Dynamic Hot Data Stream Prefetching for General-Purpose Programs

Trishul M. Chilimbi
Microsoft Research
One Microsoft Way
Redmond, WA 98052
trishulc@microsoft.com

Martin Hirzel
Computer Science Dept.
University of Colorado
Boulder, CO 80309
martin.hirzel@colorado.edu

ABSTRACT

Prefetching data ahead of use has the potential to tolerate the growing processor-memory performance gap by overlapping long latency memory accesses with useful computation. While sophisticated prefetching techniques have been automated for limited domains, such as scientific codes that access dense arrays in loop nests, a similar level of success has eluded general-purpose programs, especially pointer-chasing codes written in languages such as C and C++.

We address this problem by describing, implementing and evaluating a dynamic prefetching scheme. Our technique runs on stock hardware, is completely automatic, and works for general-purpose programs, including pointer-chasing codes written in weakly-typed languages, such as C and C++. It operates in three phases. First, the profiling phase gathers a temporal data reference profile from a running program with low-overhead. Next, the profiling is turned off and a fast analysis algorithm extracts hot data streams, which are data reference sequences that frequently repeat in the same order, from the temporal profile. Then, the system dynamically injects code at appropriate program points to detect and prefetch these hot data streams. Finally, the process enters the hibernation phase where no profiling or analysis is performed, and the program continues to execute with the added prefetch instructions. At the end of the hibernation phase, the program is de-optimized to remove the inserted checks and prefetch instructions, and control returns to the profiling phase. For long-running programs, this profile, analyze and optimize, hibernate, cycle will repeat multiple times. Our initial results from applying dynamic prefetching are promising, indicating overall execution time improvements of 5–19% for several memory-performance-limited SPECint2000 benchmarks running their largest (*ref)* inputs.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors – *code generation, optimization, run-time environments.*

General Terms

Measurement, Performance.

Keywords

dynamic profiling, temporal profiling, data reference profiling, dynamic optimization, memory performance optimization, prefetching.

PLDI'02, June 17-19, 2002, Berlin, Germany.

1. INTRODUCTION

The demise of Moore's law has been greatly exaggerated and processor speed continues to double every 18 months. By comparison, memory speed has been increasing at the relatively glacial rate of 10% per year. The unfortunate, though inevitable consequence of these trends is a rapidly growing processor-memory performance gap. Computer architects have tried to mitigate the performance impact of this imbalance with small high-speed cache memories that store recently accessed data. This solution is effective only if most of the data referenced by a program is available in the cache. Unfortunately, many general-purpose programs, which use dynamic, pointer-based data structures, often suffer from high cache miss rates, and are limited by their memory system performance.

Prefetching data ahead of use has the potential to tolerate this processor-memory performance gap by overlapping long latency memory accesses with useful computation. Successful prefetching is accurate—correctly anticipating the data objects that will be accessed in the future—and timely—fetching the data early enough so that it is available in the cache when required. Sophisticated automatic prefetching techniques have been developed for scientific codes that access dense arrays in tightly nested loops (for e.g., [24]). They rely on static compiler analyses to predict the program's data accesses and insert prefetch instructions at appropriate program points. However, the reference pattern of general-purpose programs, which use dynamic, pointer-based data structures, is much more complex, and the same techniques do not apply.

If static analyses cannot predict the access patterns of general-purpose programs, perhaps program data reference profiles may suffice. Recent research has shown that programs possess a small number of hot data streams, which are data reference sequences that frequently repeat in the same order, and these account for around 90% of program references and more than 80% of cache misses [8, 28]. These hot data streams can be prefetched accurately since they repeat frequently in the same order and thus are predictable. They are long enough (15–20 object references on average) so that they can be prefetched ahead of use in a timely manner.

In prior work, Chilimbi instrumented a program to collect the trace of its data memory references; then used a compression algorithm called Sequitur to process the trace off-line and extract hot data streams [8]. These hot data streams have been shown to be fairly stable across program inputs and could serve as the basis for an off-line static prefetching scheme [10]. On the other hand, for programs with distinct phase behavior, a dynamic prefetching scheme that adapts to program phase transitions may perform better. In this paper, we explore a dynamic software prefetching scheme and leave a comparison with static prefetching for future work.

A dynamic prefetching scheme must be able to detect hot data

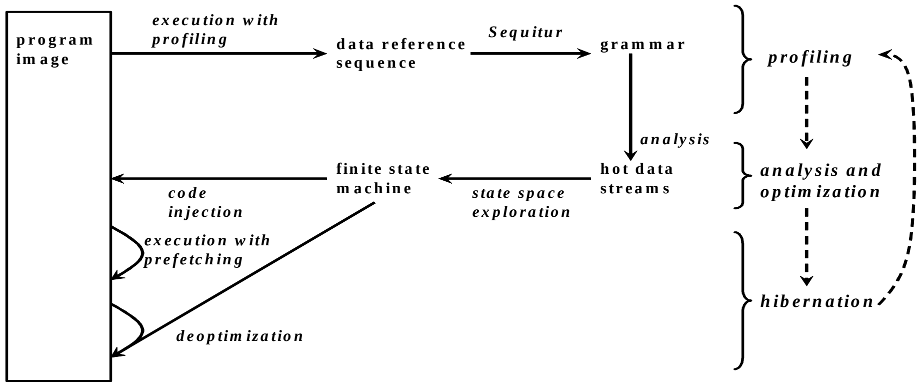

Figure 1. Dynamic prefetching overview

streams online with little overhead. This paper describes a dynamic framework for online detection of hot data streams and demonstrates that this can be accomplished with extremely low-overhead. Rather than collect the trace of all data references, our dynamic framework uses sampling to collect a temporal data reference profile. Unlike conventional sampling, we sample data reference bursts, which are short sequences of consecutive data references. The framework uses Sequitur to process the trace online, and a novel algorithm for fast detection of hot data streams from the temporal profile data.

The hot data streams consist of a sequence of <*pc, addr*> pairs. Our hot data stream analysis is configured to only detect streams that are sufficiently long to justify prefetching (i.e., containing more than ten unique references). Once these streams have been detected, our prefetching engine dynamically injects checks in the program to match stream prefixes, followed by prefetch instructions for the remaining stream addresses. For example, given a hot data stream *abacdce*, once the addresses *a.addr, b.addr, a.addr* are detected by checks inserted at *a.pc, b.pc, a.pc* respectively, prefetches are issued for the addresses, *c.addr, d.addr, e.addr.* The hot data stream prefix length that must match before prefetching is initiated needs to be set carefully. A prefix that is too short may hurt prefetching accuracy, and too large a prefix reduces the prefetching opportunity and incurs additional stream matching overhead.

Conceptually, one can think of the prefix-matching mechanism for a hot data stream as corresponding to a deterministic finite state machine (DFSM), where the states correspond to possible stream prefixes, and transitions are implemented by inserted prefix-match checks. To avoid redundant checks, and efficiently orchestrate matches for all hot data streams, our prefetching engine constructs a single DFSM that keeps track of matching prefixes for all hot data streams simultaneously (see Section 3.1). The prefetching engine uses a dynamic implementation of Vulcan [32] (a binary editing tool for the x86, similar to ATOM [31]), to insert checks into the running program that implement the stream prefix matching DFSM. In addition, it adds prefetch instructions that target the remaining data stream addresses, on successful stream prefix matches.

