Journal of Instruction-Level Parallelism 6 (2004) 1-24 Submitted 2/04; published 4/04

Design and Implementation of a Lightweight Dynamic Optimization System *

Jiwei Lu JIWEI@CS.UMN.EDU
Howard Chen CHENH@CS.UMN.EDU
Pen-Chung Yew YEW@CS.UMN.EDU
Wei-Chung Hsu HSU@CS.UMN.EDU
Department of Computer Science and Engineering
University of Minnesota, Twin Cities
Minneapolis, MN

Abstract

Many opportunities exist to improve micro-architectural performance due to performance events that are difficult to optimize at static compile time. Cache misses and branch mis-prediction patterns may vary for different micro-architectures using different inputs. Dynamic optimization provides an approach to address these and other performance events at runtime. This paper describes a software system of real implementation that detects performance problems of running applications and deploys optimizations to increase execution efficiency. We discuss issues of detecting performance bottlenecks, generating optimized traces and redirecting execution from the original code to the dynamically optimized code. Our current system speeds up many of the CPU2000 benchmark programs having large numbers of D-Cache misses through dynamically deployed cache prefetching. For other applications that don't benefit from our runtime optimization, the average cost is only 2% of execution time. We present this lightweight system as an example of using existing hardware and software to deploy speculative optimizations to improve a program's runtime performance.

1. Introduction

Recent work in dynamic optimization has shown that a run-time system can improve program performance by performing optimizations that are difficult to deploy statically due to dynamically linked libraries, micro-architecture specific features, and inaccurate run-time profiles [1],[2],[3],[4],[5],[6]. Existing dynamic optimization schemes have focused on finding the most frequent execution paths through interpretation or instrumentation and making these "hotspots" more efficient. Since the time spent optimizing a program incurs overhead to the total program execution, focusing on the most frequently executed code can ideally gain the most benefit for the work done. However, there are more things the dynamic optimization can do. Dynamic optimization presents an opportunity to apply more aggressive optimizations by allowing optimizations to be applied speculatively. It also has potential to remove any previously applied optimizations that do not provide a performance benefit. On the other hand, collecting information to guide more aggressive optimizations is not practical through interpretation or instrumentation based profiling, because hardware support is

*. 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.

©2004 AI Access Foundation and Morgan Kaufmann Publishers. All rights reserved.

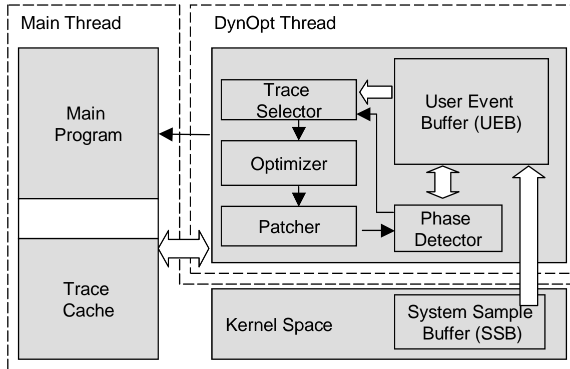


Figure 1: ADORE Framework

required to efficiently detect changes in cache, branch prediction, and many other types of performance behavior. In addition, overhead limitations of interpretation and instrumentation make it expensive to continuously monitor the program, including optimized code, for the entire execution [3]. Our work describes a system that continuously monitors program execution using existing performance monitoring hardware, and deploys optimizations only to hot-spot specific performance bottlenecks. This scheme allows us to apply optimizations only to existing program bottlenecks and continue to monitor program behavior after applying these optimizations, creating a framework for applying more aggressive optimizations. In our current dynamic optimization system, we use hardware performance monitoring registers on Itanium ®2 machines to detect hot regions and performance bottlenecks in place of instrumentation and interpretation schemes presented in the past. We compile a set of SPEC2000 benchmarks and use a performance-monitoring tool [7] provided with the Linux kernel to detect performance bottlenecks in these benchmarks. By applying runtime optimizations such as data cache prefetching, our system is able to speed up some programs having large cache misses by 3%-106% while limiting the overhead to only 1-2% on average. The remainder of the paper is organized as follows. Section 2 introduces the framework of our runtime optimization system including performance monitoring, sampling, trace selection and phase detection. Section 3 discusses the runtime optimization and trace patching. In Section 4, we present the performance evaluation of runtime optimization. Section 5 and Section 6 contain the related works, conclusion and future work.

2. System Framework

2.1 Overview

ADORE (Adaptive Object code RE-optimization) is a trace-based user-mode dynamic optimization system. Unlike other dynamic optimization/translation systems [1],[2],[8],[9],[10], [11],[12], it is based on Hardware Performance Monitoring (HPM). It is implemented as a shared library on Linux for IA64 that can be automatically linked to the application at startup. Figure 1 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 run-

```
int BP_SYM(__libc_start_main) (int (*main)(int, char **, char **), ...)
{
      void (*dyn_open)(void);
      void (*dyn_close)(void);
      void * dyn_handle;

      ...

      dyn_handle = dlopen ("libdyn.so", RTLD_LAZY);

      if (dyn_handle) {
          dyn_open  = dlsym(dyn_handle, "dyn_open_session");
          dyn_close = dlsym(dyn_handle, "dyn_close_session");

          on_exit(dyn_close, NULL);

          (*dyn_open)();
      }

      exit ((*main) (argc, argv, __environ));
}
```

Figure 2: Entry-point Startup Code

time optimization. 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 adding our startup code as shown in Figure 2.

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 area for the original process. This is where the trace cache resides to keep the optimized traces. Second, it initiates *perfmon*. *Perfmon* is a generic kernel interface developed by HP labs [7]. It provides controls for setting and retrieving performance counter values from the Itanium 2's Performance Monitoring Unit (PMU). *Perfmon* resets the PMU, determines the sampling rate and creates a kernel buffer, which is called System Sampling Buffer (SSB) in Fig. 1. Third, *dyn_open* installs a signal handler (call back function) to copy all the sample events from the SSB to a user buffer every time the SSB overflows. The user buffer is a larger circular buffer, shown as the User Event Buffer (UEB) in Fig. 1. Finally, *dyn_open* creates a new thread for dynamic optimization that has the same lifetime as the main thread. *dyn_close* is registered by the library function *on_exit* so that it can be invoked at the end of the main program execution. *dyn_close*'s job is to free the memory and notify the optimizer thread that the main program is complete.

