The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System *

Jiwei Lu, Howard Chen, Rao Fu, Wei-Chung Hsu, Bobbie Othmer, Pen-Chung Yew
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
{jiwei, chenh, rfu, hsu, bothmer, yew}@cs.umn.edu

Dong-Yuan Chen
Microprocessor Research Lab
Intel Corporation
dychen@intel.com

Abstract

Traditional software controlled data cache prefetching is often ineffective due to the lack of runtime cache miss and miss address information. To overcome this limitation, we implement runtime data cache prefetching in the dynamic optimization system ADORE (ADaptive Object code REoptimization). Its performance has been compared with static software prefetching on the SPEC2000 benchmark suite. Runtime cache prefetching shows better performance. On an Itanium 2 based Linux workstation, it can increase performance by more than 20% over static prefetching on some benchmarks. For benchmarks that do not benefit from prefetching, the runtime optimization system adds only 1%-2% overhead. We have also collected cache miss profiles to guide static data cache prefetching in the ORC®compiler. With that information the compiler can effectively avoid generating prefetches for loops that hit well in the data cache.

1. Introduction

Software controlled data cache prefetching is an efficient way to hide cache miss latency. It has been very successful for dense matrix oriented numerical applications. However, for other applications that include indirect memory references, complicated control structures and recursive data structures, the performance of software cache prefetching is often limited due to the lack of cache miss and miss address information. In this paper, we try to keep the applicability of software data prefetching by using cache miss profiles and a runtime optimization system.

*This work is supported in part by the U.S. National Science Foundation under grants CCR-0105574 and EIA-0220021, and grants from Intel, Hewlett Packard and Unisys.

1.1. Impact of Program Structures

We first give a simple example to illustrate several issues involved in software controlled data cache prefetching. Fig. 1 shows a typical loop nest used to perform a matrix multiplication. Experienced programmers know how

```
void matrix_multiply(long A[N][N], long B[N][N], long C[N][N])
{       int  i, j, k;
        for ( i = 0; i < N; ++i )
                for (j = 0; j < N; ++j )
                        for (k = 0; k < N; ++k )
                                A[i][j] += B[i][k] * C[k][j];
}
```

Figure 1. Matrix Multiplication

to use cache blocking, loop interchange, unroll and jam, or library routines to increase performance. However, typical programmers rely on the compiler to conduct optimizations. In this simple example, the innermost loop is a candidate for static cache prefetching. Note that the three arrays in the example are passed as parameters to the function. This introduces the possibility of aliasing, and results in less precise data dependency analysis. We used two compilers for Itanium system in this study: the Intel®C/C++ Itanium®Compiler (i.e. ECC V7.0) [15] and the ORC open research compiler 2.0 [26]. Both compilers have implemented static cache prefetch optimizations that are turned on at O3. For the above example function, the ECC compiler generates cache prefetches for the innermost loop, but the ORC compiler does not. Moreover, if the above example is re-coded to treat the three arrays as global variables, the ECC compiler generates much more efficient code (over 5x faster on an Itanium 2 machine) by using loop unrolling. This simple example shows that program structure has a significant impact on the performance of static cache prefetching. Some program structures require more complicated

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)
0-7695-2043-X/03 $ 17.00 © 2003 IEEE

analysis (such as interprocedural analysis) to conduct efficient and effective cache prefetching.

1.2. Impact of Runtime Memory Behavior

It is in general difficult to predict memory working set size and reference behaviors at compile time. Consider Gaussian Elimination, for example. The execution of the loop nest usually generates frequent cache misses at the beginning of the execution and few cache misses near the end of the loop nest. Initially, the sub-matrix to be processed is usually too large to fit in the data caches; hence frequent cache misses will be generated. As the sub-matrices to be processed get smaller, they may fit in the cache and produce fewer cache misses. It is hard for the compiler to generate one binary that meets the cache prefetch requirements for both ends.

The *memcpy* library routine is another example. In some applications, the call to *memcpy* may involve a large amount of data movement and intensive cache misses. In some other applications, the calls to the *memcpy* routine have few cache misses. Once again, it is not easy [1] to provide one *memcpy* routine that meets all the requirements.

```
void daxpy( double *x, double *y, double a, int n )
{           int i ;
            for ( i = 0; i < n; ++i  )
                        y[i] += a * x[i];
}
```

Figure 2. DAXPY

1.3. Impact of Micro-architectural Constraints

Microarchitecture can also limit the effectiveness of software controlled cache prefetching. For example, the issue bandwidth of memory operations, the memory bus and bank bandwidth [14][34], the miss latency, the non-blocking degree of caches, and memory request buffer size will affect the effectiveness of software cache prefetching. Consider the DAXPY loop in Fig. 2, for example. On the latest Itanium 2 processor, two iterations of this loop can be computed in one cycle (2 *ldfpd*s, 2 *stfd*s, 2 *fma*s, which can fit in two MMF bundles). If prefetches must be generated for both x and y arrays, the requirement of two extra memory operations per iteration would exceed the "two bundles per cycle" constraint. Since the array references in this example exhibit unit stride, the compiler could unroll the loop to reduce the number of prefetch instructions. For non-unit stride loops, prefetch instructions are more difficult to reduce.

Stride-based prefetching is easy to perform efficiently. Prefetching for pointer chasing references and indirect memory references [23][25][29][35][22] are relatively challenging since they incur a higher overhead, and must be used more carefully. A typical compiler would not attempt high overhead prefetching unless there is sufficient evidence that a code region has frequent data cache misses.

Due to the lack of cache miss information, static cache prefetches usually are less aggressive in order to avoid undesirable runtime overhead. Therefore we attempt to conduct data cache prefetching at runtime through a dynamic optimization system. This dynamic optimization system automatically monitors the performance of the execution of the binary, identifies the performance critical loops/traces with frequent data cache misses, re-optimizes the selected traces by inserting cache prefetch instructions, and patches the binary to redirect the subsequent execution to the optimized traces. Using cache miss and cache miss address information collected from the hardware performance monitors, our runtime system conducts more efficient and effective software cache prefetching, and significantly increases the performance of several benchmark programs over static cache prefetching. Also in this paper, we will establish a secondary process for comparison to feedback cache miss profile to the ORC compiler so that the compiler could perform cache prefetch optimizations only for the code region/loop that has frequent cache misses.

