Performance Driven Data Cache Prefetching in a Dynamic Software Optimization System

Jean Christophe Beyler
ICPS/LSIIT, Universite Louis Pasteur, Strasbourg
Pole API, Bd Sbastien Brant
67400 Illkirch - France
beyler@icps.u-strasbg.fr

Philippe Clauss
ICPS/LSIIT, Universite Louis Pasteur, Strasbourg
Pole API, Bd Sbastien Brant
67400 Illkirch - France
clauss@icps.u-strasbg.fr

ABSTRACT

Software or hardware data cache prefetching is an efficient way to hide cache miss latency. However effectiveness of the issued prefetches have to be monitored in order to maximize their positive impact while minimizing their negative impact on performance. In previous proposed dynamic frameworks, the monitoring scheme is either achieved using processor performance counters or using specific hardware. In this work, we propose a prefetching strategy which does not use any specific hardware component or processor performance counter. Our dynamic framework wants to be portable on any modern processor architecture providing at least a prefetch instruction. Opportunity and effectiveness of prefetching loads is simply guided by the time spent to effectively obtain the data. Every load of a program is monitored periodically and can be either associated to a dynamically inserted prefetch instruction or not. It can be associated to a prefetch instruction at some disjoint periods of the whole program run as soon as it is efficient. Our framework has been implemented for Itanium-2 machines. It involves several dynamic instrumentations of the binary code whose overhead is limited to only 4% on average. On a large set of benchmarks, our system is able to speed up some programs by 2%-143%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Run-time environments, Optimization*

General Terms

Performance

Keywords

Dynamic optimization, binary instrumentation, data cache prefetching

1. INTRODUCTION

ICS'07 June 18-20, Seattle, WA, USA.

Many works have shown that software or hardware data cache prefetching is an efficient way to hide cache miss latency [5, 10, 11, 12, 15, 22]. However prefetching can degrade performance if the predicted memory addresses are not accurate or if the prefetch distance (the time between when a prefetch is issued and the data is used) is not appropriate. Although a prediction might be correct, a too large prefetch distance can result in a prefetched line being replaced before being referenced, while a too short prefetch distance can leave an incomplete prefetch at the time the processor requests the data.

Hence effectiveness of the issued prefetches have to be monitored in order to maximize their positive impact while minimizing their negative impact on performance. In previous proposed dynamic frameworks, a monitoring scheme is either achieved using processor performance counters or using specific hardware. In [11, 12], Adore uses a sampling based phase analysis to detect performance bottlenecks by collecting performance counter values on the Itanium processor. In [15], a hardware mechanism incorporating dynamic feedback into the design of the prefetcher is proposed.

In this work, we propose a prefetching strategy which does not make use of any specific hardware or processor performance counter. Our dynamic framework is totally software and wants to be portable on any modern processor architecture providing at least a prefetch instruction. Opportunity and effectiveness of prefetching loads is simply guided by the time spent to effectively obtain a data. Every load of a program is monitored periodically and can be either associated to a dynamically inserted prefetch instruction or not. A prefetch instruction is associated to each load whose average latency is over a fixed threshold. Effectiveness of each prefetch instruction is monitored by comparing latencies between both configurations of issuing prefetches or not. Moreover, the prefetch distance is dynamically adjusted to reach the best performance. In order to minimize the monitoring process overhead, periods of monitoring are not fixed and can vary depending on the behavior of each load. Such mechanism allows to catch the different phases in the whole program behavior. Hence a load can be associated to a prefetch instruction at some disjoint periods of the whole program run as soon as it is efficient. Each monitored load of a program is handled separately by our system. It does not use any trace detection scheme or any distinction between loads that are all handled in a unified way.

Our framework has been implemented for Itanium-2 machines. It involves several dynamic instrumentations of the

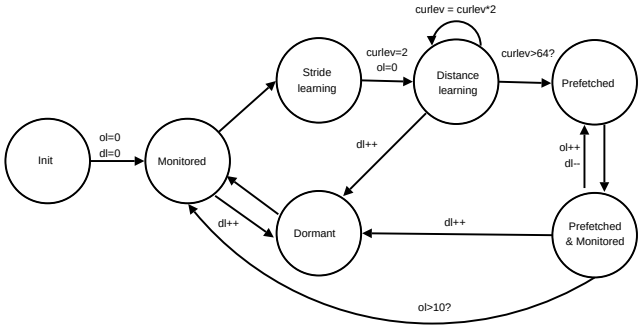

Figure 1: The finite state machine representing the states and transitions of the load instructions

Benchmark	Latency changes	Loads changing latencies	Executed loads
art	90.33	54.67	424
equake	71.33	43.33	728
mcf	307.67	101.33	336
ammp	550.33	172	1113
bzip2	88.33	32.33	808
parser	2674.67	930.67	3571
gzip	43	26	649
vpr	254.67	96.33	3399
swim	52	34.33	272
mesa	180	53	1403
ft	28.3	22.3	108
bc	43.3	26	836
treeadd	0	0	21

Table 1: Load latency variations

binary code whose overhead is limited to only 4% on average. On a large set of benchmarks compiled with either `gcc -O3` or `icc -O3`, our system is able to speed up some programs by 2%-143%.

