HDOS:An Infrastructure for Dynamic Optimization

Tomoaki Ukezono and Kiyofumi Tanaka
Japan Advanced Institute of Science and Technology
School of Information Science
1-1 Asahidai, Nomi, Ishikawa 923-1292 Japan
e-mail: {t-ukezo,kiyofumi}@jaist.ac.jp

***Abstract*—Recently, CPUs with an identical ISA tend to have different architectures, different computation resources, and special instructions. To achieve efficient program execution on such hardware, compilers have machine-dependent code optimization. However, software vendors cannot adopt this optimization for software production, since the software would be widely distributed and therefore it must be executable on any machine with the same ISA. To solve this problem, several dynamic optimization techniques such as JAVA JIT compiler and Microsoft .NET platform are effective. The techniques can perform the machine-dependent code optimization without source code. However, the techniques have to exploit special environment (VM) for program execution or special executable code (e.g. code augmentation). In this paper, we propose an infrastructure for dynamic optimization, *HDOS*. The HDOS is organized with dedicated hardware inside a CPU and operating system support. The HDOS provides more lightweight dynamic optimization than VM-based or interpreter-based dynamic optimization. Furthermore, the HDOS can reuse optimized binary codes at the next execution time since it translates native binary codes into native binary ones.**

Keywords : dynamic optimization, binary translation, trace, prefetching,

I. Introduction

Recently, software development methods are increasingly growing by using dynamic link library, dynamic class loading, and virtual machine techniques. Furthermore, by using those techniques, automatic program update can be limited only to code difference, and only the difference should be distributed across computer networks.

In such software distribution, software vendors do not deliver source codes to clients because of easy installation. In most case, the clients can only receive pre-compiled binary codes of the software.

On the other hand, advances in hardware are notable as typified by evolution of recent CPUs. Even if CPUs follow an identical ISA (Instruction Set Architecture), the CPUs can have different microarchitectures, different computation resources, and special instructions on each implementation. (i.e. modern x86 architecture family maintains upward i386 ISA compatibility to execute older binary codes.) To achieve efficient program execution on such CPUs, compilers provide machine-dependent code optimization. However, software vendors cannot adopt the optimization if the products are distributed, since the software must be executable on any machine with the same ISA.

In order to solve the problem, dynamic optimization techniques are effective. Dynamic optimization techniques can optimize binary codes at run time. In other words, the dynamic optimization is client-side (not vendor-side) optimization. For Example, JAVA JIT compiler translates byte codes to native (optimized) binary codes at class loading time, and reduces overheads due to virtual machine execution. However, the translation cannot be applied to all parts of codes, and therefore the remaining byte code execution is ten times or more inherently-slow compared with native code execution. Moreover, the JIT compilation overhead is incurred at every class loading time even if the same class is loaded again.

In this paper, we propose a new infrastructure for dynamic optimization, *Hybrid Dynamic Optimization System (HDOS)*. The HDOS aims to perform translation from native binary codes to optimized native binary codes. The HDOS evaluates program behavior while the program is running. One feature is that the HDOS can reuse optimized binary codes at the next execution time without optimization overhead. The HDOS is organized with dedicated hardware inside a CPU and operating system support. The dedicated hardware is *User Definable Trap (UDT)*. The UDT allows CPU users (not CPU designers) to define conditions of trap generation. Software called by the trap is optimizer routines which is provided by an operating system. In this paper, we describe those mechanisms with this hardware and software. In addition, we show some performance evaluation of two examples of optimization using the HDOS.

II. The HDOS

The HDOS consists of auxiliary hardware inside a CPU, dedicated to trap functions, and trap handling software installed in an operating system. The optimizer routine is implemented as a trap handler. This section details requirements of the interface between the auxiliary hardware and trap handler. In this paper, we focus on how to implement the HDOS on the Alpha[6] instruction set architecture.