Figure 1 provides an overview of our dynamic prefetching process that operates in three phases—profiling, analysis and optimization, and hibernation. First, the profiling phase collects a temporal data reference profile from a running program with low-overhead. This is accomplished using bursty tracing [15], which is an extension of Arnold and Ryder's low-overhead profiling technique [3]. The Sequitur compression algorithm incrementally builds an online grammar representation of the traced data references. Once sufficient data references have been traced, profiling is turned off and the analysis and optimization phase begins. A fast analysis algorithm extracts hot data streams from the Sequitur grammar representation. The prefetching engine builds a stream prefix matching DFSM for these hot data streams, and dynamically injects checks at appropriate program points to detect and prefetch these hot data streams. Finally, the process enters the hibernation phase where no profiling or analysis is performed, and the program continues to execute with the added prefetch instructions. At the end of the hibernation phase, the program is de-optimized to remove the inserted checks and prefetch instructions, and control returns to the profiling phase. For long-running programs this profile, analyze and optimize, hibernate cycle will repeat multiple times.

The paper makes the following contributions:

- It presents a dynamic, low-overhead framework for detecting hot data streams (see Section 2).
- It describes an automatic, dynamic prefetching scheme that works for general-purpose programs. The prefetching is driven by the hot data streams supplied by the online profiling and analysis framework (see Section 3).
- It presents empirical evidence that dynamic prefetching is effective, producing overall execution time improvements of 5–19% for several memory performance limited SPECint2000 benchmarks (see Section 4).

2. DYNAMIC DATA REFERENCE PROFILING AND ANALYSIS

This section discusses our online, low-overhead framework for detecting hot data streams. The framework first collects a temporal data reference profile with low-overhead, and then uses a fast analysis algorithm to extract hot data streams from this temporal profile.

2.1 Bursty Tracing Framework for Low-Overhead Temporal Profiling

A data reference *r* is a load or store of a particular address, represented as a pair (*r.pc*,*r.addr*). The sequence of all data references during execution is the data reference trace. A temporal data reference profile captures not only the frequencies of individual data references in the trace, but also temporal relationships between them. For example, it would distinguish the traces *cdeabcdeabfg* and *abcdefabcdeg*, even though all data

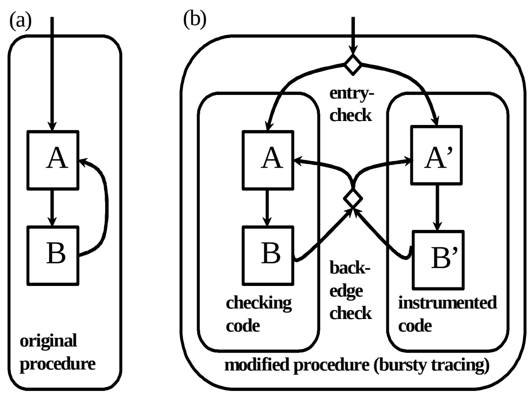

Figure 2. Instrumentation for low-overhead temporal profiling

references have the same frequencies in both of them. In the second trace, the subsequence *abcde* is a hot data stream and presents a prefetching opportunity.

Our framework must collect a temporal data reference profile with low overhead, because the slow-down from profiling has to be recovered by the speed-up from optimization. A common way to reduce the overhead of profiling is sampling: instead of recording all data references, sample a small, but representative fraction of them. Our profiler obtains a temporal profile with low overhead by sampling bursts of data references, which are subsequences of the reference trace.

We use the bursty tracing profiling framework [15], which is an extension of the Arnold-Ryder framework [3]. The code of each procedure is duplicated (see Figure 2). Both versions of the code contain the original instructions, but only one version is instrumented to also profile data references. Both versions of the code periodically transfer control to checks at procedure entries or loop back-edges. The checks use a pair of counters, *nCheck* and *nInstr*, to decide in which version of the code execution should continue.

At startup, *nCheck* is $nCheck_0$ and *nInstr* is zero. Most of the time, the checking code is executed, and *nCheck* is decremented at every check. When it reaches zero, *nInstr* is initialized with $nInstr_0$ (where $nInstr_0 \ll nCheck_0$) and the check transfers control to the instrumented code. While in the instrumented code, *nInstr* is decremented at every check. When it reaches zero, *nCheck* is initialized with $nCheck_0$ and control returns back to the checking code.

The bursty tracing profiling framework does not require operating system or hardware support and is deterministic. We implemented it using Vulcan [32], (an executable-editing tool for x86, similar to ATOM [31]), and hence it does not require access to program source code or recompilation. The profiling overhead is easy to control: there is a basic overhead for the checks, and beyond that the overhead is proportional to the sampling rate $r = nInstr_0/(nCheck_0+nInstr_0)$. Via $nCheck_0$ and $nInstr_0$, we can freely chose the burst length and the sampling rate.

2.2 Extensions for Online Optimization

The counters $nCheck_0$ and $nInstr_0$ of the bursty tracing profiling framework control its overhead and the amount of profiling information it generates. For example, setting $nCheck_0$ to 9900 and $nInstr_0$ to 100 results in a sampling rate of 100/10000=1% and a burst length of 100 dynamic checks. We term $nCheck_0+nInstr_0$ dynamic checks a burst-period (see Figure 3).

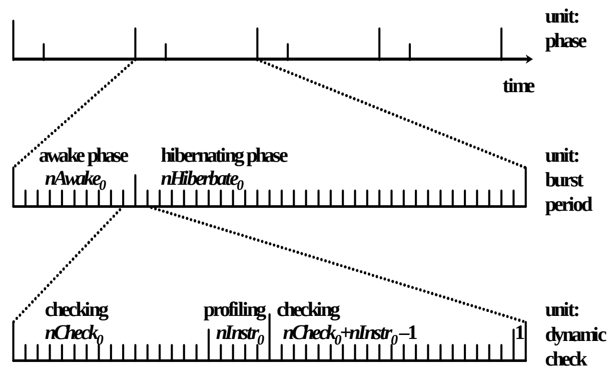

Figure 3. Profiling timeline.

For online optimization, we extended the bursty tracing framework to alternate between two phases, awake and hibernating. The profiler starts out awake and stays that way for $nAwake_0$ burst-periods, yielding $nAwake_0*nInstr_0$ checks's worth of traced data references. Then, the online optimizer performs the optimizations; after that, the profiler hibernates. This is done by setting $nCheck_0$ to $nCheck_0+nInstr_0 - 1$ and $nInstr_0$ to 1 for the next $nHibernate_0$ burst-periods, where $nHibernate_0 \gg nAwake_0$. When the hibernating phase is over, the profiler is woken up by resetting $nCheck_0$ and $nInstr_0$ to their old values (see Figure 3).