2.2 Performance Monitoring on Itanium 2 Processor

ADORE is an HPM based dynamic optimizer. It depends on continuous hardware samples to accomplish trace selection and to guide a variety of runtime optimizations. The collected

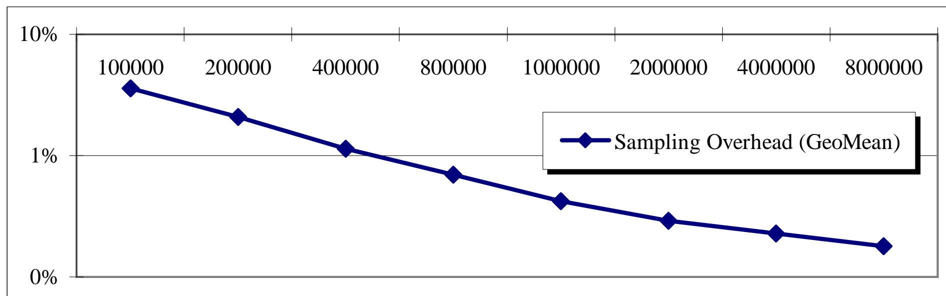


Figure 3: Sampling and Phase Detection Overhead. It shows the overhead (percentage) in Geometric Mean for a set of CPU2000 benchmark programs (Note that x axis is sampling intervals in cycles, y axis is the percentage of execution time in logarithmic scale). The longer the sampling interval, the less overhead.

samples incorporate most of the information that the dynamic optimizer needs. On the Itanium 2 processor, 4 performance counters can monitor hundreds of performance metrics that give users a complete snapshot of a program's runtime performance from all aspects. Through this HPM sampling model, a dynamic optimizer is able to shift two critical and time consuming tasks to the hardware: profiling and diagnosing. In addition to *CPU cycles*, *Branch Mis-Prediction rate*, *Pipeline Flush*, *Memory Stalls* and many other counters, the PMU on Itanium 2 processor implements registers for special event based sampling. For instance, the *Branch Trace Buffer* (BTB) is a set of eight registers that bank the four most recent branch outcomes and source/target address information. The *Data Event Address Registers* (DEAR) and the *Instruction Event Address Registers* (IEAR) hold the most recent data/instruction related events, such as *D/I-Cache misses*, *D/I-TLB misses*, and *ALAT misses* [13]. Continuous HPM sampling in the ADORE system is achieved through a signal handler communicating with *perfmon*. This signal handler is a callback function that copies the sample data from SSB to UEB when *perfmon* raises a signal indicating the overflow of SSB. UEB is eight times the size of the SSB in our current system. We call the time period for the SSB to overflow a *profile window*. Hence the UEB holds the most recent eight *profile windows*. Usually one *profile window* consists of $4K$ samples. Each sample is in the form of an n-tuple: < *sample index*, *PC address*, *CPU cycles*, *Retired Instruction Count*, *D-Cache Miss Count*, *I-Cache Miss Count*, BTB values and D/I-EAR values >. After every $4K$ sampling intervals elapse, a new *profile window* arrives. To avoid spin-wait, the dynamic optimizer sleeps until a *profile window* arrives.

2.3 Sampling Rate

The sampling rate is very crucial because it not only controls the granularity of runtime profiles, but it also affects the decision making of ADORE and its response time as well. Higher sampling rates bring more overhead to the entire system. Figure 3 shows the cost of HPM sampling plus phase detection of ADORE with various sampling intervals (*CPU cycles*) for a set of CPU2000 benchmark programs. As we can see, sampling intervals larger than $200,000$ cycles/sample incur negligible overhead. With shorter sampling intervals, the

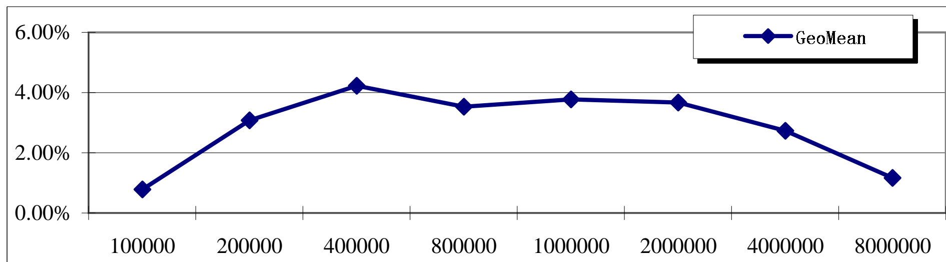


Figure 4: Average (geometric mean of CPU2000 benchmarks) speedup achieved by dynamic optimization with different sampling intervals.

overhead rises, and it becomes unacceptable when fewer than $100,000$ cycles/sample are taken.

However, a larger sampling interval (i.e. smaller sampling rate) doesn't necessarily ensure better performance of an HPM based dynamic optimizer. The optimizer needs a reasonable number of samples to truly reflect the performance bottlenecks in an appropriate time interval. Too few samples may lead to incorrect judgments. Further, a slightly higher sampling rate results in a higher arrival rate of new *profile-window*s and improves the system's sensitivity to performance changes. Fig. 4 shows the net speedup achieved by ADORE on the same set of CPU2000 benchmarks with various sampling intervals (from $10,000$ cycles/sample to $8,000,000$ cycles/sample). Best performance is obtained when the sampling interval is around $400,000$ cycles/sample. Hence the sampling rate is usually set to be $300,000$ to $500,000$ cycles/sample.

2.4 Runtime Trace Selection

In our work, a *trace* is the same as a *superblock* [14], but different from a *fragment*[15]. It is a single-entry, multiple-exit code sequence, which may contain tens or even hundreds of basic blocks. ADORE's goal of binary optimization at runtime is simple: select a small set of the hottest traces (2% of original code size on average) that represent the hottest code in the most recent time intervals of a program's execution and optimize them based on the profile collected through HPM sampling. To find and construct these hot traces, we first build a *path profile* [16] using the BTB samples. An example of a BTB sample is shown in Figure 5. The BTB is a circular buffer where the four most recent branch source/target pairs are stored. These four pairs represent a unique execution path at runtime. If this unique path shows up enough times in the profiles, we consider it to be a hot path and use it to build traces. Hot path fragments are often linked together to form larger paths in trace selection. Trace selection starts from the hottest branch target address and extends the current trace as far as the path profile may guide. To expedite the path lookup from the samples, we build a special hash-table for path search. This hash-table is referenced using the three most recent branch source addresses and the next branch address in the current trace selection. If there is one hottest path fragment that matches the same four branch addresses, the fourth branch's target address (predicted as taken) will be returned, and the new basic block from that address can be added into the current trace. If only the first three