The paper is organized as follows. The next main section is dedicated to describing the system. An overview of the implemented algorithm is given in the first subsection, where it is explained how the loads of a program are associated to states of a finite automata. The next subsection presents some technical aspects of the system, as well as the general organization of the different software modules. Then the initial tasks of the system are described, and particularly how all the loads are instrumented by the monitoring code. The dynamically inserted code needs a few registers for its own operations, and dynamic register allocation has to take care not to alter the main program execution. Hence it is described in a dedicated subsection how we handle dynamic register allocation. In this main section, it is finally explained how dynamic modification of the code is handled by a cooperative processing of a monitoring thread and a signal handler. Section 3 shows the impact of our system on several benchmark programs. Overhead and performance are evaluated for codes compiled with both compilers `gcc` and `icc`. The impact of evaluating several prefetch distances is also evaluated. Finally, a statistical study of the state evolution of the loads is made for one of the benchmarks. Section 4 highlights the related works and section 5 contains the conclusion and future work.

2. THE RUNTIME ANALYSIS & OPTIMIZATION SYSTEM

2.1 Overview of the System

While the system is running concurrently with the target program, each load instruction is considered separately and associated to a state according to the finite state machine represented on figure 1.

After initialization, all the loads are monitored: each load latency is collected by some specifically inserted instructions surrounding each load. This monitoring lasts for each load until the load has been executed a significant number of times which is predefined in the system.

Then it enters one of two possible states. If its average latency is greater than the cache latency, the load is selected to be associated with a prefetch instruction. First it enters the state where the memory stride is learnt in order to adjust the prefetch distance that will be used in the next state. Otherwise, the load has already an acceptable average latency and enters the "dormant" state: the previously added instructions are removed and the original code is restored.

Variable `dl` is a counter used to adjust the time period while a given load stays in the dormant state. At first, it does not stay a long time in this state and reenters quickly into the monitored state, in order to check whether it can now become a candidate for prefetching. If it is still not the case, variable `dl` is incremented so that the load stays longer in the dormant state and so on. Hence a load which is constantly uninteresting for the system will finally stay for a long time in the dormant state. This has the positive effect of reducing significantly the monitoring overhead since those loads are considered less often.

A load stays in the "stride learning" state until it has been executed a significant number of times. In that state, the last occuring stride is collected. Then it enters the "distance learning" state where a prefetch instruction is inserted before the load instruction. The best prefetch distance is evaluated. Its value is $curlev \times stride$, where $curlev$ varies from 2^1 to 2^6. Each distance is tested during the next executions of the corresponding load. When all the distances have been evaluated, either none of them results in an effective prefetch and the load enters the dormant state, or the best distance is used to prefetch the load that enters now the "prefetched" state, where the prefetch is no longer monitored.

Variable `ol` is a counter used to adjust the time period while a load stays in the prefetched state. At first, it quicly enters the "monitored & prefetched" state where effectiveness of the prefetch is evaluated. Some additional code collects the load latency while it is executed a significant number of times. The average latency is compared to the original latency where no prefetch was issued. If the prefetch is effective, variable `ol` is incremented so that the load reenters the "prefetched" state for a longer period of time. Otherwise, the prefetch is not effective and the load enters the "dormant" state. During this time, the original latency without prefetching could have changed since the last time it was measured and thus could have been better than the latency with prefetching. This can happen at a change of phase in the program execution. For example, a single load instruction used to access several times a data array would at first be considered as delinquent, since the data is not yet stored

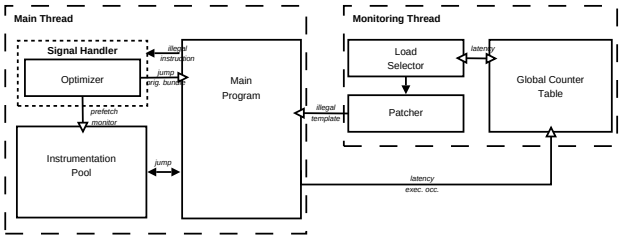

Figure 2: The framework

in the cache. Then if the data remains in the cache, the load would no longer be considered as delinquent. Hence variable `ol` is also used as a confidence level. When it reaches a fixed threshold, the load reenters the "monitored" state, thus gathering the current latency of that load.

