y, Serrano M J, Gupta M, Singh J P. Characterizing the memory behavior of Java workloads: A structured view and opportunities for optimizations. *SIGMETRICS Perform. Eval. Rev.*, June 2001, pp.194–205.

[17] Hosking A L, Moss J E, Stefanovic D. A comparative performance evaluation of write barrier implementation. In *Proc. Object-Oriented Programming Systems, Languages, and Applications*, Vancouver, British Columbia, Canada, October 18–22, 1992, pp.92–109.

[18] Intel Corporation. VTune performance analyzer. http://www. intel.com/cd/software/products/apac/zho/vtune/275878.htm.

[19] Chen W, Bhansali S, Chilimbi T, Gao X, Chuang W. Profile-guided proactive garbage collection for locality optimization. *SIGPLAN Not.*, Jun. 2006, pp.332–340.

[20] Chilimbi T M, Davidson B, Larus J R. Cache-conscious structure definition. In *Proc. the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, United States, May 1–4, 1999, pp.13–24.

[21] Huang X, Blackburn S M, McKinley K S, Moss J B, Wang Z, Cheng P. The garbage collection advantage: Improving program locality. *SIGPLAN Not.*, Oct. 2004, 39(10): 69–80.

Qiong Zou received her B.S. degree in computer science from University of Science and Technology of China. She is currently a Ph.D. candidate of the University of Science and Technology of China. Her research interests include system architecture, dynamic compilation and Java virtual machine.

Xiao-Feng Li is a VM architect in Enterprise Solutions Software Division of Intel Software and Solutions Group, where he leads a team developing managed runtime technologies. Before he joined Intel in 2001, Li worked in Nokia Research Center. Li is active in open source community, developing open source software, and giving technical lectures and trainings frequently. His research interests are programming language design and implementations. Currently his research is focused on runtime technology and interactions with underlying platforms.

Long-Bing Zhang received the Ph.D. degree from the University of Science and Technology of China in 2002. And he completed the postdoctoral work in Institute of Computing Technology, Chinese Academy of Sciences, in 2004. He is now an associate researcher in Institute of Computing Technology, Chinese Academy of Sciences. His research interests include microprocessor design, system architecture, and cluster computing.