The remainder of the paper is organized as follows. Section 2 introduces the performance monitoring features on Itanium processors and the framework of our runtime optimization system. Section 3 discusses the implemention of the runtime data cache prefetching. In Section 4, we present the performance evaluation of runtime prefetching and a profile-guided static prefetching approach. Section 5 highlights the related works and Section 6 offers the conclusion and future work.

2. Runtime Optimization and ADORE

The ADORE (ADaptive Object code REoptimization) system is a trace based user-mode dynamic optimization system. Unlike other dynamic optimization systems [10][3][7], its profiling and trace selection are based on sampling of hardware performance monitors. This section explains performance monitoring, trace selection and optimization mechanisms in ADORE. Runtime prefetching is part of the ADORE system and will be discussed in detail in Section 3.

[1] A compiler may generate multiple versions of *memcpy* to handle different cases.

2.1. Performance Monitoring and Runtime Sampling on Itanium processor

Intel's Itanium architecture offers extensive hardware support for performance monitoring [19]. The Itanium-2 processor provides more than one hundred counters to measure the performance events such as memory latency, branch prediction rate, CPU cycles, retired-instruction counts, and pipeline stalls. Two usage models are supported using these performance counters: *workload charaterization*, which gives the overall runtime cycle breakdown, and *profiling*, which provides information for identifying program bottlenecks. Both models are programmable and fully supported in the latest IA64 Linux kernel. Based on these two models, Stéphane Eranian developed a generic kernel interface called *perfmon* [28] to help programmers access the IA64 PMU (Performance Mornitoring Unit) and collect performance profiles on Itanium processors. The sampling of ADORE is built on top of this kernel interface.

In ADORE, sample collection is achieved through a signal handler communicating with the *perfmon* interface. *Perfmon* samples the IA64 PMU every N CPU cycles (e.g. $N = 300,000$ cycles). Once the kernel sample buffer fills up, it throws a buffer-overflow signal to invoke the signal handler routine to move the samples out to a user buffer for trace selection/optimization.

Each PMU sample consists of three accumulative counter values required by ADORE: *CPU cycles*, *Retired Instruction Count*, and *Data Cache Load Miss Count*. In addition, ADORE needs the samples of the *Branch Trace Buffer* (BTB) registers and the *Data Event Address Registers* (DEAR). BTB is a circular register file recording the most recent 4 branch outcomes and the source/target address information. DEAR holds the most recent address/latency information related to a data cache load miss, a TLB miss or an ALAT miss. For example, ADORE will sample DEAR for the latest data cache miss events with load latency $\geq$ 8 cycles. Finally, each sample is in the form of an n-tuple: <sample index, *pc* address of sample, CPU_Cycle, D-Cache Miss Count, Retired-Instruction Count, BTB values, DEAR values>.

2.2. Dynamic Optimization System

ADORE is implemented as a shared library on Linux for IA64 that can be automatically linked to the application at startup. It is also a runtime trace optimizer like Dynamo [3], but ADORE relies on the Itanium hardware performance monitor (HPM) to identify hotspots instead of using interpretation. A *trace* is a single entry, multi-exit code sequence. In Dynamo, a trace is selected once the reference count of the target block of a backwards taken branch exceeds a threshold. In ADORE, trace selection is based on

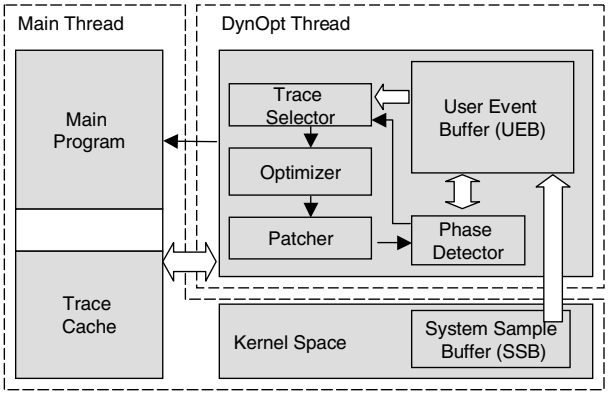


Figure 3. ADORE Framework

```
int __libc_start_main (…) {
        …
        dyn_open( );
        on_exit (dyn_close);
        exit ((*main)(argc, argv, __environ));
}
```

Figure 4. Startup code

the branch trace samples collected by HPM. Fig. 3 illustrates the framework of ADORE. In ADORE, there are two threads existing at runtime, one is the original main thread running the unmodified executable; the other is a dynamic optimization thread in charge of phase detection, trace selection and runtime prefetching. When the Linux system starts a program, it invokes a *libc* entry-point routine named *__libc_start_main*, within which the *main* function is called. We modified this routine by including our startup codes as shown in Fig. 4.

Function *dyn_open* and *dyn_close* are used to open/close the dynamic optimizer. *dyn_open* carries out four tasks. First, it creates a large shared-memory block for the original process. This is the *trace pool*, which stores the optimized traces. Second, it initiates the *perfmon* kernel interface. *Perfmon* resets the PMU (Performance Monitoring Unit) in the Itanium processor, determines the sampling rate and creates a *kernel sampling buffer*, which is called *System Sampling Buffer*(SSB) in this paper. Third, *dyn_open* installs the signal handler to copy all sample events from the SSB to a larger circular *User Event Buffer* (UEB) every time the SSB overflows. Finally, it creates a new thread for dynamic optimization that has the same lifetime as the main thread. Next, *dyn_close* is registered by the library function *on_exit* so that it can be invoked at the end of main program execution. *dyn_close* frees the memory and notifies the optimizer thread that the main program is complete.

2.3. Coarse-Grain Phase Detection

Recent research [31][13][6] shows that a running program often exhibits multiple phases during execution. In order to detect execution phases, ADORE implements a coarse-grain phase detector. This phase detector is lightweight, quick to respond, and reasonably accurate in detecting stable phases and catching phase changes. Since a stable phase may have different meanings in different contexts, we define a *stable phase* for the purpose of prefetching as a stage in which the program is repeatedly executing the same set of code with a relatively stable CPI and cache miss rate. The phase detection algorithm used in ADORE is as follows:

We define a *profile window* as the period of time for the SSB to fill up. Let $SIZE_{UEB} = SIZE_{SSB} * W$, where W is a positive number ($W = 16$ in ADORE). Hence the UEB can consist of up to W profile windows. By setting the sampling interval to R cycles/sample and $SIZE_{SSB}$ to N samples, the SSB will overflow every $R \times N$ cycles. The UEB will then contain the latest performance profile of $W \times R \times N$ cycles. For instance, if $W = 8$, $R = 250,000$ and $N = 4,000$, the UEB can contain the most recent 8 seconds of performance history on a 1GHz machine.