While the profiler is hibernating, it traces next to no data references and hence incurs only the basic overhead of executing checks. We designed the hibernation extension so that burst-periods correspond to the same time (measured in executed checks) in either phase (see Figure 3). This makes it easy to control the relative length of the awake and hibernating phases using the counters, $nAwake_0$ and $nHibernate_0$. Note that with our extension, bursty tracing is still deterministic. Since our optimization is also deterministic, executions of deterministic benchmarks are repeatable, which helps testing. When $nHibernate_0 \gg nAwake_0 \gg 1$ and $nChecking_0 \gg nInstr_0 \gg 1$ the sampling rate approximates to

$$(nAwake_0*nInstr_0)/((nAwake_0+nHibernate_0)*(nInstr_0+nCheck_0)).$$

2.3 Fast Hot Data Stream Detection

Bursty tracing collects a temporal data reference profile. This must

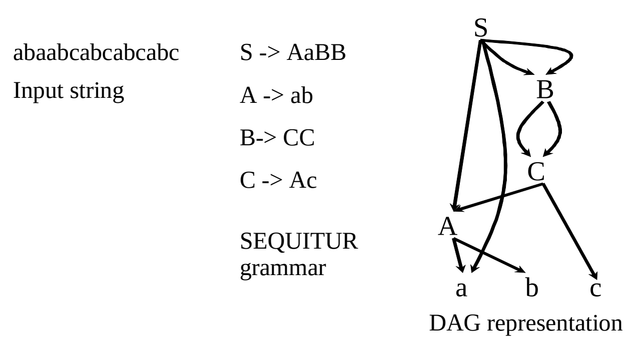

Figure 4. Sequitur grammar for w=abaabcabcabcabc.

```
//find reverse post-order numbering for non-terminals
    int next = nRules;
    function doNumbering = lambda(NonTerminal A){
      if(have not yet visited A){
        for(each child B of A)
          doNumbering(B);
        next--;
        A.index = next;
      }
    }
    doNumbering(S);
//find uses for non-terminals, initialize coldUses to uses
    for(each non-terminal A)
      A.uses = A.coldUses = 0;
    S.uses = S.coldUses = 1;
    for(each non-terminal A, ascending order of A.index)
      for(each child B or A)
        B.uses = B.coldUses = (B.uses + A.uses);
    //find hot non-terminals
    for(each non-terminal A, ascending order of A.index){
        A.heat = w_A.length * A.coldUses;
        fHot = minLen<=A.length<=maxLen && H<=A.heat;
        if(fHot)
          reportHotDataStream(w_A, A.heat);
        subtract = fHot ? A.uses : (A.uses-A.coldUses);
        for(each child B of A)
          B.coldUses = B.coldUses - subtract;
    }
```

Figure 5. Algorithm for fast approximation of hot data streams.

be analyzed to find hot data streams. Our online profiling and analysis framework first uses the Sequitur algorithm [23] to compress the profile and infer its hierarchical structure. Each observed data reference can be viewed as a symbol, and the concatenation of the profiled bursts as a string *w* of symbols. Sequitur constructs a context-free grammar for the language *{w}* consisting of exactly one word, the string *w*. Sequitur runs in time O(*w.length*). It is incremental (we can append one symbol at a time) and deterministic. The grammar is a compressed representation of the trace, it is unambiguous and acyclic in the sense that no non-terminal directly or indirectly defines itself. Figure 4 shows a Sequitur grammar for *w*=*abaabcabcabcabc*, and its representation as a multi-dag.

Before describing our online analysis for finding hot data streams from this Sequitur grammar, we review some definitions from [8]. A hot data stream is a data reference subsequence whose regularity magnitude exceeds a predetermined heat threshold, *H*. Given a data reference subsequence *v*, we define its regularity magnitude as *v.heat* = *v.length***v.frequency*, where *v.frequency* is the number of non-overlapping occurrences of *v* in the trace. Larus describes an algorithm for finding a set of hot data streams from a Sequitur grammar [21]; we use a faster, less precise algorithm that relies more heavily on the ability of Sequitur to infer hierarchical structure.

Our analysis algorithm uses the observation that each non-terminal *A* of a Sequitur grammar generates a language $L(A) = \{w_A\}$ with just one word w_A. We define the regularity magnitude of a non-terminal *A* as $A.heat = w_A.length*A.coldUses$, where *A.coldUses* is the number of times *A* occurs in the (unique) parse tree of the complete grammar, not counting occurrences in sub-trees belonging to hot non-terminals other than *A*. A non-terminal *A* is hot iff *minLen* <= *A.length* <= *maxLen* and *H* <= *A.heat*, where *H* is the predetermined heat threshold. The result of the analysis is the set $\{w_A \mid A \text{ is a hot non-terminal}\}$ of hot data streams.

Figure 5 shows pseudo-code for the analysis. We call a non-terminal *B*, a child of another non-terminal *A*, if it occurs on the right-hand side of the grammar rule for *A*. We assume that we already have $w_A.length$ for each non-terminal *A*; this is easy to maintain in Sequitur. The analysis first numbers the non-terminals such that whenever *B* is a child of *A*, we have *A.index* < *B.index*. This important property guarantees that in the rest of the algorithm, we never visit a non-terminal before having visited all its predecessors. Then, the algorithm finds how often each non-terminal is used in the parse-tree of the grammar. Finally, it finds hot non-terminals such that a non-terminal is only considered hot if it accounts for enough of the trace on its own, where it is not part of the expansion of other non-terminals. The running time of the algorithm is linear in the size of the grammar.

Consider, for example, the grammar shown in Figure 4. Assume the heat threshold, *H* = 8, and the length restrictions are *minLen* = 2, *maxLen* = 7. The values computed by the analysis are shown in Figure 6 and Table 1. Note that the non-terminal *C* is completely subsumed by the hot non-terminal *B* and therefore not considered hot. Note that even though the non-terminal *A* also appears outside of the parse trees of hot non-terminals, its regularity magnitude *A.heat* = 2 does not exceed the heat threshold *H*. In this example, we would find just one hot data stream w_B = *abcabc* with heat 12 that accounts for 12/15=80% of all data references.

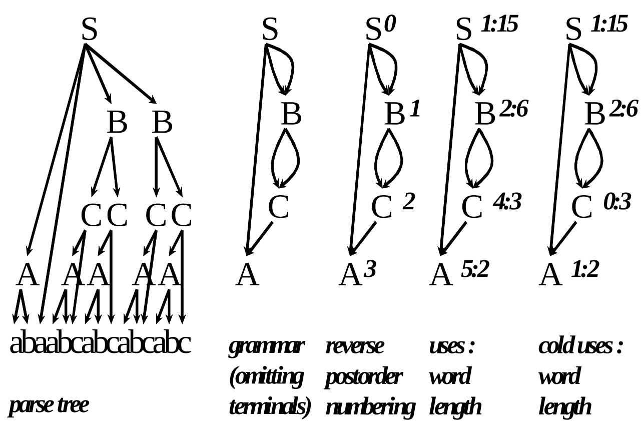

Figure 6. Hot data stream analysis example.