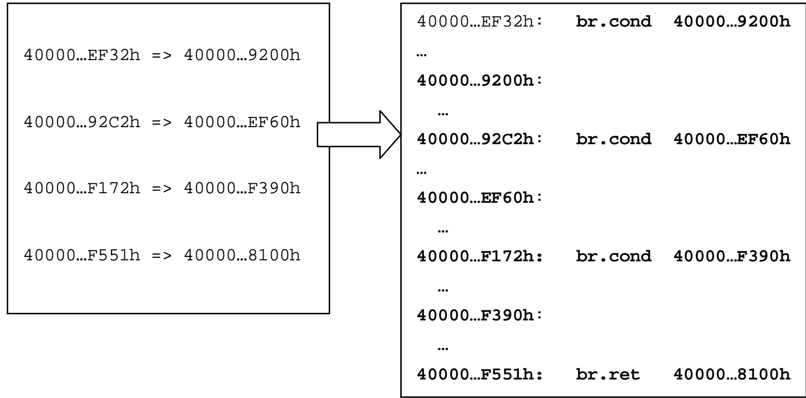


Figure 5: BTB Sample (left) and Its Corresponding Path

branch addresses can be matched with a hot path fragment, the fourth branch is usually predicted as fall-through. In other cases, the hash-table tries to return sub-optimal results (i.e. the hottest path that begins with only the one or two most recently taken branch addresses). Except for that, ADORE also creates two other smaller hash tables for simple branch source/target and target/source mapping. The total space overhead for those tables is around 1-2MB and will not be larger than 10MB, because the number of samples that we handle is fixed (there are only eight *profile windows*).

To reduce the impact of sampling error, we set a threshold Tp for the relative frequency of each path fragment (e.g. $Tp = 90\%$). In this sense, not all path fragments are used. Fragments with relative frequency less than Tp will not be considered in trace selection. Therefore, only the hottest code region will be selected for traces.

Instructions along the hottest path are decoded into a low-level IR and added into the current trace until a trace stop point is reached. Trace stop points include function returns, back-edge branches (i.e. a loop), and conditional branches whose taken/fall-through bias is balanced. On the Itanium 2 processor, function call/returns implicitly shift the register stack. If trace selection goes across them, it must preserve the change of the register stack, which is hard in user mode. Fortunately, we can assume that all function calls are fall-through branches because sooner or later most of the function calls will return to their call sites. For the hot paths in the function body, as long as the function is called frequently, trace selection will select them as additional traces into the trace cache.

When a stop point indicates that the current trace has to end, ADORE will add it into the trace queue and prepare to select the next trace. After all traces have been constructed, the runtime optimizer starts. The PMU provides the information we need to choose the hottest traces. However, to predict which traces will continue to dominate execution time in the future and prevent optimizing traces with performance issues that only occur for very short periods of time, we perform additional work to estimate when program behavior reaches stability.

2.5 Phase Detection

Modern processor design often includes hardware that can adapt to the system's runtime behavioral change and achieve performance benefit. For instance, some processors can automatically raise or lower the CPU clock rate to reduce power consumption [17]. A program during its runtime may also exhibit significantly different behavior, in terms of *CPI*, branch miss prediction, cache miss rate, and so forth. Hence a *phase* is defined to be a time period in which there is little performance variance in a running program's behavior. Recent research [18],[19],[20] extensively explored phase detection schemes for hardware and software implementations. Characteristics of some well-known methods are carefully evaluated in Dhodapkar's work [21]. These approaches detect a program's phase change accurately with good sensitivity. The phases detected by them usually exhibit good performance stability and suitable length.

An interesting property of program phases is that a high-level phase is often made up of a number of small low-level phases. For different purposes, a phase detector can decide to detect small sub-phases, or large phases. Hardware approaches usually recognize phases of finer granularity.

A dynamic optimizer can benefit from phase detection. By detecting the phase change, the dynamic optimizer may evict code fragments optimized for the prior phases and start the new optimization. This new optimization can use completely different strategies depending on the new phase's behavior. In HP's Dynamo system [1], when the fragment cache goes through a period in which a large number of new created fragments occur, the system will consider this to be a phase change and flush its fragment cache completely. However, such implicit phase detection is inadequate for ADORE since it optimizes only a selected set of hot traces. After the optimization is done, the optimizer will hibernate. Therefore we need to design an explicit phase detector to wake up the optimizer to apply new optimizations when new phases arrive.

Before doing this, we must first define a set of goals that a dynamic optimizer wants the phase detector to achieve. This phase detector should be less sensitive and detect only major phase changes. Optimizing short phases is not cost effective for a software-based dynamic optimizer. As long as the real phase change hasn't occurred, it may tolerate random performance variance at an appropriate level. Second, since most of the optimization requires a fair amount of samples to be collected, the phase detector needs to slightly delay announcing the occurrence of a new phase after detecting it. Third, this phase detection must be efficient and easy to implement.

To fulfill these goals, the latest version of ADORE implements a state-driven phase detector. It works as follows: once a new *profile window* arrives, ADORE calculates the mean PC address (PC_{center}) for all the samples in it. For the prior 7 *profile windows* (The system only keeps 8 *profile windows*), it calculates the expectation **E** and standard deviation **D** of the 7 PC_{center}s . After that a state machine will check the current PC_{center} to see whether it has drifted away from the bound of [**E**-**D**, **E**+**D**]. If the out-of-bound distance is larger than a threshold, e.g. 80% of **D**, and is followed by another difference larger than a second threshold, e.g. 30% of **D**, it enters a *Phase_Change* state. Similarly, after two *profile windows* having their PC_{center}s within the bound, the state machine enters a *Stable_Phase_Ready* state. After staying in this state for one more *profile window*, a

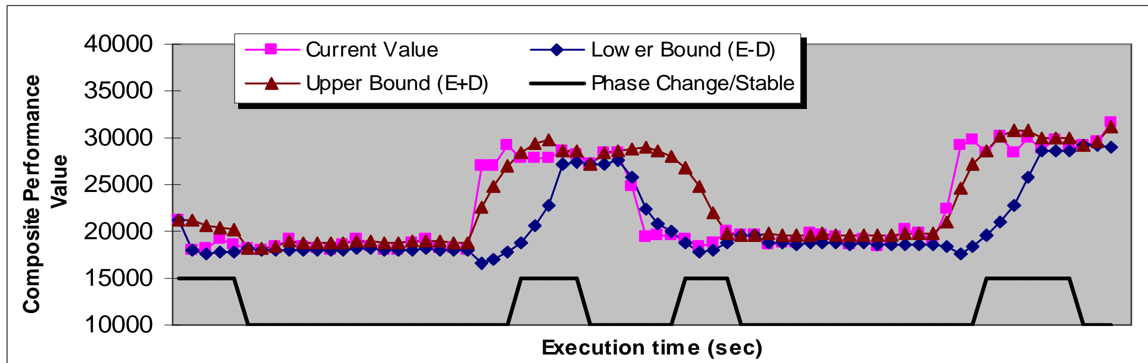


Figure 6: Phase Detection for *256.bzip2*

Phase_Stable signal is raised to trigger the runtime optimizer. This one more *profile window* ensures that enough samples for the new stable phase have been collected. Fig. 6 depicts this procedure of detecting phases for *256.bzip2*. The straight lines at the bottom represent phase changes (upper line) and stable phases (floor line).