A. UDT

The proposed trap hardware generates trap events as the need arises, which is controlled by the software. The proposed trap hardware provides a mechanism, *User Definable Trap (UDT)*. Figure 1 shows a block diagram of the UDT hardware. In the figure, for example, execution of BNE generates a trap when the branch is taken.

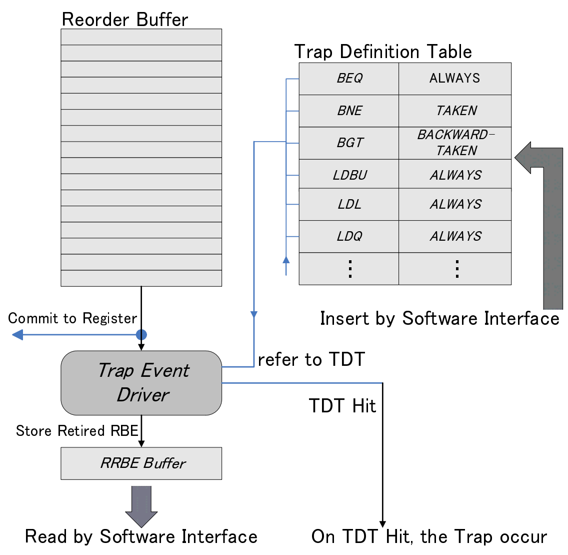

Fig. 1. User Definable Trap (UDT) Hardware.

The UDT generates a trap when the reorder buffer commits a bottom entry to the register file. The *Trap Event Driver (TED)* is a main circuitry. The TED performs two tasks simultaneously. One is to store the bottom entry into the *Retired Reorder Buffer Entry (RRBE)* Buffer. The other is to refer the *Trap Definition Table (TDT)* for the entry. A TDT entry is a set of an operation code and instruction behavior. When an entry in TDT matches the retired entry, the UDT generates a trap. Note that the RRBE and TDT entries can be accessed by a trap handler.

Conventional CPU design allows the trap handler to have a control only when the CPU generates traps determined by the CPU designer, such as exceptions, external interrupts, and system-calls. The UDT can invoke a trap handler, when trap conditions specified by the CPU users (not CPU designer) are satisfied. When the UDT is used as dynamic optimization environment, the trap handler can be implemented as an optimizer routine. If complicated optimizations are completely implemented as hardware, the hardware will become large circuitry and it can easily degrade clock frequency. With this function, it is made possible to shift the majority of binary optimization functionalities for dynamic optimization to the trap handler, and thereby the HDOS can offer optimization environments with high flexibility toward sophisticated optimization, without much hardware overhead.

In this paper, the UDT is used for dynamic optimization. However, the UDT can be used for run-time debugger or performance monitor. For example, if the TDT is set by software debugger properly, the trap handler can have breakpoint functions and watching-variable functions without code augmentation. The UDT can provide functionalities for many applications so that there are other ways to exploit this mechanisms. Therefore, embedding the UDT into a CPU for recent workstation or personal computers can be feasible.

TABLE I
BIT PATTERNS OF IDENTIFYING INSTRUCTIONS.

Instruction-identification bits	type of instructions
00 [6bit-opcode]	Control
01 [6bit-opcode]	Load/Store
10 [6bit-opcode]	Integer Arithmetic & Logical
11 [6bit-opcode]	Miscellaneous

TABLE II
AN EXAMPLE OF BIT PATTERNS FOR BEHAVIOR OF BRANCH INSTRUCTIONS.

bit patterns	condition of behavior
0000 0001	not-taken
0000 0010	taken
0000 0100	forwad
0000 1000	backward
0001 0000	unconditional (Branch Type)
0010 0000	indirect (Branch Type)
0100 0000	return (Branch Type)
1000 0000	reserved

B. Bit Encoding of TDT

A TDT entry consists of two fields, instruction-identification and instruction behavior. Table I and III show the bit patterns of identifying instructions and an example of use of the instruction behavior field for branch instructions, respectively.