To decide whether the main program incurs a phase change or starts a stable phase, the phase detector is invoked every 100 milliseconds to check whether a new profile window has been added to the UEB. If so, it computes the CPI (Cycles Per Instruction), DPI (D-cache Load Miss Per Instruction) and PC_{center} for this profile window. The PC_{center} is computed as the arithmetic mean of all of the *pc* addresses from the samples in that window. Thus, each profile window in UEB has three values: CPI, DPI, PC_{center}. If the phase detector detects there are consecutive profile windows having low standard deviations of the above three factors, it signals that a stable phase has occurred. Likewise, high deviations suggest a change of the current stable phase. We compute the PC_{center} to estimate the center of the code area of each profile window and compute the standard deviations to determine the fluctuation of code area centers of consecutive profile windows. To improve accuracy, the algorithm removes noise in the above computations. For runtime cache prefetching, we ignore phases that do not have high cache miss rate. Moreover, if a phase turns out to be from the trace pool, this phase will be skipped to avoid re-optimization (But we may continue to monitor the execution of the optimized trace to detect and fix nonprofitable ones).

Our study shows that occasionally a stable phase cannot be detected for a long time, i.e., the deviations are always greater than the thresholds. In such cases, the phase detector doubles the size of the profile window in case the window is too small to accommodate a large phase.

2.4. Trace Selection

The trace selector starts to build traces after a stable phase has been detected. We will only briefly discuss trace selection here and trace patching in Section 2.5 since the focus of this paper is runtime cache prefetching.

Trace selection starts by reading all samples in the UEB (User Event Buffer). Remember that the Branch Trace Buffer in performance monitoring allows for 4 consecutive branch outcomes to be recorded in each sample. These branch outcomes form a fraction of path profile [4]. The trace selector uses one hash table to save the path profiles and a secondary hash table to save all the branch targets. When selecting a trace, the trace selector starts from the target address having the largest reference count and builds the traces based on the path profiles. This work is not too hard, but there are issues unique to the Itanium processor family. For instance, IA64 instructions are encoded in bundles of 2-3 instructions with different template types. It is common that the second slot in a bundle is a taken branch. Thus we have to break the current bundle and connect the prior instruction stream with the instructions starting from the taken branch's target address, discarding the remaining instruction in the fall-through path. Furthermore, the IA64 ISA [18] provides predication, so a single basic block may have disjoint execution paths that complicates data analysis. Nested predicates also makes branch conversion (flip a taken branch into fall-through) difficult.

Instructions along the hottest path are added to the current trace until a trace stop-point is reached. This stop-point can be a function-return, a back-edge branch that makes the trace into a loop, or a conditional branch whose taken/fall-through bias is balanced. Upon this point, the trace selector adds the current trace into a trace queue and continues to select the next trace. After trace selection is completed, control will be transferred to the dynamic optimizer.

2.5. Trace Patching

Trace patching involves writing optimized traces into the trace pool. At this stage, the trace patcher prepares an unused memory area in the trace pool for each trace. Labels and branch targets must then be mapped into the trace pool. During patching, branch instructions that jump to the original code of other traces will be modified to branch to the new traces. Furthermore, the first instruction bundle of each trace in the original code is replaced by a new bundle that has only a branch instruction jumping to the optimized trace in the trace pool. The replaced bundle is not simply overwritten; it is saved so that if the dynamic optimizer wants to unpatch the trace later on, it only needs to write this bundle back.

3. Runtime Prefetching

The purpose of runtime software prefetching is to insert prefetch instructions into the binary code to hide memory latency. In the current version of ADORE, data cache prefetching is the major optimization implemented. Our experiments show that by inserting prefetch instructions at runtime, half of the SPEC2000 benchmark programs compiled with O2 gain performance.

Just like the traditional software prefetching, our runtime optimizer merges the prefetching code directly into the traces to hide large cache miss latency after identifying the delinquent loads in hot loops. The approach of runtime prefetching in ADORE is as follows: (a) Use performance samples to locate the most recent delinquent loads. (b) If the load instruction is in a loop-type trace, extract its dependent instructions for address calculation. (c) Determine its data reference pattern. (d) Calculate the stride if it has spatial or structural locality. Otherwise, insert special codes to predict strides for pointer-chasing references. (e) Schedule the prefetches.

3.1. Tracking Delinquent Loads

Each sample contains the latest data cache miss event with latency no less than 8 cycles. On the Itanium based systems, this much latency implies L2 or L3 cache misses. In general, there are more L1 cache misses. However, the performance loss due to L2/L3 cache misses is usually higher because of the greater miss latency. Therefore prefetching for L2 and L3 cache misses can be more cost effective.

To track a delinquent load, the source address, the latency and the miss address of each cache miss event are mapped to the corresponding load instruction in a selected trace (if any). With path profile based trace selection, it is possible that one delinquent load appears in two or more selected traces. But in most of the cases, there is only one loop trace that contains the load instruction, and the current implementation runtime prefetching is targeted for loop traces only. Finally, prefetching in ADORE is applied to at most the top three miss instructions in each loop-type trace, i.e., the load instructions with the greatest percentage of overall latency.

3.2. Data Reference Pattern Detection

For software prefetching, there are three important data reference patterns in loops: direct array reference, indirect array reference and pointer-based reference. It is not obvious what pattern a delinquent load belongs to at the binary-level. For pointer-chasing prefetching, some papers [11] [34] propose checking the load address to see if it is referring the memory heap. This is impractical for software based prefetching. To recognize the above three reference patterns, the runtime prefetcher in ADORE analyzes the dependent instructions for the address calculation of each delinquent load.