X	Child	Length	Index	Use	cold-Use	Heat	Report?
S	A,B ,B	15	0	1	1	15	no, start
A	-	2	3	5	1	2	no, cold
B	C,C	6	1	2	2	12	yes
C	A	3	2	4	0	0	no, cold

Table 1: Computed values for hot data stream analysis.

```
a.pc: if(accessing a.addr){
        if(v.seen == 2){
           v.seen = 3;
           prefetch c.addr,a.addr,d.addr,e.addr;
        }else{
           v.seen = 1;
        }
      }else{
        v.seen = 0;
      }
b.pc: if(accessing b.addr)
        if(v.seen == 1)
           v.seen = 2;
        else
           v.seen = 0;
      else
         v.seen = 0;
```

Figure 7. Inserted prefetching code for stream abacadae.

2.4 Discussion

Our online profiling and analysis framework implementation batches and sends traced data references to Sequitur, as soon as they are collected, rather than at the end of the awake phase. This is possible since Sequitur constructs the grammar representation incrementally. During the hibernation phase, our online profiler enters the instrumented code once per burst period (see Figure 3). These data references traced during hibernation are ignored by Sequitur to avoid trace contamination and unnecessary additional trace analysis overhead.

3. DYNAMIC PREFETCHING

Prior work has shown that the data references of programs have a high degree of regularity [8]. A data reference *r* is a load or store of a particular address, represented as a pair (*r.pc*,*r.addr*). Most data references of a program take place in only a few hot data streams, which are sequences of data references that repeat frequently, and these account for most of the program's cache misses [8]. For example, if *abacadae* is a hot data stream, then the program often performs a data access at *a.pc* from address *a.addr*, followed by a data access at *b.pc* from address *b.addr*, and so on.

Our prefetching optimizer matches hot data stream prefixes, and then issues prefetches for the remaining data stream addresses. For example, given the hot data stream *abacadae*, when the optimizer detects the data references *aba*, it prefetches from the addresses *c.addr,a.addr,d.addr,e.addr*. Ideally, the data from these addresses will be cache resident by the time the data references *cadae* take place, avoiding cache misses and speeding up the program.

Figure 1 shows an overview of our optimizer. It profiles the program to find hot data streams. When it has collected enough profiling information, it stops profiling and injects code for detecting prefixes and prefetching suffixes of hot data streams. Then it continues running the optimized program. For long-running applications, it may repeat these steps later. We use dynamic Vulcan [32], which is an executable editing tool similar to ATOM [31], to edit the binary of the currently executing program.

3.1 Generating Detection and Prefetching Code

After the profiling and analysis phase finds the hot data streams, the optimizer must match their prefixes and prefetch their suffixes. The optimizer uses a fixed constant *headLen* to divide each hot data stream $v = v_1 v_2 ... v_{\{v.length\}}$ into a head $v.head = v_1 v_2 ... v_{headLen}$ and a tail $v.tail = v_{\{headLen+1\}} v_{\{headLen+2\}} ... v_{\{v.length\}}$. When it detects the data references of *v.head*, it prefetches from the addresses of *v.tail*.

Consider how we might match and prefetch when *headLen* = 3 and there is only one hot data stream, *v* = *abacadae*. The detection/matching code makes use of a counter *v.seen*, that keeps track of how much of *v.head* has been matched. When *v.seen* = 0 nothing has been matched, when *v.seen* = 1, we have a partial match *a*, when *v.seen* = 2, we have a partial match *ab*, and when *v.seen* = 3 we have a complete match for *v.head* = *abc*, and prefetch from the addresses in *v.tail*, i.e. from addresses *c.addr, a.addr, d.addr, e.addr*. To drive *v.seen*, we need to insert detection and prefetching code at the pc's of *v.head* that make comparisons to the addresses of *v.head* and the variable *v.seen*. Figure 7 shows pseudo-code for this.

Note in Figure 7 that we have exploited the fact that the same symbol a occurs multiple times in *v.head* = *aba*. Also note that we treat the cases of initial, failed, and complete matches specially. The initial match of data reference a works regardless of how much of *v.head* we have seen. A failed match resets *v.seen* to 0. A

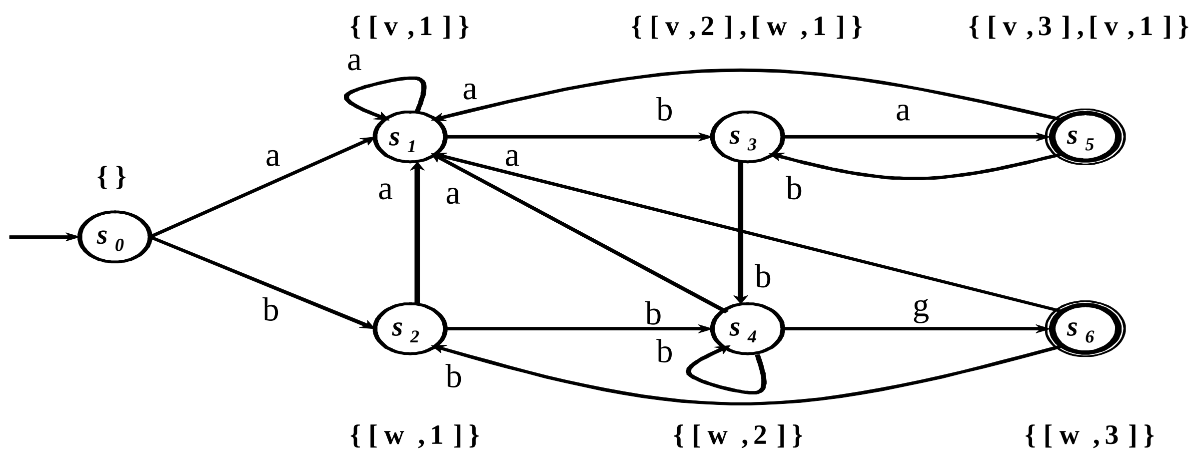

Figure 8. Prefix-matching DFSM for hot data streams v=abacadae and w=bbghij.

complete match, besides driving *v.seen*, prefetches the addresses in *v.tail*. Finally, note that it is possible that *a.pc* == *b.pc*, in which case the *if(accessing b.addr)* clause would appear in *a.pc*'s instrumentation.

Now that we know how to detect the head and prefetch the tail of a single hot data stream, there is a straight-forward way to do it for multiple hot data streams. We could introduce one variable *v.seen*, for each hot data stream *v*, and inject the code independently. While this simple approach works, it may lead to a lot of redundant work. Consider, for example, the hot data streams *v* = *abacadae* and *w* = *bbghij*. When *v.seen* == 2, we know that *w.seen* == 1, so we could save some work by combining the matching of *v* and *w*. This even holds inside one hot data stream: when *w.seen* == 2 and we observe another b, we should keep *w.seen* = 2.

Conceptually, each hot data stream *v* corresponds to a deterministic finite state machine (DFSM) *v.dfsm*, where the states are represented by *v.seen* and the detection code implements the transitions. Instead of driving one DFSM per hot data stream, we would like to drive just one DFSM that keeps track of matching for all hot data streams simultaneously. By incurring the one-time cost of constructing the DFSM, we make the frequent detection and prefetching of hot data streams faster.