Further, PC_{center} may not be enough for all the applications. For phases of huge granularity, the change of working set may not necessarily match the change of other performance characteristics, such as *CPI*. Accurate phase change can be observed by using the composite of *CPI*, PC_{center} and other performance counter values. That's why "composite performance value" ("$PC_{center} \times CPI$" at present) was used to detect phase changes of *256.bzip2* (Fig. 6). However, for most CPU2000 benchmark programs, using PC_{center} to detect phase change is enough.

3. Trace Optimization

3.1 Dynamic Register Allocation

Many runtime optimization techniques require the insertion of new code. For example, data cache prefetching inserts code to pre-compute data addresses for future memory references, and dynamic instrumentation instruments instructions to store the collected runtime profiles. These optimizations require the use of registers. In our system, although trace selection translates the binary instructions into *IR*s, all instructions still keep their original register mapping. To acquire new registers on the Itanium system, we look at three approaches: (a) Spill/Restore registers. (b) Reserve registers by compiler (c) Allocate registers using the IA64's *alloc* instruction [22]. The first approach itself needs one register to store spill address and also has the highest overhead, hence it becomes the last choice. In our current work, the optimizer assumes that static compilers reserve some free registers for dynamic optimization. But the number of the free registers is limited because reserving too many registers will incur noticeable performance degradation of statically compiled binaries. In addition, we have evaluated using *alloc* to allocate up to 16 registers for trace optimization. Details of this method and its relative overhead are discussed in subsequent paragraphs.

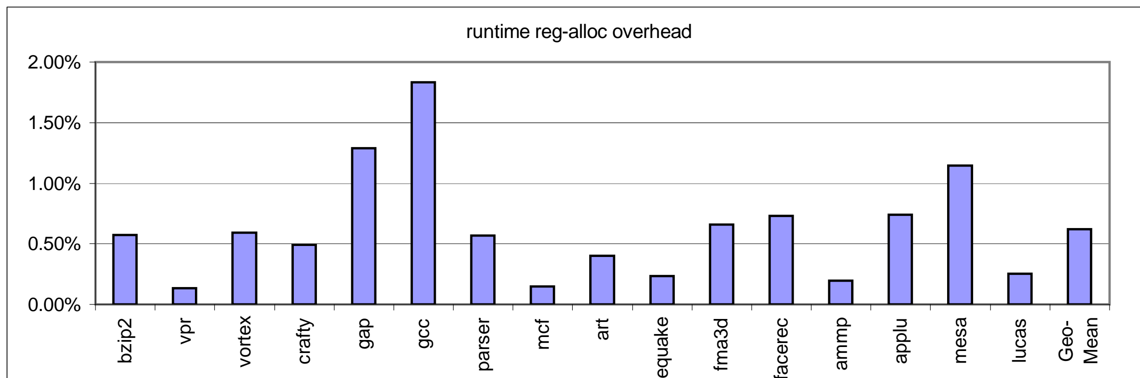


Figure 7: Overhead of using *alloc* to allocate registers at runtime for trace optimization (y axis is the percentage of cost on execution time.

Runtime Register Allocation Scheme IA64 ISA provides a special instruction called *alloc* for programs to setup the current register stack frame [22]. It can also be used in runtime to adjust the current stack frame. We give an example here to illustrate the dynamic register allocation scheme used in ADORE. In a loop-type trace, certain optimizations need m registers. By scanning backwards within the code in the current function's scope, one *alloc* for the whole function at the function's entry point can be located. ADORE duplicates this *alloc* at the entry of this loop trace and increases the number of *output* registers from the original number n to $n + m$. Since a trace is a single entry, multi-exit code block, this ensures that anytime this trace is executed, the code inside the loop body will always have m free registers. The overhead of this approach is usually trivial, because this instruction only executes once outside the loop body for each execution of the trace. This process is invoked only when the trace demands an optimization that requires registers. To handle the case where there are multiple traces in the same function scope, we build a small hash-table to speedup the scanning for the *alloc* instruction.

Runtime Register Allocation Overhead Figure 7 shows the overhead of using *alloc* to allocate registers. The overhead here includes the overhead of scanning for the *alloc* instructions, and the overhead of executing the *alloc* itself after deploying the trace into the trace cache (i.e. the *alloc* is inserted in the trace). We observe no noticeable overhead across all the tested benchmarks, except for *gcc*, *gap* and *mesa*.

Implementation Issues However, there are some shortcomings of using *alloc* to dynamically allocate registers. First, there is no coding convention that forces a compiler to generate only one *alloc* at the beginning of each subroutine. Second, if the trace to be optimized is a loop and has calls to other functions, the callee functions can possibly change the values of the new registers being used by the optimization (because of register stack overlapping between *caller* and *callee* functions). This may cause the optimization to fail completely. Finally, if the current function already uses all registers, there will be nothing left for the *alloc*. Therefore, although so far we haven't encountered any problem with this method, we still have to explore safer and better methods for allocating registers.

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

Figure 8: Matrix Multiplication

3.2 Runtime Data Cache Prefetching

3.2.1 Motivation

As we know, software controlled data cache prefetching [23] 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 precise cache miss profiles and miss-address information. In this section, we will discuss challenges faced in static data prefetching. From section 3.2.2 we will describe how to keep the efficacy of software data prefetching by using runtime cache-miss profiles in the ADORE system.

Impact of Program Structures We first give a simple example to illustrate several issues involved in software controlled data cache prefetching. Fig. 8 shows a typical loop nest used to perform a matrix multiplication. Experienced programmers know how 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 the Itanium 2 system in this study: the Intel®C/C++ Itanium ^{TM}Compiler (i.e. *ECC V*7.0) [24] and the *ORC* ®open research compiler 2.0 [25]. Both compilers have implemented static cache prefetching optimizations that are turned on at *O*3. 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 declare the three arrays as global variables, the *ECC* compiler generates much more efficient code (over five times 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 analysis (such as interprocedural analysis) to conduct efficient and effective cache prefetching.

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,

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

Figure 9: DAXPY

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. The reason is, very often the sub-matrix to be processed is initially 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. Another such well-known example is the *memcpy* library routine [1].

Impact of Micro-architectural Constraints Micro-architecture can also limit the effectiveness of software controlled cache prefetching. For example, the issue bandwidth of memory operations, the memory bus and bank bandwidth, 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 Figure 9, for example. On the latest Itanium 2 processor, two iterations of this loop can be computed in one cycle (2 *ldfpds*, 2 *stfds*, 2 *fmas*, 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.