Table 1 shows the number of latency changes for each benchmark compiled with the `gcc` compiler at the `O3` level. The results are the average of 3 runs and we defined 4 groups. The different groups are defined by using the number of cycles needed to complete the load: under 8 cycles, between 8 and 15 cycles, between 15 and 40 cycles and over 40 cycles. The second column of the table represents the total number of times a load changed groups. For example, 52 means that, after the initial group assignation of each load, the system noticed that, 52 times, a certain load was no longer in the same latency group. As we can see, the load instructions in the `parser`, `ammp` and `mcf` programs often change groups. The third column shows the number of loads that actually changed from one group to another. For example, `mcf` has in average 101 loads that changed groups for a total of 307 group changes. This means that in average, each of the 101 loads changed 3 times of average latency during the execution of the program. The last column shows the total number of loads that were at least executed once before the last poll of the execution. The average percentage throughout the benchmarks of loads that change at least once of latency groups is 12.67% of the executed loads.

2.2 Technical Issues

Figure 2 illustrates the framework. In the same way as in Adore [11, 12], our system is implemented as a shared library on Linux for IA64 that can be automatically linked to the application at startup. There are also two threads existing at runtime but whose aims are different than in Adore. The main thread runs the target program whose binary code can be modified by the other thread or by a signal handler. This latter thread is in charge of managing the loads by taking decision on their current state, and also is in charge of modifying the binary code of the main thread in order to trigger the signal handler. We also modified the *libc* entry-point routine *__libc_start_main* by including our startup codes.

While a program is running, its binary code, which is mapped into memory, can be modified if the read-only flag has been removed. Thus the code can be parsed and modified. However, since functions are generally mapped sequentially, code insertion is complex. A common way to overcome this issue is to insert an unconditional jump to another section of the memory, the *instrumentation pool*, where new code can be created and modified with another unconditional jump to return to the original code.

All the implementation choices have been motivated by the minimization of the system overhead.

2.3 Initialization and first tasks

First, our system creates a large shared-memory block for the original process. This is the *instrumentation pool* which stores the replacement code for monitoring or optimization. Second, it modifies the first instruction bundle of each function. The template of this first bundle is set to an illegal value such that its execution will result in an illegal instruction signal. To each function is associated a global array element storing its starting address, its length and the original value of the transformed bundle. This element index is stored in the transformed bundle such that the element can be directly accessed by reading the bundle. This is done by reading the binary file's header and parsing the symbol table. However, in the cases of binaries being stripped of the table, or of file formats not containing this information, a full parsing mechanism is used to directly instrument each load. The overhead of both methods is negligible but the first solution reduces the maximum memory usage for programs that do not call every function at each execution. Third, our system installs the signal handler that will be launched at an illegal instruction signal. Finally, it creates a new thread for dynamic monitoring which has the same lifetime as the main thread and which is set at first in sleep mode.

Once every function has been instrumented, the main program starts its execution. As soon as it encounters an illegal template in the first bundle of a function, an illegal instruction signal triggers the handler that parses all the load instructions occurring in the function code. Each bundle containing at least one load is replaced by a new bundle that has only a jump instruction to a different location in the instrumentation pool. For each load, an element is appended to a chained list `new_instr` storing some information as the replaced bundle, the original bundle address, the starting and ending addresses in the instrumentation pool, and both counters `dl` and `ol`. A new code segment is created in the instrumentation pool for each load. It consists in the original load instruction surrounded by a prologue and an epilogue monitoring code, and also by the original instructions that occurred in the replaced bundle. It is ended by a jump instruction to the bundle following the replaced one. An Itanium bundle contains three instructions. Depending on the content of the replaced bundle, some specific dissociation operations for the enclosed instructions have to be achieved to create the new code in the instrumentation pool.

The prologue consists in incrementing the variable counting the execution occurrences of the load instruction, and also in storing the time counter register value. The epilogue consists in executing one instruction, a *stop instruction*, guaranteeing that no other instruction is executed before the load has completed, and in accumulating in another variable the cycle count of the load latency. This stop instruction is either a move in a integer register of the load result, if it is an integer load, or a store to memory of the load result, if it is a floating-point load. A store is the best solution since floating-point arithmetic induces precision errors and since there are no temporary floating registers. Both variables, the execution counter and the time counter, are stored in a global table whose index is also stored in the list `new_instr` for the corresponding load entry.

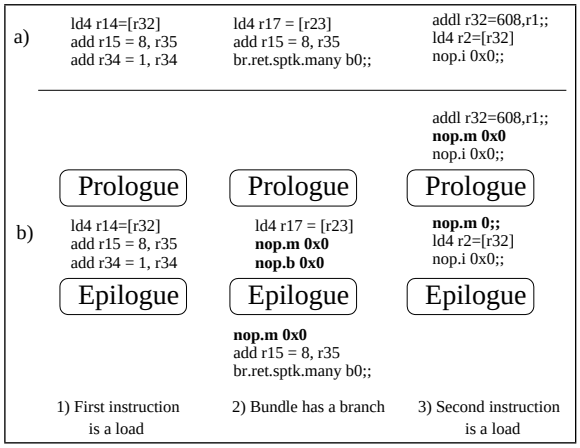

Figure 3: Transformations of certain bundles

2.4 Register usage

As it is shown in [7], dynamic register allocation is a problem that needs to be addressed when implementing a dynamic optimizer. Register allocation on the Itanium processor is achieved using the *alloc* instruction. This instruction defines the number of input, local and output registers that can be used within a function.