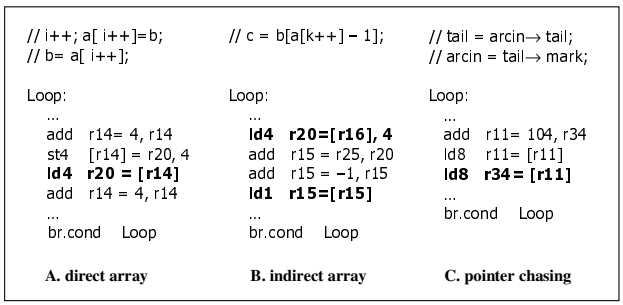


Figure 5. Data Reference Patterns and Dependence Code Slices

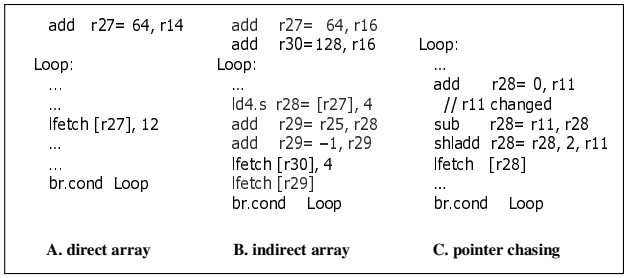


Figure 6. Prefetching for Different Reference Patterns

3.2.1. Direct/Indirect Array References

Fig. 5 illustrates the three data reference patterns recognized by ADORE's dynamic optimizer. Cases A and B in Fig. 5 show that direct array reference (single-level memory access) and indirect array reference (multi-level memory access) patterns exhibiting spatial locality. Load instructions in bold fonts are delinquent loads. Other instructions are related to address computation taken from the selected loop-type traces. For example, the stride of case A in Fig. 5 is calculated by incrementing $r14$ by 4 three times. So the stride is $4 + 4 + 4 = 12$. Case B in Fig. 5 shows a 2-level memory reference, in which both level of references have significant miss penalties. For such cases, two different code slices should be inserted to prefetch for both level of memory access, and the prefetch for the first level reference must be in several iterations ahead of that for the second level reference (See Fig. 6).

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)
0-7695-2043-X/03 $ 17.00 © 2003 IEEE

3.2.2. Pointer Chasing Reference

Pointer-based references are difficult for software prefetching. Wu [35] proposes a profile-guided prefetching technique to find the regular strides in irregular programs. His method requires profiles of address data of consecutive iterations. In the current runtime prefetching in ADORE, an approach similar to *induction pointer* [33] is used to help approximate the data traversal of pointer-chasing references. However, to insert correct prefetches, the dynamic optimizer must find the recurrent pointer from dependance analysis. Case C in Figure 5 gives a typical example in *181.mcf*. In this example, $r11$ is the pointer critical to the data traversal because $r11$ is used to compute the miss address and "*ld8 r34=[r11]*" is the delinquent load. Therefore $r11$ is chosen to apply the induction pointer based prefetching in this case.

The prefetching algorithm is straightforward. For the above example, an unused integer register remembers the value of $r11$ at the beginning of the loop. After $r11$ is updated, the address distance is calculated and multiplied by an "iteration ahead" count. Finally this amplified distance is used to prefetch future data along the traversal path (See Fig. 6). This approach has been shown to be useful for linked lists with partially regular strides. As for data structures like graphs and trees, this approach is less applicable if cache misses are evenly distributed along all traversal paths.

3.3. Prefetch Generation

New registers are needed in computing prefetching address. On many RISC machines with *base+offset* addressing mode, the computation of prefetching address can be avoided by folding the prefetch distance into the base address. For example, "*lfetch [r11+80]*". However, due to the lack of this addressing mode in IA64 ISA, we must generate explicit instructions to calculate prefetching address. There are multiple ways to obtain new registers on Itanium: (1) Dynamically allocate more registers using *alloc*, (2) Spill existing registers, (3) Work with the compiler to reserve some registers. In the current implementation, we use the third approach. Specifically, we ask the static compiler to reserve four global integer registers ($r27 - r30$) and one global predicate registers ($p6$) from the IA64 register files. We have also tried the first approach, but that requires the identification of the immediately dominating *alloc*, which may fail if multiple *alloc*s are used in a subroutine. However, the use of reserved registers makes our system less transparent, so we are now looking for a robust mechanism to acquire free registers at the runtime.

Fig. 6 illustrates the inserted prefetch instructions for all three reference patterns in the above examples. Notice for cases A and B, initialization codes must be inserted on top of the loop to preset the prefetch distance. Since accurate miss latency of each cache miss event is available in ADORE, the prefetch distance is easily computed as: **distance** = $\lceil$average latency / loop body cycles$\rceil$. In addition, for small strides in integer programs, prefetch distances are aligned to L1D cache line size (not for FP operations since they bypass L1 cache).

3.4. Prefetch Code Optimization

Prefetch code often exhibits redundancy, hence should be optimized as well. For example, in Fig. 6, case A, one "*lfetch [r27], 12*" is sufficient for both data prefetching and stride advancing. Such optimization is important because it reduces the number of instructions executed and consumes fewer registers.

3.5. Prefetch Scheduling

Prefetch code should be scheduled at otherwise wasted empty slots so that the introduced cost is kept as small as possible. For instance, it would be better if each *lfetch/ld.s* be put in an instruction group having a free memory slot. Ineffective insertion of prefetches may increase the number of bundles and cause significant performance loss. Details of the Itanium microarchitecture can be found in [18][19].

3.6. Implementation Issues

Although architectural state preservation and precise exception handling [17] are among the critical issues to be solved in dynamic optimization systems, the two optimizations, trace layout and prefetching, implemented in ADORE are considered safe. Prefetch instructions use reserved registers and non-faulting loads (*ld.s*), so they do not generate exceptions or change the architecture state. The original program's execution sequence has not been changed either, because the traces are only duplicates of the original code, and the current optimizer does not schedule other instructions. Self-modified code can be detected by write-protecting text pages.

4. Performance Results

4.1. Methodology

To evaluate the performance of runtime prefetching, nine SPEC FP2000 benchmarks and eight SPEC INT2000 benchmarks [32] are tested with reference inputs [2]. Our

[2]Other benchmarks either cannot be compiled properly at all optimization levels or do not have a stable execution time in our experiment. For benchmarks having multiple data inputs, only the first input data is used in our measurement.

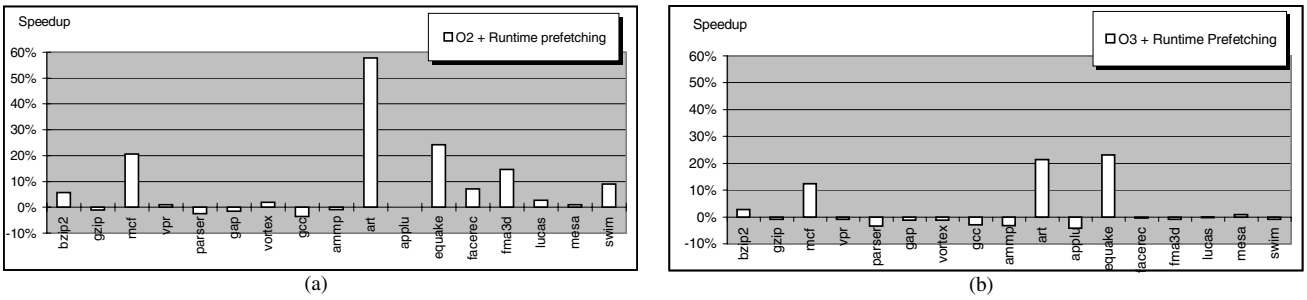


Figure 7. Performance of Runtime Prefetching

test machine is a 2-CPU 900MHz Itanium 2 zx6000 workstation. The Operating system is Redhat Linux 7.2 (kernel version 2.4.18.e25) with glibc 2.2.4. The ORC®compiler v2.0 [26] is chosen to compile the benchmark programs.