In general, stride-based prefetching is easy to perform efficiently. Prefetching for pointer chasing references and indirect memory references [26],[27],[28],[29],[30] 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. Profile-guided software prefetching may attain greater performance if the profiles provide sufficient evidence and information to guide more aggressive but expensive prefetching transformations, but this assumes the cache miss profiles collected by training runs are able to reliably predict the actual data references.

Due to the above reasons, we attempt to conduct data cache prefetching at runtime through a dynamic optimization system.

3.2.2 Runtime Prefetching Overview

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

1. A compiler may generate multiple versions of *memcpy*

```
// i++; a[ i++]=b;          // c = b[a[k++] – 1];      // tail = arcin→ tail;
// b= a[ i++];                                        // arcin = tail→ mark;

Loop:                       Loop:                     Loop:
  ...                         ...                       ...
  add  r14= 4, r14            ld4  r20=[r16], 4         add  r11= 104, r34
  st4  [r14] = r20, 4         add  r15 = r25, r20       ld8  r11= [r11]
  ld4  r20 = [r14]            add  r15 = –1, r15        ld8  r34= [r11]
  add  r14 = 4, r14           ld1  r15=[r15]            ...
  ...                         ...                       br.cond  Loop
  br.cond  Loop               br.cond  Loop

 A. direct array             B. indirect array         C. pointer chasing
```

Figure 10: Data Reference Patterns and Dependence Code Slices

is the major optimization that has been implemented. Just as the traditional software prefetching, our runtime optimizer inserts the prefetching code directly into the traces to hide large cache miss latency after identifying 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.

3.2.3 Tracking Delinquent Loads

In the current sampling module, each collected sample contains the latest data cache miss event with latency greater than 8 cycles. On Itanium based systems, this latency implies an *L2* or *L3* cache miss. 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 instruction 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). Prefetching is applied only to those delinquent loads with the greatest percentage of overall latency (e.g. $\geq 2\%$ of total miss latency of the program).

3.2.4 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. Figure 10 gives examples of these three data reference patterns. Delinquent loads are highlighted with bold fonts. To recognize which patterns they belong to, the runtime prefetcher analyzes the dependent instructions for the address calculation of each delinquent load. For stride-based array references, prefetcher usually prefetches data for a few loop iterations ahead. For pointer-based references, since they are usually difficult for software prefetching, a dynamic solution similar to the profile-guided prefetching technique proposed by Youfeng Wu [30] is used to help approximate the data traversal of pointer-chasing references in a sequen-

tial way. In contrast, our approach doesn't require pre-run profiling because the runtime prefetcher knows which load instruction has large cache miss penalty.

3.2.5 Prefetch Generation

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 (e.g. "*lfetch [r11+80]*"). However, due to the lack of this addressing mode in IA64 ISA, we must generate explicit instructions to calculate the prefetching address. Moreover, for cases A and B in Figure 10, initialization codes are usually required to preset the prefetch distance prior to the loop. Since an accurate miss latency of each cache miss event is available in ADORE, the prefetch distance can be easily computed as:

$$distance = \lceil average_miss_latency / cycles_per_iteration \rceil$$

For small strides in integer programs, prefetch distances are aligned to *L1D* cache line size (not for *FP* codes, because *fp* loads bypass *L1* cache on the Itanium 2 processor).

3.2.6 Prefetch Code Optimization and Scheduling

Prefetch code often exhibits redundancy, and hence should be optimized as well. Such optimizations as complexity reduction will reduce the number of instructions executed and save execution cycles. Secondly, on Itanium processor, prefetch code should be scheduled at otherwise wasted empty slots so that the introduced cost is kept as small as possible. Ineffective insertion of prefetches may increase the number of bundles and cause significant performance loss. Details of the Itanium micro-architecture can be found in [22] and [13].

3.3 Trace Layout

Code layout or code positioning is a commonly used technique by both static and dynamic compilation to move related code fragments closer and improve locality. For the same concern, trace layout is implemented in our system to sustain the I-Cache performance. This is particularly useful for programs having huge code footprints during execution because ADORE may generate a large number of traces for them. The algorithm for trace-layout is similar to the one for procedure ordering proposed in [31] by Kark Pettis and Robert Hansen: (1) Build an undirected graph, in which every node represents one trace and every edge represents a branch between two traces. (2) Attach weight (branch frequency) to each edge. (3) Find the edge with the largest weight. (4) Merge its head node and tail node into one node. All the edges starting from and stopping at them are also merged. (5) Continue step 3, 4 until there is only one node in the graph. Finally, the grouping sequence of the traces when they are deployed into the trace cache should be the same as that of the nodes in the merged groups.

3.4 Implementation Issues

Among the critical issues to be solved in real-world dynamic optimization systems, architectural state preservation and precise exception handling [32] are the hardest to address.

Fortunately, the two existing optimizations, trace layout and prefetching, are considered architecturally safe. Prefetch instructions use free 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 instructions. For future optimization such as Code Scheduling or Partial Dead Code Elimination, precise exception handling becomes an issue because the original instruction sequences might be changed. Many aggressive optimizations have this same property, hence need to be dealt with in a unified way in our future work.

3.5 Trace Patching

Trace patching (deployment) involves writing optimized traces into the trace cache. At this stage, the trace-patcher prepares an unused memory area in the trace cache for each trace. The starting addresses of traces are aligned to the page size. In the next step, traces are encoded from *IR*s to binary instructions placed in the trace cache. Finally, to let the optimizations take effect, the execution must be redirected from the original code to the trace cache. Trace cache size is not a big concern in ADORE as the HPM model of optimization only selects a small set of hot traces (far less than 10% of code size).

3.5.1 First Bundle Patching

Intuitively, we intend to redirect execution from the original code to traces in our trace cache by changing the target addresses of the frequently executed branch instructions. However, this requires that all branches to a hot trace be remembered beforehand, because if we plan to relinquish the trace at a later time (e.g. flushing the trace cache), all patched branches must be restored. To keep the system efficient, the current version of ADORE does patching by replacing only the first bundle of the original code of each trace. The first bundle is replaced by a new bundle, which has only one branch instruction jumping to the trace cache. The replaced bundle is not thrown away. It is saved and ready for possible reinstatement of the original code.

3.5.2 Memory Protection

The memory protection on the original code pages is read-only by default. When we wish to modify instructions in the address space of existing code, we make a system-call to allow writing to memory pages of the original code. After patching is done, the write-protection to the original code is restored to protect the original code from accidental changes.

3.5.3 Atomic Writing

Another issue with replacing instruction bundles is that bundles are 128-bits long while the processor only supports 64-bit atomic write instructions. This means that we need to take steps to prevent partially modified bundles from being executed. To deal with this case, we could first patch the first half of the bundle with an illegal bundle type and handle the exception if the bundle is executed before we finish patching it. We then complete the process by modifying the remaining portion of the bundle.