When dynamically instrumenting a function, it is often necessary for the optimizer to have access to a few registers for its own calculations. The simplest way to get enough register accesses is to extend the number of output registers. Output registers are used for passing parameters to a function call. Those added registers act as scratch registers since their value is not always preserved across function calls.

Our optimizer code transformations ensure that no value has to be preserved by the system while running the code in the instrumentation pool. Any instruction inducing a jump is moved to after the epilogue so that no branch deviates the execution from the prologue to the epilogue. Figure 3 shows how a bundle with a branch is separated to ensure that no call can modify the dynamic allocated registers.

The main advantage of this solution is that it is fairly easy to implement. While parsing a function, if the optimizer finds an *alloc* instruction, then it modifies it to add the necessary registers. The second advantage is its low overhead cost. However there are two main drawbacks. First if a function is already using all the registers, the optimizer cannot do anything for this function. But this is a very rare case scenario. We observed on the benchmarks considered in the next section that this case occurs only for two programs (`ammp` and `mesa`) and concerns about 13% of the loads in these programs. Second if there are more than one *alloc* instruction in the function, our optimizer does not do anything either. Though with a more elaborate instruction flow analysis, we would probably be able to find which registers to use, we have chosen to simply ignore functions in both cases. This case concerns about 10% of the loads in our benchmarks. However, it can happen that some functions contain no *alloc* instruction. In these cases, our system inserts the necessary *alloc* instructions. That case also occurs for about 10% of the loads in our benchmarks.

Lists	Usage
`new_instr`	monitored loads that have not yet entered another state
`used_instr`	monitored loads
`dormant`	unpatched loads that are in the original program state
`learn_stride`	loads for which the memory access stride is being learnt
`pref_dist_learn`	loads for which prefetch distances are being evaluated
`optimized`	prefetched loads
`opt_instr`	monitored prefetched loads

Table 2: lists associated to load states

Lists	Usage
`todo_instr`	loads that have to be patched to enter either the “stride learning” or “dormant” states
`todo_dorm`	loads that have to be patched to enter the “monitored” state
`todo_ls`	loads that have to be patched to enter the “distance learning” state
`todo_dl`	loads that have to be patched to enter the “prefetched” or “dormant” states, or reenter the “distance learning” state
`todo_opt`	prefetched loads that have to be patched to be monitored
`todo_optim`	prefetched and monitored loads that have to be patched to enter either the “dormant” or “prefetched” states

Table 3: intermediate lists

2.5 The Monitoring Thread and the Signal Handler

The monitoring thread and the signal handler access two kinds of chained lists: lists of loads associated to the load states described in table 2 and intermediate lists of loads whose states are going to change described in table 3. Each entry in the lists is a data structure containing the replaced bundle, the original bundle address, the starting and ending addresses in the instrumentation pool, both counters `dl` and `ol`, and the index of the counters in the global counter table.

After 10 milliseconds, which is the base sleeping period, the second thread wakes up and polls all the lists associated to states. It determines whether some loads have to change their states following the described transition mechanism of subsection 2.1. If no state has to change, its sleeping period is doubled. This period can be doubled until it reaches 128 times the base period. Otherwise if some changes occur, the concerned loads are transferred to the proper intermediate lists, the associated bundles in the main program are transformed as illegal instructions, and the sleep period is reset.

Notice that the monitoring thread transforms concurrently some bundles of the main program. Hence it can be possible for the latter to encounter an illegal instruction, and the signal handler to be triggered, although the monitoring thread has not yet finished to poll all the lists. To avoid such a situation, a simplified mutex is used to prevent con-

current modification of the code by the signal handler and the monitoring thread.

As the main thread encounters an illegal instruction, the signal handler polls all the intermediate lists. For each load appearing in those lists, it transforms properly the main program code and the instrumentation pool, and transfers the load in the adequate state list.

3. EXPERIMENTS AND PERFORMANCE EVALUATION

In the current implementation, the minimum number of times a load has to be executed before changing its state is fixed to one thousand. The average latency over which a load becomes a candidate for prefetching is fixed to 15 processor cycles. Every parameter of the system was found empirically and better combinations can be possible. However we feel our results are satisfying. Though each running platform may have a set of best parameters, finding satisfying ones was not a problem. We do not think it will be on other platforms.

Our system was run with 5 SPECFP2000 benchmarks, 5 SPECINT2000 benchmarks [14] with reference inputs, two Pointer Intensive benchmarks (`ft` and `bc`) [2] and one Olden benchmark (`treeadd`) [1]. Our test machine is a 2-CPU 1.3Ghz Itanium-2 rx2600 workstation. However, notice that our system is entirely run onto one unique processor. The operating system is Linux kernel version 2.6.9-42 with glibc 2.3.4. We used both compilers `gcc 3.4.6` and `intel icc 8.1` with `O3` to compile the benchmark programs. Notice that the `icc` compiler generates prefetch instructions with `O3` while the `gcc` compiler does not.