4.2. Static Prefetching Guided by Sample Profile

Before evaluating the performance of runtime optimization in the ADORE system, we assess a profile-guided static prefetching scheme in which we modified the existing prefetching algorithm of the ORC compiler to reduce the number of prefetches for loops guided by performance sampling profiles. The sampling profile used here has the same format as that used in runtime prefetching (Section 2.1) except that the runtime prefetcher uses a smaller most recent profile.

The existing prefetching algorithm in the ORC compiler is activated when compiling at O3. It is similar to Todd Mowry's algorithm [24]. Like other compile-time prefetching, this algorithm requires accurate array bounds and locality information. It also generates unnecessary prefetches for loads that might at runtime hit well in the data caches. In our study, we merely modify this algorithm to select loops for prefetching under the profile's guidance. We did not rewrite the whole algorithm to more aggressively prefetch for data reference patterns such as pointer chasing. We believe profile-guided software prefetching can be further improved in such cases.

Using the sampling profiles, the static compiler sorts the delinquent loads in decreasing order of total miss latency. Then these delinquent loads are added one by one to a list until the total latency caused by the loads in the list covers 90% of all profiled cache miss latency. Static prefetching is then invoked as usual (with O3) except that the compiler now generates prefetches only for loops containing at least one delinquent load in that list. Comparing this method with normal O3 optimization, 83% of loops scheduled for prefetching have been filtered out on average (See Table 1). Static code size is reduced by as much as 9%. This result adds evidence to the hypothesis that only a few instructions in a program dominate the memory access latencies during execution, for which runtime prefetching would be ideal. In fact, the reason that there is no obvious speedup from this profile-guided prefetching is because we merely use profile to filter out unnecessary prefetches. Profile-guided software prefetching may attain greater performance if the profile provides sufficient evidence and information to guide more aggressive but expensive prefetching transformations [3].

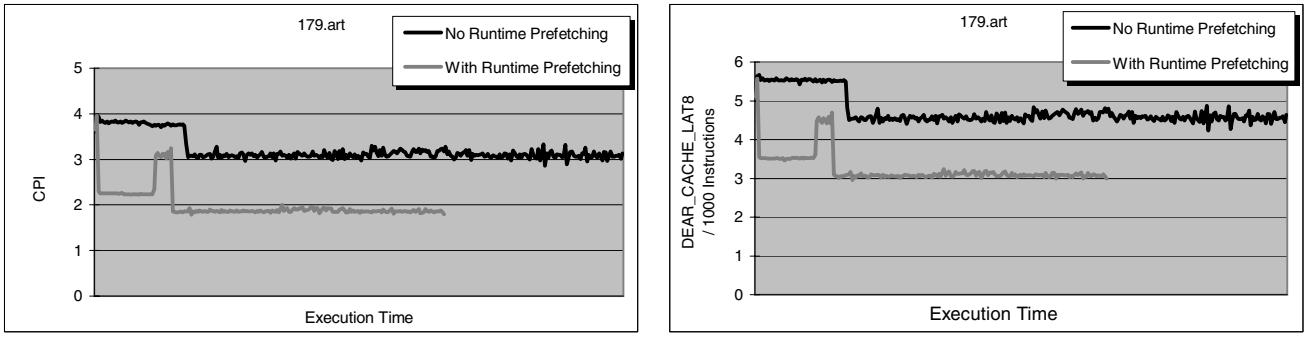
	loops scheduled for prefetch		normalized execution time		normalized binary size	
Spec2000	O3	O3+Profile	O3	O3+profile	O3	O3+Profile
ammp	113	13	1	0.989	1	0.980
applu	52	19	1	0.998	1	0.998
art	39	20	1	0.985	1	0.964
bzip2	65	11	1	1.007	1	0.927
equake	34	4	1	0.997	1	0.992
facerec	94	12	1	0.997	1	0.970
fma3d	1023	39	1	0.996	1	0.990
gap	553	18	1	1.008	1	0.938
gcc	651	21	1	0.993	1	0.986
gzip	85	2	1	1.004	1	0.939
lucas	59	23	1	0.999	1	0.992
mcf	7	3	1	0.986	1	0.973
mesa	583	14	1	0.995	1	0.911
parser	67	5	1	0.990	1	0.958
swim	19	9	1	1.001	1	0.995
vortex	20	0	1	0.995	1	0.999
vpr	120	5	1	0.990	1	0.987

Table 1. Profile Guided Static Prefetching

4.3. Runtime Prefetching

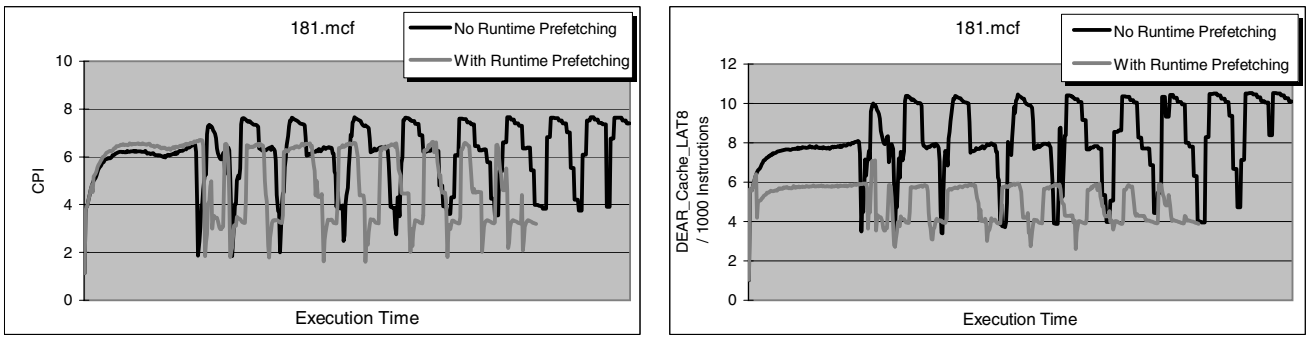
Runtime prefetching is more transparent than profile-guided static prefetching because it needs no extra run to collect profiles. In our test, runtime prefetching is applied to the benchmark programs from two most commonly-used compilations: O2 and O3. As mentioned, at O2 the ORC compiler does not generate static prefetching while at O3 it does. In both compilations, the compiler reserved 4 integer registers, 1 predicate register and disabled software pipelining. We disable software pipelining because our dynamic optimization currently does not handle software-pipelined loops with rotation registers. However, this limitation may

[3]This assumes the cache miss profile collected by training run is able to reliably predict the actual data references.