There is an alternative way to deal with atomic patching. Since many programs are only single threaded, we can register a signal handler for their main program when the ADORE system starts up. At the patching stage, ADORE sends a *PAUSE* signal to the main thread and its handler for this signal forces the main program to sleep. During its sleeping, patching can be done as simply as writing regular data to the memory. This approach works well for all CPU2000 benchmark programs.

4. Performance Evaluation

4.1 Methodology

To evaluate the performance of runtime optimization, nine SPECFP2000 benchmarks and eight SPECINT2000 benchmarks [33] were tested with reference inputs. The command line for running each benchmark's input is listed in Table 1. Our 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 [25] was chosen to compile the benchmark programs. The sampling interval used here is 400, 000 cycles/sample. We only discuss the performance results of runtime data cache prefetching since it is currently our major optimization technique. The performance results shown in the following graphs also include code straightening (i.e. trace layout). However, since the hot traces that we select for runtime optimization are usually less than 2% of the original binary size, the benefit from code straightening is not obvious (around 1% on average). For some programs like vortex, selecting more traces can improve the effectiveness of code straightening (3% speedup), but temporarily it is not an encouraging optimization compared with runtime data cache prefetching.

4.2 Runtime Prefetching

In our test, runtime prefetching is applied to the benchmark programs from the 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. For *O2* binaries, we will show the performance data with two different register allocation strategies: (1). The *ORC* compiler reserves 5 global integer registers (*r27*-*r31*). (2). The *ORC* compiler doesn't reserve any integer registers but ADORE uses *alloc* to dynamically allocate up to 16 registers. For *O3* binaries, we only show the performance when the 5 registers are reserved. In all cases, the *ORC* compiler disables the software pipelining loops, even though our current system can generate traces for software-pipelined loops without difficulty. The reason that we disable software-pipelining loops is because our dynamic optimization currently could not handle rotation registers. The impact of having this limitation to the performance was evaluated in our previous work [5].

All benchmark programs run reference data inputs. Figure 11 and Figure 12 illustrate the performance impact of *O2 + Runtime Prefetching* and *O3 + Runtime Prefetching*. In Figure 11, around half of the SPEC2000 benchmarks have speedup from 3% to 57%. For the remaining programs that show no benefit from dynamic optimization, the performance differences are around -2% to +1%. The performance gain achieved by using *alloc* is similar. The difference is that some programs get more speedup while others get lower speed up than

Benchmark	**Command Line Input**
ammp	*< ammp.in*
applu	*< applu.in*
art	*-scanfile c756hel.in -trainfile1 a10.img* *-trainfile2 hc.img -stride 2 -startx 470* *-starty 140 -endx 520 -endy 180 -objects 10*
bzip2	*input.program 58*
equake	*< inp.in*
facerec	*< ref.in*
fma3d	*< fma3d.in*
gap	*-l ./ -q -m 192M < ref.in*
gcc	*200.i -o 200.s*
gzip	*input.source 60*
lucas	*< lucas2.in*
mcf	*inp.in*
mesa	*-frames 1000 -meshfile mesa.in* *-ppmfile mesa.ppm*
parser	*2.1.dict -batch < ref.in*
sixtrack	*inp.in*
swim	*< swim.in*
vortex	*lendian1.raw*
vpr	*net.in arch.in place.in route.out -nodisp* *-route_only -route_chan_width 15* *-pres_fac_mult 2 -acc_fac 1* *-first_iter_pres_fac 4* *-initial_pres_fac 8*

Table 1: Command line inputs used for running benchmarks.

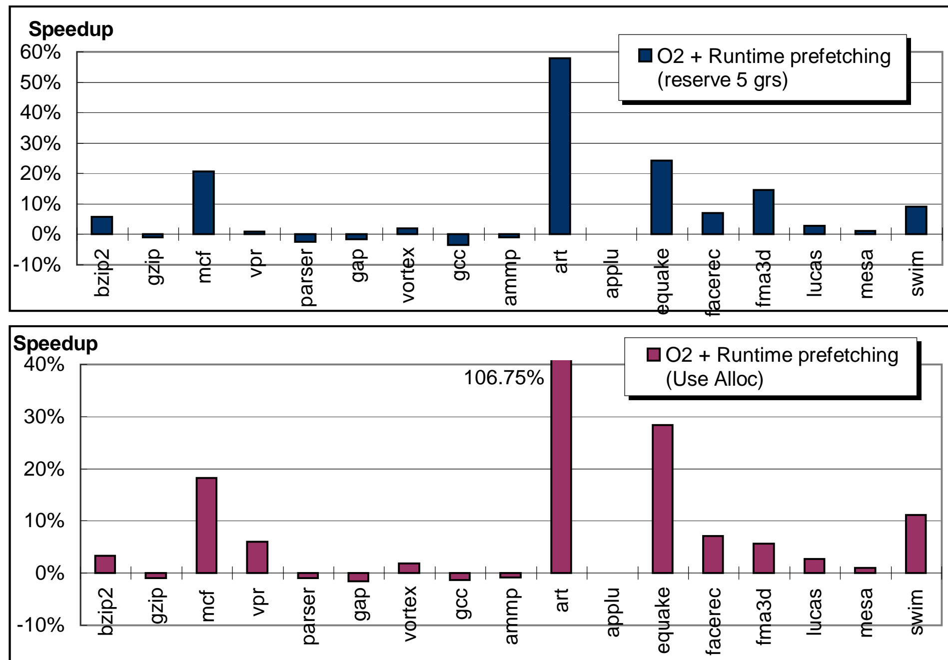


Figure 11: Performance of Runtime Prefetching on O2 Binary. The first method (upper graph) assumes compilers reserve 5 general registers for dynamic optimization to use, sacrificing part of the transparency. In the second method (lower graph), the dynamic optimizer creates new free registers by using the *alloc* instruction in the IA64 ISA. For some programs, having more registers does improve the performance of runtime cache prefetching.