3.1 System Overhead

We measured our system overhead by inhibiting the insertion of prefetch instructions. However it is impossible to measure exactly the induced overhead of the system whose behavior is necessarily different when prefetch is allowed. Thus our measures give a coarse approximation (see figure 4). This is also true for any statistical information that would require some specific code instrumentation. The major causes of overhead are continuous monitoring and code modification. In particular, jumps to the instrumentation pool yields some instruction cache misses. Anyway the measures show that the extra overhead of our system is negligible.

3.2 Performance Analysis

Figure 5 illustrates the performance impact of our system for the benchmarks either compiled with `gcc` or `icc`. About half of the benchmarks have a speedup from 2% to 132%. For the programs showing no benefit, the performance differences are about -1% to -9%. Time-consuming programs get generally a quite larger benefit of our system. For example, the execution time of program `equake` is about 300 seconds while the execution time of program `bc` is about 22 seconds. Hence the execution time of `bc` is not enough for our system to counterbalance its basic overhead. Anyway, the additional execution time for programs that are slowed down is always very low.

Even though the source code of the different benchmark programs is the same, the binary codes generated by both compilers are different to the extent that our optimizer behaves differently and adapts itself to the target code. For

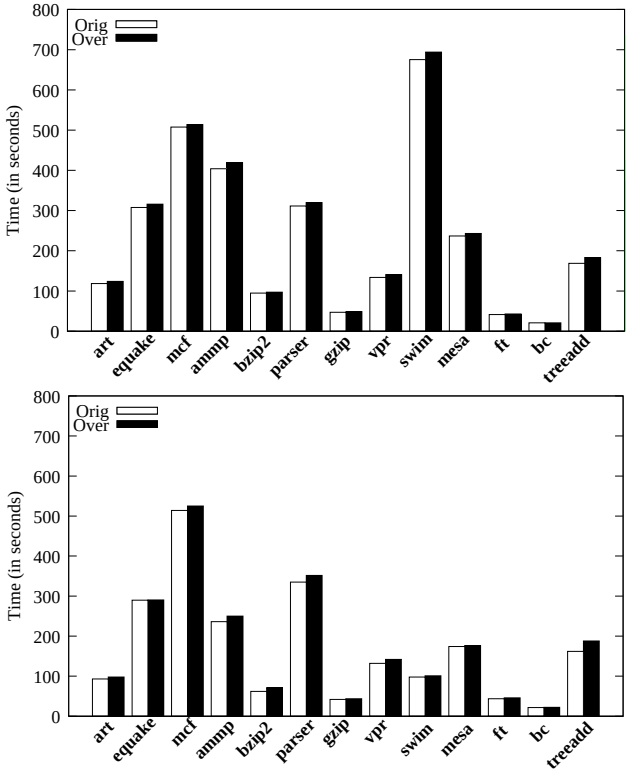

Figure 4: System overhead with `gcc` and `icc` compilers

example, notice that very different results were obtained for `art` and `ft` between the codes compiled with `gcc` and `icc`. For the `art` binary code generated by `icc`, a quite better speed-up was obtained by our system. This fact is reversed with the `ft` binary code. Hence even if `icc` generates some prefetch instructions, either these are not effective or additional prefetches are still improving performance.

For `ammp` and `mesa`, the results can be explained by the fact that some functions could not be transformed by our system due to register allocation issues as explained in subsection 2.4.

For program `swim`, amazing different execution times were obtained beetween codes compiled with `f77 -O3` and `ifort -O3`. With `f77`, the execution time was about 10 minutes while with `ifort`, it was about 90 seconds. Significant speed-up was obtained with the `f77` compiled code as expected for such an execution time.

3.3 Prefetch distance

Figure 6 illustrates the effectiveness of the prefetch distance learning process by comparing the performance due to our system when the distance is fixed to 2, 12 and 64 and when the prefetch learning distance process is switched on. It can be observed that the performance is obviously very sensitive to the distance. The performance reached with the learning process is always very close to the best performance with a fixed distance, and is even the best in some cases where the best distance is not constant during the whole program run. It is necessarily a bit lower due to the learning process overhead when the best distance is constant. For program `equake`, the larger difference can be explained by

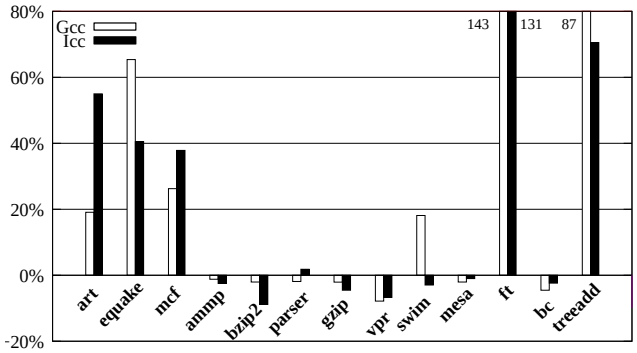

Figure 5: Performance Speed-Up of Dynamic Prefetching with `gcc` and `icc` compilers