(a) (b)

Figure 8. Runtime Prefetching for *179.art*



(a) (b)

Figure 9. Runtime Prefetching for *181.mcf*

not significantly change our results and we will discuss the performance impact later in this section. All benchmark programs run reference data inputs.

Fig. 7(a) and Fig. 7(b) illustrate the performance impact of O2+Runtime-Prefetching and O3+Runtime-Prefetching. In Fig. 7(a), 9 out of 17 Spec2000 benchmarks have speedup from 3% to 57%. For the remaining 8 programs that show no benefit from dynamic optimization, the performance differences are around -2% to +1%. A further examination of the traces generated by the dynamic optimizer shows that our runtime prefetcher did locate the right delinquent loads in *applu*, *swim*, *vpr* and *gap*. The failure in improving the performance of these programs is due to three reasons. First, for some programs, the cache misses are evenly distributed among hundreds of loads in several large loops (e.g. *applu*). Each load may have only 2-3% of total latency and their miss penalties are effectively overlapped through instruction scheduling. Furthermore, the current dynamic optimizer can only deal with the top three delinquent loads. With only four integer registers available for the dynamic optimizer, we need a more sophisticated algorithm to handle a large number of prefetches in a loop. Second, some delinquent loads have complex address calculation patterns (e.g. function call or fp-int conversion), causing the dynamic optimizer to fail in computing the stride information (in *vpr*, *lucas* and *gap*). Third, the optimizer may be unable to insert prefetches far enough to hide latencies if the loop contains few instructions and has small iteration count. For integer benchmarks, except for *mcf*, runtime data prefetching has only slight speedup. *gzip*'s execution time is too short (less than 1 minute) for ADORE to detect a stable phase. *vortex* is sped up by 2% but that is partly due to the improvement of I-cache locality from *trace layout*. *gcc*, in contrast, suffers from increased I-cache misses plus sampling overhead and ends up with a 3.8% performance loss. This may be improved by further tuning on trace selection or I-cache prefetching.

As expected, runtime prefetching shows different results when applied to the O3 binaries (Fig. 7(b)). For programs like *mcf*, *art* and *equake*, the current static prefetching cannot efficiently reduce the data miss latency, but runtime-prefetching is able to. The performance improvement is almost as much as those received from O2 binaries. However, the remaining programs have been fully optimized by O3, so the runtime prefetcher skips many traces to optimize since they either don't have cache misses or already

have compiler generated "*lfetch*". For this reason the performance differences for many programs are around -3% to +2%.

Now let's look at an example to understand how runtime prefetching works for these benchmark programs. In Fig. 8, the left graph shows the runtime *CPI* change for *179.art* with/without runtime prefetching (O2 binary). The right graph shows the change of DEAR Load Miss per 1000 instructions. There are two clear phases shown in both graphs. One is from the beginning of the execution; the other starts at about $\frac{1}{4}$ of way in the execution. Phase detection works effectively in this case. The first phase is detected about ten seconds after startup and prefetching codes are applied immediately. Both *CPI* and *DEAR Load Miss Per 1000 instructions* are reduced by almost half. About one and a half minutes later, a phase change occurs followed by the second stable phase till the end. The phase detector catches this second phase too. Since the prefetching effectively reduces the running time, the second lines in both graphs are shorter than the top lines. Fig. 9 is the same type of graph for *181.mcf*. Other programs like *bzip2*, *fma3d*, *swim* and *equake* also exhibit similar patterns.

Only a small number of prefetches have been inserted into the trace code to achieve the above speedup. Table 2 shows the total number of stable phases applied with runtime prefetching and the individual number of the three reference patterns prefetched in each benchmark program (O2 binary). The majority of speedup comes from prefetching for direct/indirect array references. Prefetching for pointer chasing references is not widely applicable because not many LDS (linked data structure) intensive applications exhibit regular stride. For such kind of data structures, runtime prefetching should consider more elaborate approaches such as correlation prefetching [25].

SpecFP2000	ammp	applu	art	equake	facerec	fma3d	lucas	mesa	swim
direct array	0	21	10	6	17	11	6	1	9
indirect array	2	0	6	1	0	2	0	0	0
pointer-chasing	2	0	0	0	0	0	0	0	0
optimized phase #	3	2	2	1	3	4	1	1	1
SpecINT2000	bzip2	gap	gcc	gzip	mcf	parser	vortex	vpr	
direct array	10	3	2	0	0	1	2	1	
indirect array	6	0	0	0	0	0	0	0	
pointer-chasing	0	0	0	0	3	2	0	0	
optimized phase #	2	3	2	0	2	1	2	1	

Table 2. Prefetching Data Analysis

Since we have disabled software-pipelined loops and reserved 4 *gr*s when compiling the benchmarks, we must evaluate the impact to performance. Fig. 10 measures this impact by comparing the original O2 with our restricted O2. For most of the 17 programs, the impact of fewer registers and disabling software pipelining is minor. Four programs show difference greater than 3%. They are *equake*, *mcf*, *facerec* and *swim*. These performance differences come primarily from SWP (Software Pipelining). However, the reason for disabling SWP is that the current dynamic optimizer in ADORE cannot insert prefetches into software pipelined loops where rotation registers are used. In the future, this limitation can be relaxed. At this time, the performance differences can be minimized by applying runtime prefetching directly to these programs compiled with O3, although the speedups for other programs might be lower. On the other hand, for *mcf*, *equake*, *art*, *bzip2*, *fma3d*, *mesa*, *vpr*, and *vortex*, their O2 binaries (without SWP), when optimized by runtime prefetching, are always faster than the O3 binary, whether with SWP or not.