Figure 8 illustrates a prefix-matching DFSM that simultaneously tracks hot data streams *abacadae* and *bbghij*. Before we describe how to come up with a DFSM that matches all hot data streams simultaneously, let us consider how we would generate code to drive it. Without loss of generality, let *S = {0,...,m}* be the set of states and let *A* be the set of data references (symbols) that appear in prefixes of hot data streams. The transition function *d:S*A-->S* indicates that when you are in a state *s* and observe the data reference *a*, you drive the state to *s' = d(s,a)*. In other words, *a.pc* has instrumentation of the form

a.pc: if((accessing a.addr) && (state==s))

state = s';

Additionally, some states *s* in *S* would be annotated with prefetches *s.prefetches*, for the suffixes of the streams that have been completely matched when state *s* is reached. Thus, the instrumentation would become

a.pc: if((accessing a.addr) && (state==s)){

state = s';

prefetch s'.prefetches;

}

We again treat the cases of initial, failed, and complete matches specially as indicated in Figure 7. Note that besides combining matches for the same address, but different states under the same outer if branch, we can sort the if-branches in such a way that more likely cases come first. This further reduces the work for detecting prefixes of hot data streams.

Now let us examine how to construct a DFSM that matches all hot data streams simultaneously. A state is a set of state elements, where state element *e* is a pair of a hot data stream *e.hotDataStream* and an integer *e.seen*. If the current state is *s={[v,2],[w,1]}* this means the prefix matcher has seen the first two data accesses of the hot data stream *v*, and the first data access of hot data stream *w*, and no data accesses of any other hot data streams. State s_0 = *{}* is the start state where nothing has been matched.

```
add {} to the workList;
while(!workList.isEmpty){
  take state s out of workList;
  function addTransition = lambda(Symbol a){
    if(s doesn't yet have a transition for a){
      s' = {[v,n+1] | n<headLen && [v,n] in s &&
                 a==v_{n+1}} union {[w,1] | a==w_1}
      if(s' doesn't yet exist){
        add s' to the states of the DFSM;
        add s' to the workList;
      }
      if(s' != {})
        introduce the transition (a,s') for s;
    }
  }
  for(each state element e in s)
    if(e.seen < headLen)
      addTransition(e.hotDataStream_e.seen+1);
  for(each symbol a for which there
             exists a hot data stream v with v_1==a)
    addTransition(a);
}
```

Figure 9. Algorithm for prefetching FSM construction.

Let *s* be a state and *a* be a data reference. The transition function *d:S*A*-->A yields a target state (set of state elements) as follows:

$$d(s,a) = \{[v,n+1] \mid n<headLen \;\&\&\; [v,n] \in s \;\&\&\; a==v_{\{n+1\}}\} \;\; union \;\; \{[w,1] \mid a==w_1\}$$

We construct the DFSM with a lazy work-list algorithm starting from s_0. We represent the DFSM as a directed graph, where the nodes are reachable states and a transition *d(a,s)* is stored as an edge from *s* to *d(a,s)* labelled with *a*. We do not explicitly represent any edges to the start state. Figure 9 shows the pseudo-code. Let *n* be the number of hot data streams, and *n* <= 100 if *H* is set such that each hot data stream covers at least 1% of the profile. Then there are *headLen*n* different state elements and thus up to $2^{(headLen*n)}$=O(2^n) different states. We have never observed this exponential blow-up; we usually find close to *headLen*n+1* states.

3.2 Injecting Detection and Prefetching Code

Our online optimizer uses dynamic Vulcan to inject the detection and prefetching code into the running benchmark image [32]. Dynamic Vulcan stops all running program threads while binary modifications are in progress and restarts them on completion. For every procedure that contains one or more pc's for which the optimizer wants to inject code, it does the following. First, it makes a copy of the procedure. Second, it injects the code into the copy. Third, it overwrites the first instruction of the original with an unconditional jump to the copy. When the optimizer wants to deoptimize later, it need only remove those jumps.

Note that we do not patch any pointers to the original code of optimized procedures in the data of the program. In particular, the return addresses on the stack still refer to the original procedures. Hence, we will return to original procedures for at most as many times as there were activation records on the stack at optimization time. This is safe, but may lead to a few missed

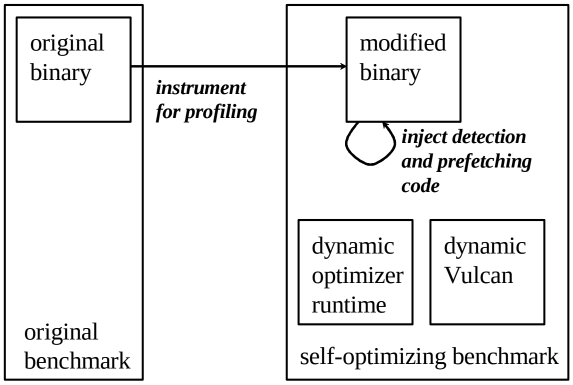

Figure 10. Dynamic Injection of Prefetching Code.

prefetching opportunities.

Figure 10 shows how our system uses Vulcan. Before execution, static Vulcan modifies the x86 binary of the benchmark to implement the bursty tracing framework from Section 2.1. The resulting modified binary is linked with the runtime system of our dynamic optimizer, which includes code for the algorithms described in Section 2.3 and Section 3.1.

4. EXPERIMENTAL EVALUATION

This section evaluates our online profiling and analysis framework and investigates the performance impact of dynamic prefetching.

4.1 Experimental Methodology

The programs used in this study include several of the memory-performance-limited SPECint2000 benchmarks, and *boxsim*, a graphics application that simulates spheres bouncing in a box. We applied our dynamic prefetching framework to these benchmarks and used the *prefetcht0* instruction supplied on the Pentium III to prefetch data into both levels of the cache hierarchy. The following framework settings were used for all experiments, unless mentioned otherwise. The bursty tracing sampling rate was set at 0.5% during the active profiling period, with profiling bursts extending through 60 dynamic checks (i.e., $nCheck_0$=11,940 and $nInstr_0$= 60). The online optimization controls were set to actively profile and analyze 1 second of every 50 seconds of program execution, where active periods are 50 burst periods long (i.e., $nAwake_0$ = 50, $nHibernate_0$ = 2,450). The hot data stream analysis detected streams that contain more than 10 references, and account for at least 1% of the collected trace. These settings are not the result of careful tuning; rather our experience indicates that a fairly broad range of reasonable settings performs equivalently. Measurements were performed on a uniprocessor 550 Mhz Pentium III PC with 512 MB of memory, 256 KB, 8-way L2, and 16KB, 4-way L1 data cache, both with 32 byte cache blocks, running Windows 2000 Server. The SPEC benchmarks were run with their largest input data set (ref). *boxsim* was used to simulate 1000 bouncing spheres. All measurements report the average of five runs.