Benchmark	Distance changes	Loads changing distances	Optimized loads
art	40.67	16.33	26.33
equake	105.67	27	46.67
mcf	11.6	6.67	27.3
ammp	20.67	8.67	15
bzip2	5	2.67	9.67
parser	76	39.33	156.67
gzip	0	0	0
vpr	29.33	9.33	32.67
swim	83.33	24.33	34
mesa	0	0	3.67
ft	1.67	1	4
bc	0	0	1
treeadd	16.33	1	1

Table 4: Prefetch distance variations

the fact that value 12 is not handled by the learning process, since it handles only values that are powers of two, from 2^1 to 2^6. Anyway, this could be easily modified in the system, and the obtained speed-up is already quite significant. For program `treeadd` compiled with `icc`, the best speed-up is obtained with the learning process. By tracking our system transformations, we observed that different prefetch distances are used for the same load at different phases.

Table 4 shows the average number of changes in the prefetch distance during the executions of the different programs. A change in the prefetch distance is defined when, after an initial distance is calculated, a better distance is found. For example, `treeadd` only has one load that is optimized by the system but the prefetch distance changes in average 16 times during the execution of the program. The prefetch distance can change on two occasions. First, when the load goes from the dormant state to the instrumented state. If it is still delinquent, a stride and prefetch distance calculation will occur. Second, if the load is continuously optimized, the load will be reinstrumented to recalculate the current latency. When that happens, if the load is still delinquent, another stride and prefetch distance calculation will occur. The percentage of loads that have a prefetch distance change out of the loads that have been optimized is 40.1% across the different benchmarks and the average number of changes per load is 3.97 per benchmark for every load that changed distance at least once.

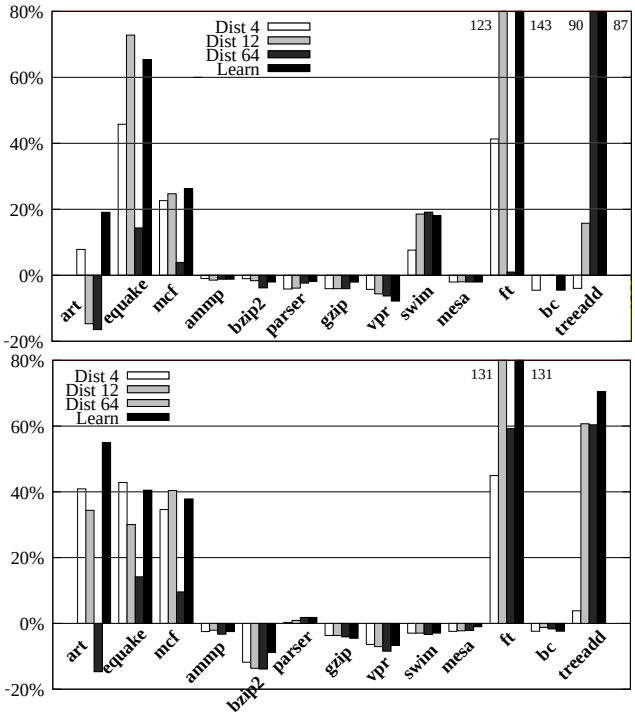

Figure 6: Effectiveness of the prefetch distance learning process with `gcc` and `icc` compilers

3.4 Behavior Analysis

Since our system associates states to all loads of a program, it can be interesting to observe the evolution of the number of loads in each state along the whole execution time. The system can be easily modified to output this information each time the monitoring thread wakes up. Figure 7 shows such evolution for program `equake`. The "monitored" state is split into two states, "new" and "used", where loads in the "new monitored" state are loads that have never been executed more than 1000 times. Those loads are considered ten times less frequently by the monitoring thread. At a given time of the execution, the number of loads in each state is given by the area separating successive curves. For example, near the end of the execution time, the number of loads in the "new monitored" state is about 65% while the number of loads in the "used monitored" state is about $87 - 65 = 22\%$.

The arrows that are represented under the time axis highlight some interesting phase transitions. The first arrow shows the time when program initialization is over and when the program enters the main computation phase. In this new phase, many loads stay in the "used monitored" state since they are no longer executed. The second arrow shows a peak generated by the monitoring thread. Observe that these peaks are increasingly spaced since the dormant level (`dl`) is incremented each time a non-delinquent load is instrumented and, after a thousand executions, returns to the dormant state. The third arrow shows a phase transition where some loads that were in the "new monitored" state were now executed a significant number of times with high latencies and thus enter the "stride learning" state. From this state, these loads enter either the "dormant" state or

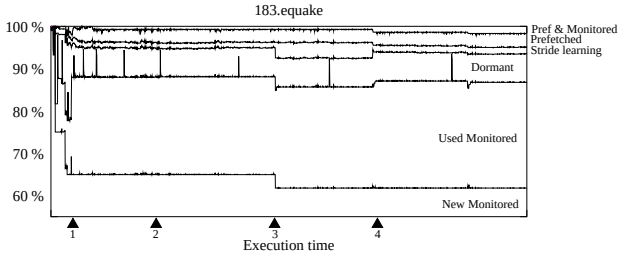

Figure 7: Evolution of the number of loads in each state for program `equake`

the "prefetched" state depending on the effectiveness of the associated prefetches (fourth arrow).