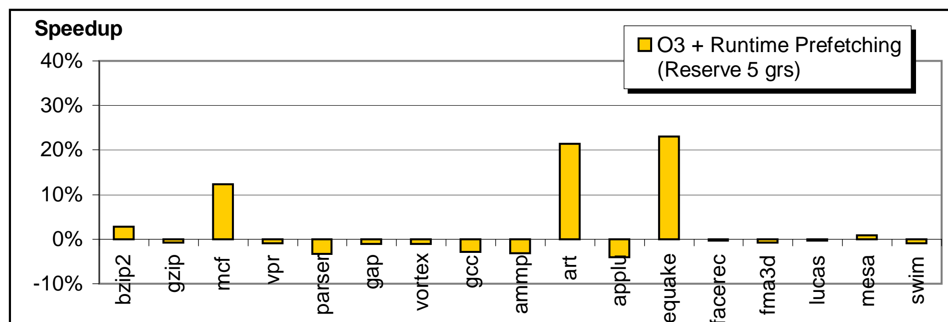
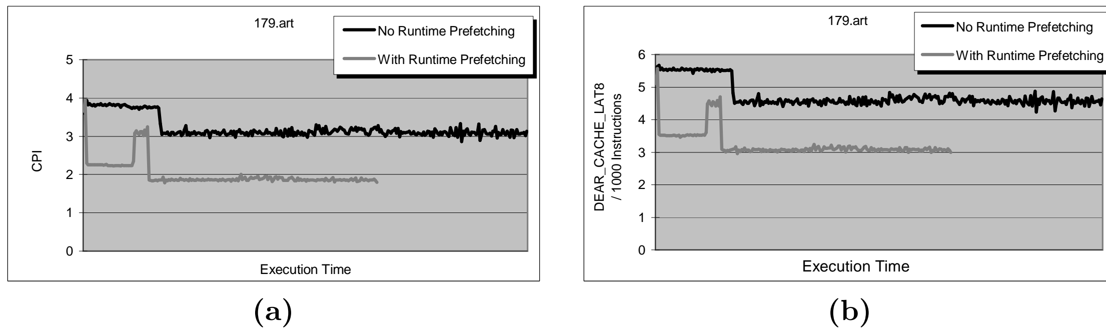


Figure 12: Performance of Runtime Prefetching on O3 Binary. Less performance gain can be acquired since the static compiler already generates prefetches for data intensive loops. Here, only the registers reservation model has been evaluated.



(a) **(b)**

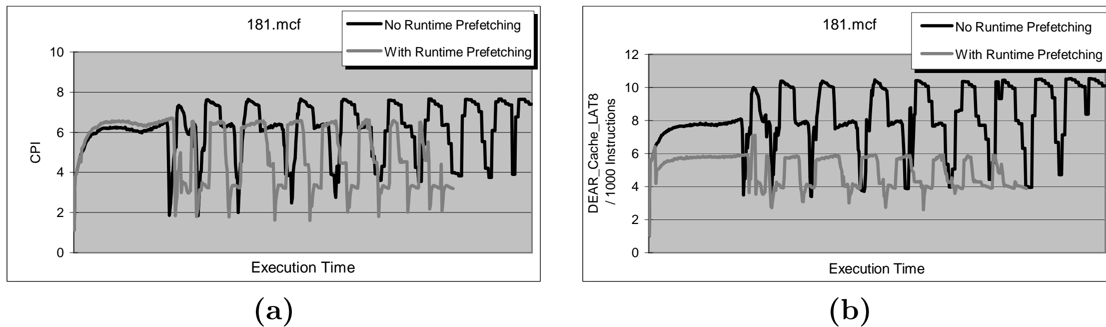
Figure 13: Runtime prefetching for *179.art*

in the case where free registers are reserved.

A further examination of the traces generated by the dynamic optimizer shows that our runtime pre-fetcher did locate the right delinquent loads in *applu*, *parser* 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. 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 *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 and ends up with a 3.8% performance loss. This may be improved by further tuning on trace selection or I-cache prefetching. Furthermore, the number of free registers available for prefetching is critical for some program. For instance, we see the difference in *vpr* from no speedup to 6% speedup when more free registers are used. *applu* and *gcc* get little more improvement if more registers are available. *Art* doubles its speedup from 57% to 106%. However, *fma3d* has less speedup when using *alloc* since some of its functions use all 128 *gr*s up and leave none to *alloc*.

As expected, runtime prefetching shows different results when applied to the *O3* binaries (Figure 12). 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 pre-fetcher 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 Figure 13, the left graph shows the runtime *CPI* change for *179.art* with/without runtime prefetching (*O2* binary). The right graph shows the change of *Load Miss Per 1000 instructions*. There are two clear phases shown in both graphs. One is from



(a) **(b)**

Figure 14: Runtime prefetching for *181.mcf*

the beginning of the execution; the other starts at about one fourth of way in the execution. Phase detection works effectively in this case. The first phase is detected after a few seconds after startup and prefetching codes are applied immediately. Both *CPI* and *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. Figure 14 is the same type of graph for *181.mcf*. Other programs like *bzip2*, *facerec*, *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. The majority of speedup comes from prefetching for *direct/indirect* array references. Prefetching for *pointer chasing* references is not widely applicable because not many Linked Data Structure (LDS) intensive applications exhibit regular stride.

4.3 Two Thread Model

As mentioned, the dynamic optimization is carried out in the second thread created at the main thread's startup. Due to the HPM profiling and the periodically hibernation strategy, the second thread is idle during most of the execution time. This fact is true on both dual-processor machines and single-processor machines. However, this property can be weakened in the future. Our current system is a lightweight system, i.e, the hot trace optimizations (cache prefetching and trace layout) are of $O(n)$ complexity. Nevertheless, some of our research has already revealed that if more complex optimizations are added, the overhead of the second thread will increase quickly and can only be amortized by using the dual-processor model.

4.4 System Overhead

At the end of this section, we evaluate the runtime overhead incurred by our dynamic optimization system. The major causes of overhead are continuous sampling, phase detection and trace optimization. Figure 15 shows the benchmark program's "real time" when prefetch insertion is disabled in ADORE (i.e. there will be no speedup). It is measured using the shell command *time*. The "user time" is not shown here since they are always smaller than "real time" in our experiment. These results demonstrate that the extra over-

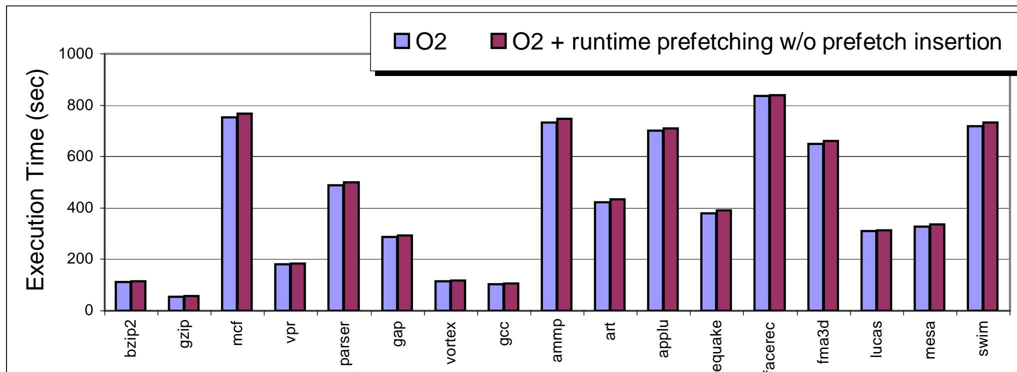


Figure 15: Overhead of Runtime Prefetching

head of ADORE system is trivial. The data was collected on the dual-processor machine. We also evaluated the single-processor machine and got very close results.