4.2 Evaluating the Online Profiling and Analysis Framework

Figure 11 reports the overhead of our online profiling and analysis infrastructure. The *Basic* bar indicates the overhead of just the dynamic checks without (virtually) any data reference profiling. This is measured by setting $nCheck_0$ to an extremely large value and $nInstr_0$ to 1. We applied the techniques described in [15] to reduce this dynamic check overhead. It is important that this overhead be small since any dynamic optimization must overcome this to produce performance improvements. In addition, unlike other sampling-related overhead, this cannot be reduced by changing the framework's counter settings. As Figure 11 shows, this overhead is reasonably low, ranging from around 2.5% for boxsim to 6% for parser. The *Prof* bar indicates the overhead of collecting the temporal data reference trace at the counter settings discussed in Section 4.1. Data reference profiling at this sampling rate adds very little additional overhead, which ranges from almost nothing for mcf to 1.6% overhead for vortex. Thus, we can collect sampled temporal data reference profiles for all our benchmarks with a maximal overhead of only 6.5%, in the case of twolf and parser. Finally, the *Hds* bar indicates the overhead of collecting the temporal data reference profiles and analyzing them to detect hot data streams according to the parameters in Section 4.1. Again, this adds very little overhead; vortex at 1.4% incurs the largest additional overhead. Considering all three contributors to overhead, we see that at the current sampling rate most of the overhead arises from the dynamic checks. The overall overhead of our online profiling and analysis is reasonably low, and ranges from around 3% for mcf to 7% for parser and vortex. Any dynamic optimization based on hot data streams, that operates in our framework must produce greater improvements than this to positively impact overall program performance.

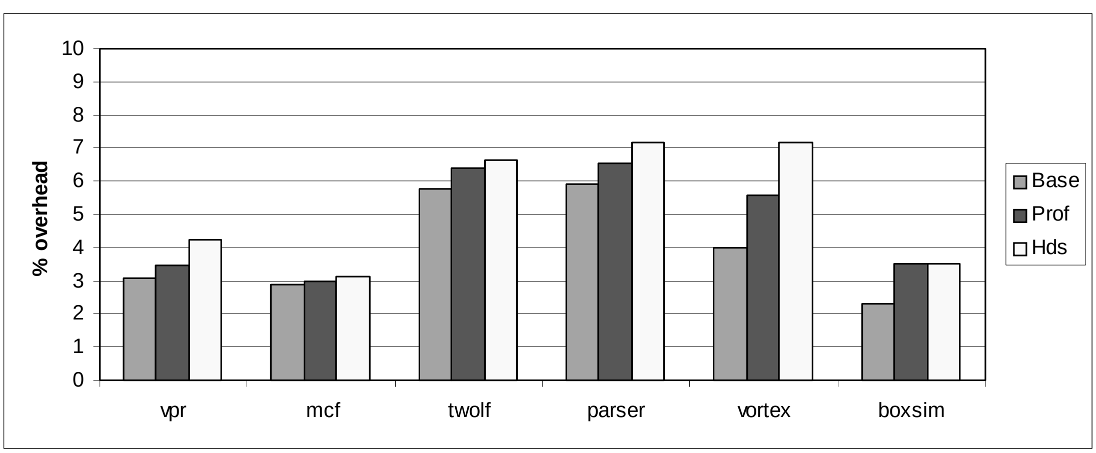

Figure 11. Overhead of online profiling and analysis.

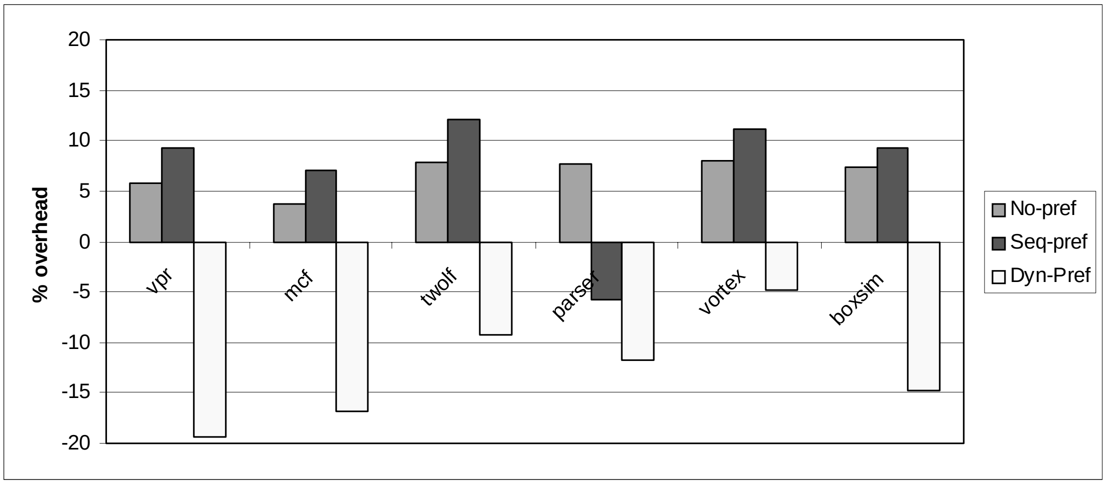

Figure 12. Performance impact of dynamic prefetching.

4.3 Dynamic Prefetching Evaluation

Figure 12 shows the overall impact of our dynamic prefetching scheme on program performance, normalized to the execution time of the original unoptimized program. The Y axis measures percentage overhead; positive values indicate performance degradation, and negative values indicate speedups. The *No-pref* bars report the cost of performing all the profiling, analysis and hot data stream prefix matching, yet not inserting prefetches. This measures the overhead of our dynamic prefetching analysis, which must be overcome by effective prefetching to yield net performance gains. The prefix-match checks add an additional 0.5% (mcf, parser) to 4% (boxsim) overhead compared with the hot data stream analysis (compare *No-pref* with *Hds* bar in Figure 11), for a configuration that matched the first two references of a hot data stream prior to initiating prefetching. Changing this to match a single data stream element before initiating prefetching lowered this overhead, but at the cost of less effective prefetching, yielding a net performance loss. Matching the first three data stream elements before initiating prefetching increased this overhead without providing any corresponding benefit in prefetching accuracy, resulting in a net performance loss as well.. In addition, our current implementation makes no attempt to schedule prefetches (they are triggered as soon as the prefix matches). More intelligent prefetch scheduling could produce larger benefits.

The *Seq-pref* bars measures the benefit of a prefetching scheme that uses the hot data stream analysis to insert dynamic prefetches at appropriate program points, but ignores the data stream addresses. Instead, it prefetches cache blocks that sequentially follow the last prefix-matched hot data stream reference (i.e., the stream reference, which when matched, causes the prefetch sequence to be initiated). This scheme is equivalent to our dynamic prefetching scheme if hot data streams are sequentially allocated. The data indicates that with the sole exception of parser, which has several sequentially allocated hot data streams and runs around 5% faster overall, none of the benchmarks benefit from this approach. The other benchmarks suffer performance degradations that range from 7% (mcf) to 12% (twolf), which indicates that these prefetches pollute the cache.