4. RELATED WORK

Specific prefetching algorithms have been evaluated in numerous works proposing either software or hardware implementations. We refer the reader to [9, 18] for a survey of data prefetch algorithms.

Dynamic Optimization Systems are often seen in the Java Virtual Machines (JVM) [17, 6]. Just-In-Time (JIT) compilers apply recompilation at runtime to enable optimizations that were not feasible statically. These systems, using code instrumentation, modify the code depending of the current behavior of the program.

Dynamic Optimization Frameworks such as Dynamo [3] or, more recently, DynamoRIO [4] are transparent systems since the base program does not need to be compiled with specific options. They are dynamic native-to-native optimization systems. Dynamo is provided as a user-mode dynamically linked library. It interprets the base code at runtime searching for hot traces. Using a low threshold to detect hot traces, Dynamo creates an optimized version of the trace and patches the code accordingly. The system is divided into an interpreted code that is not optimized and the hot traces that run natively. To compensate the overhead of the system, Dynamo moves as much code from interpretation to natively run as possible. In a more recent work [8], a framework called Deli is used to build client applications that manipulate or observe running programs. It is shown how a user can define *on the fly* code modifications at low cost.

UMI [21] is a lightweight dynamic system built on DynamoRIO [4] that can be used to profile and optimize programs. DynamoRIO's internal framework discovers hot-traces and sends them to an internal instruction cache. When DynamoRIO selects a trace, UMI creates a clone that can profile the different loads contained in the trace. This instrumentation enables UMI to gather short memory reference profiles to simulate the cache behavior and select the delinquent loads. Using a sampling period and turning the profiling on and off, UMI keeps the overhead at an average of 14%, which is only 1% greater than the base of 13% incurred by DynamoRIO. Finally, UMI integrates a simple stride prefetching solution into the framework to achieve an 11% improvement on the tested benchmarks.

Lu *et al.* [11, 12] implemented a dynamic optimization system called Adore that inserts prefetching instructions into the instruction stream. Using Itanium performance counters, the system handles hot traces and detects delinquent loads in loop nests. Loads are derived into three categories : direct array, indirect array and pointer chasing loads. By studying the code of the hot trace, Adore decides how to prefetch correctly the required data. In [11], the optimizer uses a compiler option to reserve the registers needed for the inserted code. In [12], a modification of the bundles containing the `alloc` instruction enables the inserted code to use new scratch registers. Adore uses the profiling tool `pfmon`. The prefetch distance is calculated using the average miss latency of the load and the number of cycles spent by one iteration.

Post-link optimization such as Ispike [13] instruments and optimizes codes after they have been compiled. It is also an optimization tool on the Itanium Architecture. It uses a polling mechanism and `pfmon` to collect data from the Itanium performance counters, as it is done with Adore. Post-link optimizations must handle similar issues while modifying pre-compiled code. Special register allocation and special care in conserving program semantics are needed.

Chilimbi and Hirzel's framework [5] samples the program's execution to decide what portions of the code should be optimized. Using Vulcan [16], an executable-editing tool for x86, the framework creates two versions of each function. While both versions contain the original code, one of them also contains the instrumentation code and the other only contains some *checks* to decide whether to instrument again the function or not. Using a global hibernation phase, the overhead is lowered enough to achieve a speed-up on several SPECint2000 programs. Once a trace is selected and profiled, an analysis is done to find hot data streams. Finally, when the data streams are detected, a finite state machine is created to prefetch the data. The code handling this state machine is then injected dynamically into the program.

Trident [19] is a event-driven multithreaded dynamic optimization framework. Using hardware support to identify hot traces and helper threads to guide and perform optimizations, it generates low overhead. The software side of the framework consumes the hardware generated events and then decides whether to optimize or not the base program. In [20], Trident is extended to dynamically decide what prefetch distance has to be used. Using a *Delinquent Load Table*, the system monitors loads within a hot trace. It counts the number of executions of each load, the number of misses and measures the miss latency. As with Adore [11], stride and pointer loads are distinguished. By continuously monitoring the load latencies, the prefetch distance is adjusted until the loads are no longer delinquent.