5. Related Work

Runtime Optimization Systems are commonly seen in the Java Virtual Machines (JVM) [34],[35], [36], where Just-In-Time (JIT) engines apply recompilation at runtime to achieve higher performance. On these systems, JIT usually employs adaptive profile-feedback optimization (e.g. by instrumentation or interpretation) to take advantages of Java programs' dynamic nature.

Dynamic optimization has been presented in the past in frameworks such as Dynamo [1] and Continuous Profiling and Optimization (CPO) [2]. Dynamo 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. Since interpretation is expensive, Dynamo tries to avoid it by translating as many hot traces as possible to the fragment cache. To achieve this goal, it uses a small threshold to quickly determine whether a trace is hot. This approach often ends up with generating too much code and less effective traces. In the recent work Dynamo RIO [6], this feature has been changed. As a dynamic optimization system but with entirely different design model and implementation, ADORE has many aspects that distinguish it from the above two systems. In Table 2, some major differences between Dynamo and ADORE have been listed.

CPO [2] presents a model closer to traditional PBO where the original code is instrumented, and the profile information is used to compile optimized versions of code. In CPO, profiled information is used to drive PBO while the program is running and the compiled result is hot-swapped into the program. The advantage of this scheme is that the IR information makes application of many optimizations easier.

There are a lot of additional research topics on dynamic optimization. Many of them are often seen in dynamic translation [8],[9],[12] and binary transformation [15].

Tasks	Dynamo/RIO	ADORE
Observation(Profiling)	Interpretation /Instrumentation based	HPM and branch trace sampling based
Major Optimization	Trace layout and classic optimization	D-cache related optimizations
Code cache management	Need large code cache	Small cache (only for hot traces)
Execution Redirection	Interpretation and dynamic linking	Patching only one bundle per trace

Table 2: Dynamo vs. ADORE

6. Conclusion and Future Work

In this paper we present a lightweight runtime optimizer implemented on a real system with existing hardware and software support. We discuss much detail of the design and implementation including profiling, hot trace selection, phase detection, optimization and code patching. The overhead of this system is very low due to the use of hardware performance monitoring and sampling on the Itanium 2 processor. Using ADORE, we can improve the runtime performance of several SPEC2000 benchmarks compiled at *O2* and *O3* for Itanium-2 processors, especially those suffering from data cache misses.

Future directions of our work seek to improve the detection, optimization, and deployment of our optimizations. Detection work includes monitoring current optimizations and tracking the performance of generated traces, finding ideal metrics for detecting stable phases, evaluating burst sampling techniques, and adapting the number of samples to the size of the executable footprint in memory to minimize error. For optimization, we will look at additional optimizations and applying optimizations in different micro-architectures and across shared-libraries. For deployment, we will focus on efficient management of the trace cache.

Acknowledgements

The authors wish to thank Dong-Yuan Chen, Rao Fu, Sagar Dalvi, Sourabh Joshi, Abhinav Das, Jinpyo Kim and Bobbie Othmer for their assistance and advice. We also thank all the anonymous reviewers for their comments.

References

[1] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo: A Transparent Dynamic Optimization System,” in *PLDI’00*, pp. 1–12, ACM Press, 2000.

[2] T. Kistler and M. Franz, “Continuous Program Optimization: Design and Evaluation,” *IEEE Trans. Comput.*, vol. 50, no. 6, pp. 549–566, 2001.

[3] T. Kistler and M. Franz, “Continuous Program Optimization: A Case Study,” *ACM Trans. Program. Lang. Syst.*, vol. 25, no. 4, pp. 500–548, 2003.

[4] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W.-M. W. Hwu, "A Hardware Mechanism for Dynamic Extraction and Relayout of Program Hot Spots," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pp. 59–70, ACM Press, 2000.

[5] J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen, "The Performance of Runtime Data Cache Prefetching in a Dynamic Optimization System," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 180, IEEE Computer Society, 2003.

[6] D. Bruening, T. Garnett, and S. Amarasinghe, "An Infrastructure for Adaptive Dynamic Optimization," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 265–275, IEEE Computer Society, 2003.

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

[8] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32: A Profile-Directed Binary Translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, 1998.

[9] R. S. Cohn, D. W. Goodwin, and P. G. Lowney, "Optimizing Alpha Executables on Windows NT with Spike," *Digital Tech. J.*, vol. 9, no. 4, pp. 3–20, 1998.

[10] Kemal Ebcioğlu and Erik R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 26–37, ACM Press, 1997.

[11] A. Srivastava, A. Edwards, and H. Vo, "Vulcan: Binary Transformation in a Distributed Environment," Tech. Rep. MSR-TR-2001-50, April 2001.

[12] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 Execution Layer: A Two-phase Dynamic Translator Designed to Support IA-32 Applications on Itanium ^{TM}based Systems," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 191, IEEE Computer Society, 2003.

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

[14] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An Effective Technique for VLIW and Superscalar Compilation," *J. Supercomput.*, vol. 7, no. 1-2, pp. 229–248, 1993.

[15] S. J. Patel and S. S. Lumetta, "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Trans. Comput.*, vol. 50, no. 6, pp. 590–608, 2001.

[16] T. Ball and J. R. Larus, "Efficient Path Profiling," in *Micro-29*, pp. 46–57, IEEE Computer Society Press, 1996.

[17] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing ^{TM}Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-life Challenges," in *Proceedings of the International Symposium on Code Generation and Optimization*, pp. 15–24, IEEE Computer Society, 2003.

[18] T. Sherwood, S. Sair, and B. Calder, "Phase Tracking and Prediction," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pp. 336–349, ACM Press, 2003.

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

[20] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory Hierarchy Reconfiguration for Energy and Performance in General-Purpose Processor Architectures," in *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 245–257, ACM Press, 2000.

[21] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 217, IEEE Computer Society, 2003.

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

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

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

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

[26] C.-K. Luk and T. C. Mowry, "Compiler-Based Prefetching For Recursive Data Structures," in *ASPLOS-7*, pp. 222–233, ACM Press, 1996.

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

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

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

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

[31] K. Pettis and R. C. Hansen, "Profile Guided Code Positioning," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pp. 16–27, ACM Press, 1990.

[32] M. Gschwind and E. Altman, "Optimization and Precise Exceptions in Dynamic Compilation," *ACM SIGARCH Computer Architecture News*, vol. 29, no. 1, pp. 66–74, 2001.

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

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

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

[36] M. Paleczny, C. Vick, and C. Click, "The Java^{TM}HotSpot Server Compiler," in *Java^{TM} VM'02*, 2001.