Finally, the *Dyn-pref* bars reports the performance of our dynamic prefetching implementation (achieved by setting the hot data stream prefix matching length to 2). Prefetching produces a net performance improvement of 5% (vortex) to 19% (vpr). This is despite the 4–8% overhead that the prefetching has to overcome to show net performance improvements. Comparing these results to the *Seq-pref* numbers highlights the importance of using the hot data streams addresses as prefetch targets. In addition, manual examination of the hot data addresses indicates that many will not be successfully prefetched using a simple stride-based prefetching scheme. However, a stride-based prefetcher could complement our

Table 2: Detailed dynamic prefetching characterization

Benchmark	# of opt. cycles	# of traced refs (per cycle avg.)	# of hds (per cycle avg.)	# of DFSM states, transitions (per cycle avg.)	# of procs. modified (per cycle avg.)
vpr	17	83,231	41	<79 states, 68 checks>	7
mcf	36	72,537	37	<75 states, 74checks>	6
twolf	55	87,981	25	<42 states, 41checks>	11
parser	4	73,244	21	<43 states, 42 checks>	9
vortex	3	67,852	14	<29 states, 28 checks>	12
boxsim	19	87,818	23	<40 states, 36 checks>	7

scheme by prefetching data address sequences that do not qualify as hot data streams.

Table 2 provides a more detailed characterization of our dynamic prefetching implementation. The second column indicates the number of prefetch optimization cycles performed during program execution. Longer running programs produce a greater number of these optimization cycles. The next three columns show the number of traced references, hot data streams detected, and the size of the DFSMs used for prefix matching, all averaged on a per optimization cycle basis. The last column contain the number of procedures modified to insert prefix-match checks or prefetches, again averaged on a per cycle basis. The results indicate that the prefetching benefits arise from targeting a small set of program hot data streams.

5. RELATED WORK

This section discusses related work on prefetching and software dynamic optimization.

5.1 Prefetching

Prefetching is a well known optimization that attempts to hide latency resulting from poor reference locality. We are concerned with data prefetching (as opposed to instruction prefetching) into the processor cache. Prefetching mechanisms can be classified as software prefetching (using non-blocking load instructions provided by most modern processors) and hardware prefetching (extending the memory management subsystem architecture). Prefetching mechanisms can also be characterized by the kind of regularity they require of the target program and by their degree of automation. We review only the most closely related techniques here; a survey of prefetching techniques is [35].

Early prefetching techniques mainly focused on improving the performance of scientific codes with nested loops that access dense arrays. Both software and hardware techniques exist for such regular codes. The software techniques use program analysis to determine the data addresses needed by future loop iterations, and employ program transformations, such as loop unrolling and software pipelining to exploit that information [20, 24]. Hardware prefetching techniques include stride prefetchers and stream buffers. Stride prefetchers learn if load address sequences are related by a fixed delta and then exploit this information to predict and prefetch future load addresses [7]. Stream buffers can fetch linear sequences of data and avoid polluting the processor cache by buffering the data [17]. These techniques are mostly limited to programs that make heavy use of loops and arrays, producing regular access patterns.

Jump pointers are a software technique for prefetching linked data structures, overcoming the array-and-loop limitation. Artificial jump pointers are extra pointers stored into an object that point to an object some distance ahead in the traversal order. On future traversals of the data structure, the targets of these extra pointers are prefetched. Natural jump pointers are existing pointers in the data structure used for prefetching. For example, greedy prefetching makes the assumption that when a program uses an object o, it will use the objects that o points to, in the near future, and hence prefetches the targets of all pointer fields. These techniques were introduced by Luk and Mowry in [22] and refined in [5, 18]. Stoutchinin et al. describe a profitability analysis for prefetching with natural jump pointers [33]. A limitation of these techniques is that their static analyses are restricted to regular linked data structures accessed by local regular control structures.

Various hardware techniques, related to greedy prefetching, have been proposed for prefetching linked data structures. In dependence-based prefetching, producer-consumer pairs of loads are identified, and a prefetch engine speculatively traverses and prefetches them [26]. Dependence-based prefetching has also been combined with artificial jump-pointer prefetching in software or hardware [27]. In dependence-graph precomputation, a backward slice of instructions in the instruction fetch queue is used to chose a few instructions to execute speculatively to compute a prefetch address [1]. And in content-aware prefetching, data that is brought in to satisfy a cache miss is scanned for values that may resemble addresses, and those addresses are used for prefetching [12].

The hardware technique that best corresponds to history-pointers is correlation-based prefetching. As originally proposed, it learns digrams of a key and prefetch addresses: when the key is observed, the prefetch is issued [6]. Joseph and Grunwald generalized this technique by using a Markov predictor [16]. Nodes in the Markov-model are addresses, and the transition probabilities are derived from observed digram frequencies. Upon a data cache miss to an address that has a node in the Markov model, prefetches for a fixed number of transitions from that address are issued, prioritized by their probabilities.

Our techniques differs from prior software prefetching techniques in at least three ways. First, it is profile-based and does not rely on static analysis. Second, being profile-based it works for arbitrary data structure traversals. Finally, it is a dynamic technique that is capable of adaptation as the program executes. Our dynamic prefetching is most similar to correlation-based hardware prefetching in that it observes past data accesses to predict future accesses. Unlike the correlation-based prefetchers mentioned above, it is a software technique that can be easily configured and tuned for a particular program, performs more global access pattern analysis, and is capable of using more context for its predictions than digrams of data accesses.

5.2 Software Dynamic Optimization

Common examples of software dynamic optimizers are some of the more sophisticated Java virtual machines such as Intel's Microprocessor Research Lab VM [11], Sun's HotSpot VM [25], and IBM's Jikes RVM [2]. All of these contain just-in-time compilers and use runtime information to concentrate optimization efforts on frequently executing methods. Unlike our system, they do not focus on memory hierarchy optimizations, and possess only limited cross-procedure optimization capabilities.

Recently, some dynamic optimizers that operate on compiled object code have been proposed. The Wiggins/Redstone system uses hardware performance counters to profile a program executing on the Alpha processor, and optimizes single-entry multiple-exit regions of hot basic blocks [13]. The University of Queensland Dynamic Binary Translator translates an program that is compiled for one architecture just in time for execution on another architecture, and collects a full edge-weight profile to identify groups of connected hot blocks for optimization [34]. The Dynamo system interprets a program to collect a basic block profile. Once a basic block reaches a heat threshold, Dynamo considers the linear sequence of blocks executed directly afterwards as a hot path, which it then optimizes [4]. All of these systems optimize code in hot control paths that may cross procedure boundaries. Unlike our system, they do not focus on memory hierarchy optimizations.

A few dynamic memory hierarchy optimizers implemented in software do exist. Saavedra and Park dynamically adapt the prefetch distance of array-and-loop software prefetching to the changing latencies of a NUMA architecture [29]. They also discuss adaptive profiling: when profiling information changes, the profiler starts polling more frequently. This idea may be a useful extension to our simpler hibernation approach. Chilimbi and Larus use a copying generational garbage collector to improve reference locality by clustering heap objects according to their observed data access patterns [9]. Harris performs dynamic adaptive pretenuring for Java programs by identifying allocation sites that often allocate long-lived objects [14]. His system modifies these allocations to directly place objects into the old generation of a generational garbage collector, saving the work of repeatedly scanning them in the young generation. Kistler and Franz reorder fields in objects so fields accessed together reside in the same cache block, and discuss how this can be done during copying garbage collection [19].