Our system includes many aspects that are inspired by all of the above works. It is totally transparent by using the linux entry point *__libc_start_main*. Like Adore, it inserts prefetch instructions in the instruction flow and, like Trident, it uses a variable prefetch distance. But it also differs by being totally software. The system does not use any of the Itanium performance counters, hence it is portable to any kind of processor. Moreover it does not handle only loads in loops, but almost every load in the program. Hence often executed loads as those occurring in recursive functions are also handled. Moreover they are monitored individually, instead of using a hot trace detection scheme. Lastly, the system does not categorize loads into stride or pointer loads but handles all loads in a unified way. Our system can be compared to Chilimbi and Hirzel's system: we add checks to detect delinquent loads but our system lets the code run

natively compared to leaving lightweight checks in the code. The hibernation system is also almost identical but our system handles each load independently whereas Chilimbi and Hirzel's solution uses a global hibernation system.

5. CONCLUSION

In this paper, we propose a dynamic analysis and optimization system which is totally software and that inserts prefetch instructions where it has been evaluated as effective by measuring the load latency. The software implementation has been achieved by constantly optimizing the code in order to minimize the induced overhead. It has been shown on the presented benchmarks that even if it is totally software, significant speed-up is obtained for several programs. Moreover, our performance results look similar to results obtained in other works entirely or partially based on specific hardware.

However we think that the overhead could still be lowered by avoiding jumps to the instrumentation pool and by inserting monitoring and prefetching code in the place of existing `nop` instructions surrounding loads. We will soon experiment this strategy. We will also implement some more advanced strategy for dynamic register allocation in order to fix the cases presented in subsection 2.4.

Since it is totally software, in the near future our system will be implemented on x86 platforms. We will also experiment the system on smaller processors as the *Freescale Coldfire* microprocessor in order to evaluate the usability of the system on small embedded systems. Although this processor does not provide a prefetch instruction, we will experiment other mechanisms as issuing some well-timed loads.

We also plan to allow our system to be used as a dynamic memory access profiling tool that transparently tracks all loads in a program and generates statistical outputs.

6. REFERENCES

[1] The Olden benchmark suite. http://www.cs.princeton.edu/~mcc/olden.html.

[2] Pointer-intensive benchmark suite. http://www.cs.wisc.edu/~austin/ptr-dist.html.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[4] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *International Symposium on Code Generation and Optimization*, Mar. 2003.

[5] T. M. Chilimbi and M. Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209. ACM Press, 2002.

[6] M. Cierniak, G.-Y. Lueh, and J. M. Stichnoth. Practicing judo: Java under dynamic optimizations. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 13–26, New York, NY, USA, 2000. ACM Press.

[7] A. Das, R. Fu, A. Zhai, and W.-C. Hsu. Issues and support for dynamic register allocation. In *Asia-Pacific Computer Systems Architecture Conference*, pages 351–358, 2006.

[8] G. Desoli, N. Mateev, E. Duesterwald, P. Faraboschi, and J. A. Fisher. Deli: a new run-time control point. In *35th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 257–268, December 2002.

[9] P. G. Emma, A. Hartstein, T. R. Puzak, and V. Srinivasan. Exploring the limits of prefetching. *IBM J. Res. Dev.*, 49(1):127–144, 2005.

[10] A. Ki and A. E. Knowles. Adaptive data prefetching using cache information. In *ICS '97: Proceedings of the 11th international conference on Supercomputing*, pages 204–212. ACM Press, 1997.

[11] J. Lu, H. Chen, R. Fu, W. Hsu, B. Othmer, P. Yew, and D. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *36th Annual IEEE/ACM International Symposium on Microarchitecture*, December 2003.

[12] J. Lu, H. Chen, P.-C. Yew, and W.-C. Hsu. Design and implementation of a lightweight dynamic optimization system. *J. Instruction-Level Parallelism*, 6, 2004.

[13] C.-K. Luk, R. Muth, H. Patil, R. Cohn, and G. Lowney. Ispike: A post-link optimizer for the intel®itanium®architecture. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 15, Washington, DC, USA, 2004. IEEE Computer Society.

[14] SPEC CPU2000. http://www.spec.org/cpu2000/.

[15] S. Srinath, O. M. H. Kim, , and Y. N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *Proc. of the 13th Int. Symp. on High-Performance Computer Architecture (HPCA)*, Feb. 2007.

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

[17] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. Design and evaluation of dynamic optimizations for a java just-in-time compiler. *ACM Trans. Program. Lang. Syst.*, 27(4):732–785, 2005.

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

[19] W. Zhang, B. Calder, and D. M. Tullsen. An event-driven multithreaded dynamic optimization framework. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 87–98, Washington, DC, USA, 2005. IEEE Computer Society.

[20] W. Zhang, B. Calder, and D. M. Tullsen. A self-repairing prefetcher in an event-driven dynamic optimization framework. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 50–64, Washington, DC, USA, 2006. IEEE Computer Society.

[21] Q. Zhao, R. Rabbah, S. Amarasinghe, L. Rudolph, and W.-F. Wong. Ubiquitous memory introspection. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, Washington, DC, USA, March 2007. IEEE Computer Society.

[22] Q. Zhao, R. Rabbah, and W.-F. Wong. Dynamic memory optimization using pool allocation and prefetching. *SIGARCH Comput. Archit. News*, 33(5):27–32, 2005.