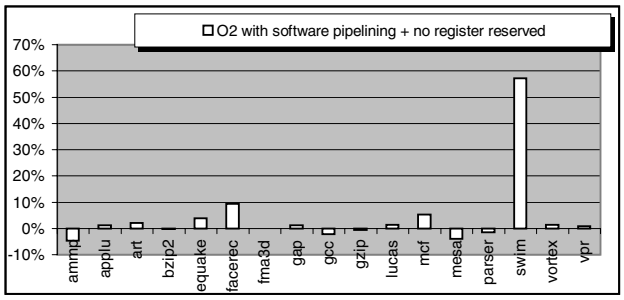


Figure 10. Impacts of Register Count and Software Pipelining to Performance

At the end of this section, we evaluate the runtime overhead incurred by our dynamic optimization. In this system, the major overhead is introduced by continuous sampling, phase detection and trace optimization. Our experience suggests the sampling interval be no less than 100,000 cycle/sample. The phase detector polls the sample buffer in a while-loop. We let it hibernate for 100 milliseconds after each poll to save cpu time. Furthermore, the current working mechanism of our phase detection model prevents trace optimization from being frequent. Consequently, although running on the second processor, the second processor for the dynopt thread is idle almost all of the time [4]. Fig. 11 shows the benchmark program's "real clock time" when prefetch insertion is disabled in ADORE (compared with O2 binary). It is measured using the shell command *time*. The "user cpu time" are not shown here since they are always smaller than "real clock time" in our experiment. These results demonstrate that the extra overhead of ADORE system is trivial.

5. Related Work

5.1. Software Prefetching

Many software based prefetching techniques have been proposed in the past few years. Todd C. Mowry et al. first

[4]The same speedup can be achieved on a single cpu system.

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)
0-7695-2043-X/03 $ 17.00 © 2003 IEEE

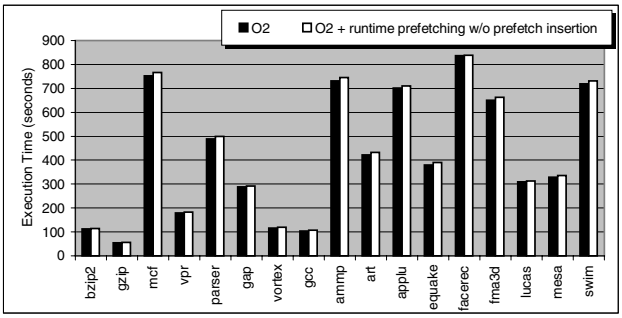


Figure 11. Overhead of Runtime Prefetching

presented a general compiler prefetching algorithm working effectively with scientific program [24]. Later Luk and Mowry proposed a compiler-based prefetching scheme for recursive data structures [22]. This requires extra storage at runtime. Santhanam et al. discussed the major implementation issues of compile time prefetching on a particular RISC processor: HP-PA8000 [30]. Jump Pointer [29], which has been used widely to break the serial LDS (linked data structure) traversal, stores pointers several iterations ahead in the node currently visited. Other research tried to improve spatial locality and runtime memory reference efficiency with improved compilation methods. In a recent work, Gautam Doshi et al. [14] discussed the downside of software prefetching and exploited the use of rotating registers and predication to reduce the instruction overhead.

5.2. Profile Guided Software Prefetching

Software prefetching is ineffective in pointer-based programs. To address this problem, Chi K. Luk et al. presented a Profile Guided Post-Link Stride Prefetching [23] using a stride profile to obtain prefetching guidance for the compilers. In an earlier research, Luk and Mowry presented a correlation-profiling scheme [25] to help software-based techniques detect data access correlations.

Recently, Chilimbi and Hirzel [8] explored the use of burst profiling to prefetch for hot data stream in their dynamic optimization framework, where three stages (profiling, optimization, hibernation) repeat in a fixed timing sequence. In the profiling stage, hot stream patterns are catched in the form of <pc, addr> pairs. The optimizer, in the next stage, generates detection code and prefetches into the procedures duplicated from the program's code segment. The detection code behaves like a finite state machine that matches the prefixes of a hot data stream and triggers prefetching for the suffixes. Although their framework is also a real dynamic system, the binaries must be statically instrumented for profiling and pre-linked with a dynamic optimizer library, which is not required by ADORE.

5.3. Hardware Prefetching

Among the many studies of Hardware prefetching [12][1][20][11], Collins, et al., attempted to prefetch delinquent loads in a multithreaded architecture based on Itanium ISA [12]. Annavaram, et al., proposed a hardware Dependence Graph Precomputation mechanism (DGP) [1] aiming to reduce the latency of a pending data cache miss.

Although hardware techniques can exhibit more flexibility and aggressiveness in data prefetching, they may be expensive to implement. Design and performance evaluation of the above schemes are mostly carried out by simulations.

5.4. Dynamic Optimization Systems

Software Runtime Optimization Systems are commonly seen in Java Virtual Machines (JVM) [9][2][27], where Just-In-Time engines apply recompilation at runtime to achieve higher performance. On these systems, JIT usually employs adaptive profile-feedback optimization (e.g. by runtime instrumentation or interpretation) to take advantages of Java programs' dynamic nature.

Dynamo [3] is a transparent dynamic native-to-native optimization system. Dynamo starts running a statically compiled executable by interpretation, waiting for hot traces to show up. Once hot traces are detected, Dynamo stops the program and generates code fragments for these traces. Subsequent execution on the same trace will be redirected to the newly optimized code in the fragment cache. The interpretation approach in Dynamo is expensive, and as a result, Dynamo tries to avoid interpretation by converting as many hot traces as possible to the fragment cache. To achieve this goal, it uses a small threshold to quickly determine if a trace is hot. This approach often ends up with translating too much code and less effective traces. Therefore in a recent work called DynamoRIO [5], this feature has been changed. Other research on dynamic optimization also explored dynamic translation [7][10], continuous program optimization [21] and binary transformation [16].

6. Conclusion and Future Research

In this paper we propose a runtime prefetching mechanism in a dynamic optimization system. The overhead of this prefetching scheme and the dynamic optimization is very low due to the use of sampling on the Itanium processor's performance monitoring unit. Using this scheme, we can improve runtime performance by as much as 57% on some SPEC2000 benchmarks compiled at O2 for Itanium-2 processors. In contrast, compile time prefetching approaches need sophisticated analysis to achieve similar or lower performance gains. For binaries compiled at O3 with static prefetching, our system can improve performance by

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)
0-7695-2043-X/03 $ 17.00 © 2003 IEEE

as much as 20%. In this paper, we also examined a profile-guided static prefetching using the HPM based sampling profiles. The results show that a minor modification of existing static prefetching to use cache miss profiles can avoid generating most of the unbeneficial prefetches without losing performance.