5.3 State Machine Predictor Generation

Sherwood and Calder propose an algorithm that generates FSM predictors from temporal profiling data [30]. In their case study, the profile is a trace of branch executions. Each FSM is driven by the global branch direction bitstring, and predicts whether a particular branch is taken or not taken. While we also generate an FSM predictor from temporal profiling data, there are some fundamental differences to the Sherwood-Calder approach. First of all, Sherwood and Calder generate FSM predictors in hardware for special-purpose processors, while we use a dynamic software approach. They restrict FSMs to be driven by bitstrings and predict a single bit (one step of their FSM generation algorithm represents the predictor by a boolean formula), while we predict sets of prefetch addresses. They use fixed-sized histories, while our hot data streams are variable-length. They drive several FSMs in parallel, while we combine all FSMs into one.

6. CONCLUSIONS

This paper describes a dynamic software prefetching framework for general-purpose programs. The prefetching scheme runs on stock hardware, is completely automatic, and can handle codes that traverse pointer-based data structures. It targets a program's hot data streams, which are consecutive data reference sequences that frequently repeat in the same order. We show how to detect hot data streams online with low-overhead, using a combination of bursty tracing and a fast hot data stream analysis algorithm. Our experimental results demonstrate that our prefetching technique is effective, providing overall execution time improvements of 5–19% for several memory-performance-limited SPECint2000 benchmarks running their largest (ref) inputs.

7. REFERENCES

[1] M. Annavaram, J. Patel, and E. Davidson. “Data prefetching by dependence graph precomputation.”In *International Symposium on Computer Architecture (ISCA)*, 2001.

[2] M. Arnold et al. “Adaptive optimization in the Jalapeno JVM”, In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2000.

[3] M. Arnold, and B. Ryder. “A Framework for Reducing the Cost of Instrumented Code.” In *ACM SIGPLAN’01 Conference on Programming Languages Design and Implementation (PLDI)*, 2001.

[4] V. Bala, E. Duesterwald, and S. Banerjia. “Dynamo: A transparent dynamic optimization system.” In *ACM SIGPLAN’00 Conference on Programming Languages Design and Implementation (PLDI)*, 2000.

[5] B. Cahoon, and K. McKinley. “Data flow analysis for software prefetching linked data structures in Java.” In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[6] M. Charney, and A. Reeves. “Generalized correlation based hardware prefetching.” *Tech report EE-CEG-95-1*, Cornell University, 1995.

[7] T. Chen, and J. Baer.” Reducing memory latency via non-blocking and prefetching caches.”In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*,1992.

[8] T.M. Chilimbi. “Efficient Representations and Abstractions for Quantifying and Exploiting Data Reference Locality.” In *Proceedings of the ACM SIGPLAN’01 Conference on Programming Language Design and Implementation*, June 2001

[9] T. M. Chilimbi, and J. R. Larus. “Using generational garbage collection to implement cache-conscious data placement.” In *Proceedings of the 1998 International Symposium on Memory Management*, Oct. 1998.

[10] T. M. Chilimbi. “On the stability of temporal data reference profiles.” In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2001.

[11] M. Cierniak, G. Lueh, and J. Stichnoth. “Practicing JUDO: Java under dynamic optimizations.” In *ACM SIGPLAN’00 Conference on Programming Languages Design and Implementation (PLDI)*, 2000.

[12] R. Cooksey, D. Colarelli, and D. Grunwald, “Content-based prefetching: Initial results”, In *Workshop on Intelligent Memory Systems*, 2000.

[13] D. Deaver, R. Gorton, and N. Rubin, ”Wiggins/Redstone: An online program specializer.”, In *Hot Chips*, 1999.

[14] T. Harris. “Dynamic adaptive pre-tenuring.” In *International Symposium on Memory Management (ISMM)*, 2000.

[15] M. Hirzel and T. Chilimbi. “ Bursty Tracing: A Framework for Low-Overhead Temporal Profiling”, In *Workshop on Feedback-Directed and Dynamic Optimizations (FDDO)*, 2001.

[16] D. Joseph and D. Grunwald. “ Prefetching using Markov predictors”, In *International Symposium on Computer Architecture (ISCA)*, 1997.

[17] N. Jouppi. “Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers”, In *International Symposium on Computer Architecture (ISCA)*, 1990.

[18] M. Karlsson, F. Dahlgren, and P. Stenstrom. “A Prefetching Technique for Irregular Accesses to Linked Data Structures, In *High Performance Computer Architectures (HPCA)*, 1999.

[19] T. Kistler and M. Franz. “Automated data-member layout of heap objects to improve memory-hierarchy performance.” In *Transactions on Programming Languages and Systems (TOPLAS)*, 2000.

[20] A. Klaiber and H. Levy. “An architecture for software-controlled data prefetching.” In *International Symposium on Computer Architecture (ISCA)*, 1991.

[21] J. R. Larus. “Whole program paths.” In *Proceedings of the ACM SIGPLAN’99 Conference on Programming Language Design and Implementation,* pages 259-269, May 1999.

[22] C. K. Luk, and T. Mowry. “Compiler-based prefetching for recursive data structures.” In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996

[23] C. G. Nevill-Manning and I. H. Witten. “Linear-time, incremental hierarchy inference for compression.” In *Proceedings of the Data Compression Conference (DCC’97)*, 1997.

[24] T. Mowry, M. Lam, and A. Gupta. “Design and Analysis of a Compiler Algorithm for Prefetching.”, In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1992.

[25] M. Paleczny, C. Vick, and C. Click. “The Java HotSpot server compiler.”, In *USENIX Java Virtual Machine Research and Technology Symposium (JVM)*, 2001.

[26] A. Roth, A. Moshovos, and G. Sohi. “Dependence based prefetching for linked data structures.” In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1998.

[27] A. Roth and G. Sohi. “Effective jump pointer prefetching for linked data structures.” In *International Symposium on Computer Architecture (ISCA)*, 1999.

[28]S. Rubin, R. Bodik, and T. Chilimbi. “An Efficient Profile-Analysis Framework for Data-Layout Optimizations.” In *Principles of Programming Languages, POPL’02*, Jan 2002.

[29] R. Saavedra and D. Park. “Improving the effectiveness of software prefetching with adaptive execution.” In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 1996.

[30] T. Sherwood and B. Calder. “Automated design of finite state machine predictors for customized processors.” In *International Symposium on Computer Architecture (ISCA)*, 2001.

[31] A. Srivastava and A. Eustace. “ATOM: A system for building customized program analysis tools.” In *Proceedings of the ACM SIGPLAN’94 Conference on Programming Language Design and Implementation*, pages 196-205, May 1994.

[32] A. Srivastava, A. Edwards, and H. Vo. “Vulcan: Binary transformation in a distributed environment.”, In *Microsoft Research Tech Report, MSR-TR-2001-50*, 2001.

[33] A. Stoutchinin et al. “Speculative prefetching of induction pointers.” In *International Conference on Compiler Construction (CC)*, 2001.

[34] D. Ung, and C. Cifuentes.”Opimising hot paths in a dynamic binary translator.”In *Workshop on Binary Translation*, 2000.

[35] S. VanderWiel, and D. Lilja. “Data prefetch mechanisms”, In-*ACM Computing Surveys*, 2000.