For future work on runtime prefetching, we plan to enhance our algorithm to also handle software pipelined loops. The current phase detection scheme in our system does not work very well on programs with rapid phase changes, hence needs to be improved. Finally, we are investigating the possibility of adding selective runtime instrumentation to collect information not available from HPM.

References

[1] M. Annavaram, J. M. Patel, and E. S. Davidson. Data Prefetching by Dependence Graph Precomputation. In *ISCA-28*, pages 52–61. ACM Press, 2001.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive Optimization in the Jalapeño JVM. In *OOPSLA'00*, pages 47–65. ACM Press, 2000.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A Transparent Dynamic Optimization System. In *PLDI'00*, pages 1–12. ACM Press, 2000.

[4] T. Ball and J. R. Larus. Efficient Path Profiling. In *Micro-29*, pages 46–57. IEEE Computer Society Press, 1996.

[5] D. Bruening, T. Garnett, and S. Amarasinghe. An Infrastructure for Adaptive Dynamic Optimization. In *CGO'03*, pages 265 –275, 2003.

[6] H. Chen, W.-C. Hsu, J. Lu, P.-C. Yew, and D.-Y. Chen. Dynamic Trace Selection Using Performance Monitoring Hardware Sampling. In *CGO'03*, pages 79–90, 2003.

[7] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates. FX!32 a Profile-Directed Binary Translator. *Micro, IEEE*, 18(2):56–64, Mar/Apr 1998.

[8] T. M. Chilimbi and M. Hirzel. Dynamic Hot Data Stream Prefetching for General-Purpose Programs. In *PLDI'02*, pages 199–209. ACM Press, 2002.

[9] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing JUDO: Java Under Dynamic Optimizations. In *PLDI'00*, pages 13–26. ACM Press, 2000.

[10] R. S. Cohn, D. W.Goodwin, and P. G. Lowney. Optimizing Alpha Executables on Windows NT with Spike. *Digital Technical Journal*, 9(4), Jun 1998.

[11] J. Collins, S. Sair, B. Calder, and D. M. Tullsen. Pointer Cache Assisted Prefetching. In *Micro-35*, pages 62–73. IEEE Computer Society Press, 2002.

[12] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen. Speculative Precomputation: Long-Range Prefetching of Delinquent Loads. In *ISCA-28*, pages 14–25. ACM Press, 2001.

[13] A. S. Dhodapkar and J. E. Smith. Managing Multi-Configuration Hardware via Dynamic Working Set Analysis. In *ISCA-29*, pages 233–244. IEEE Computer Society, 2002.

[14] G. Doshi, R. Krishnaiyer, and K. Muthukumar. Optimizing Software Data Prefetches with Rotating Registers. In *PACT'01*, pages 257 –267, 2001.

[15] Intel®C++ Compiler for Linux, http://www.intel.com/software/products/compilers/clin/.

[16] A. Edwards, A. Srivastava, and H. Vo. Vulcan: Binary Transformation in A Distributed Environment. Technical Report MSR-TR-2001-50, Apr 2001.

[17] M. Gschwind and E. Altman. Optimization and Precise Exceptions in Dynamic Compilation. *ACM SIGARCH Computer Architecture News*, 29(1):66–74, 2001.

[18] Intel Corp. *Intel®IA-64 Architecture Software Developer's Manual*, revision 2.1 edition, Oct 2002.

[19] Intel Corp. *Intel®Itanium®2 Processor Reference Manual for Software Development and Optimization*, Jun 2002.

[20] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *ISCA24*, pages 252–263. ACM Press, 1997.

[21] T. Kistler and M. Franz. Continuous Program Optimization: Design and Evaluation. *Computers, IEEE Transactions on*, 50(6):549–566, Jun 2001.

[22] C.-K. Luk and T. C. Mowry. Compiler-Based Prefetching For Recursive Data Structures. In *ASPLOS-7*, pages 222–233. ACM Press, 1996.

[23] C.-K. Luk, R. Muth, H. Patil, R. Weiss, P. G. Lowney, and R. Cohn. Profile-Guided Post-link Stride Prefetching. In *ICS-16*, pages 167–178. ACM Press, 2002.

[24] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of A Compiler Algorithm for Prefetching. In *ASPLOS-5*, pages 62–73. ACM Press, 1992.

[25] T. C. Mowry and C.-K. Luk. Predicting Data Cache Misses in Non-Numeric Applications Through Correlation Profiling. In *Micro-30*, pages 314–320. IEEE Computer Society Press, 1997.

[26] Open Research Compiler for Itanium™Processor Family, http://ipf-orc.sourceforge.net.

[27] M. Paleczny, C. Vick, and C. Click. The Java™HotSpot Server Compiler. In *Java™VM'02*, 2001.

[28] http://www.hpl.hp.com/research/linux/perfmon.

[29] A. Roth and G. S. Sohi. Effective Jump-Pointer Prefetching for Linked Data Structures. In *ISCA-26*, pages 111–121. IEEE Computer Society Press, 1999.

[30] V. Santhanam, E. H. Gornish, and W.-C. Hsu. Data Prefetching on the HP PA-8000. In *ISCA-24*, pages 264–273. ACM Press, 1997.

[31] T. Sherwood, S. Sair, and B. Calder. Phase Tracking and Prediction. In *ISCA-30*, Jun 2003.

[32] SPEC: http://www.spec.org/cpu2000.

[33] A. Stoutchinim, J. N. Amaral, G. R. Gao, J. C. Dehnert, S. Jain, and A. Douillet. Speculative Prefetching of Induction Pointers. In *CC-10*, April 2001.

[34] Z. Wang, D. Burger, S. K. Reinhardt, K. S. Mckinley, and C. C. Weems. Guided Region Prefetching: A Cooperative Hardware/Software Approach. In *ISCA30*, 2003.

[35] Y. Wu. Efficient Discovery of Regular Stride Patterns in Irregular Programs and Its Use in Compiler Prefetching. In *PLDI'02*, pages 210–221. ACM Press, 2002.

Proceedings of the 36th International Symposium on Microarchitecture (MICRO-36'03)
0-7695-2043-X/03 $ 17.00 © 2003 IEEE