Instruction-identification bits are provided as an eight-bit field. Upper two bits represent a type of instruction, and lower six bits represent opcode of Alpha instruction set. Basically, the TDT can represent individual instructions by using the 6-bit opcode field. In addition, in order to reduce the number of TDT entries to be used, it can consolidate the same type of instructions, by using the 2-bit and setting the opcode field to zeros, into an TDT entry. For example, the pattern, “01”+“000000”, means all load and store instructions invoke a trap. Instruction-behavior bits are also provided as an eight-bit field. Individual bit position represents condition of behavior and branch-type. The branch-type bits (unconditional, indirect, and return) are necessary when two or more branch instructions are consolidated into a TDT entry.

The instruction-behavior bits can be set for multiple conditions. Introducing this encoding, calling trap handlers can be finely controlled. In experiments of optimization shown in this paper, the ability of the consolidation and multiple-condition setting enables the optimization to be achieved by using only two TDT-entries. This means that a few TDT-entries are enough to provide practical optimizations, which does not much influence either hardware size/cost or clock frequency.

C. UDT Implementation for Alpha Architecture

In the Alpha architecture, the trap by UDT is treated just as the system service call exception. In the Alpha, the `system service call exception` is generated when the `callsys` instruction is executed. In the UDT mechanism, any instruction can generate the exception if condition specified in the TDT is satisfied. The UDT calls a system service call exception handler using the `entSys` (exception entry-point) register as in the general system service call exception.

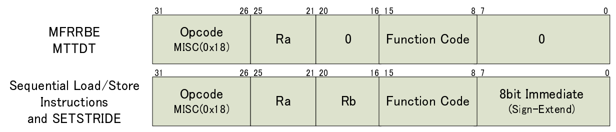
MFRRBE
MTTDT
Sequential Load/Store Instructions and SETSTRIDE
31 26 25 21 20 16 15 8 7 0
Opcode MISC(0x18)
Ra
0
Rb
Function Code
8bit Immediate (Sign-Extend)

Fig. 2. Formats of FRRBE and MTTDT in Alpha Instruction Set.

Simultaneously, the UDT sets the 8th bit (called UDT Field) of `Exception Summary Register` to one. (In most Alpha architecture implementations, this bit position is reserved.) The `system service call exception` handler checks the UDT Field before it starts general system service call handling. If the UDT field is found to be one, it detects the UDT trap request, and jumps to optimizer routine.

Traps by UDT should occur only when user codes are running, since the traps are for optimization of the user codes, and kernel codes including trap handlers are out of the scope. In the UDT implementation on Alpha architecture, if the processor is running in a kernel mode (status register's field, `PS<mode>`, is zero), the UDT does not generate any trap even when a matched TDT entry is found.

D. Software Interface for UDT

New instructions, `MFRRBE` (Move From RRBE) and `MTTDT` (Move To TDT) are added to the Alpha instruction set to access the RRBE buffer and TDT entries. The `MFRRBE` instruction transfers a data from the RRBE buffer to a general purpose register. The `MTTDT` instruction transfers a data from a general purpose register to a TDT entry. The Both instructions are privileged instructions. The format of `MFRRBE` and `MTTDT` are illustrated in figure 2. These instructions belong to Miscellaneous Instructions in the Alpha.

Using `MFRRBE`, the trap handler can analyze behavior of the instruction that generated the trap. Using `MTTDT`, an operating system sets conditions of UDT traps in the TDT.

III. Examples of Optimization

In this section, we introduce two optimization techniques using the HDOS. One is software trace optimization, the other is sequential data prefetching optimization. The following subsections describes them in detail.

A. Software Trace Optimization

One of applications of the HDOS is the software trace optimization. This optimization improves efficiency of instruction fetching in superscalar processors. Most of integer programs have complex control structure. In such program execution, performance is degraded when taken branch causes inefficient instruction fetching. To avoid this inefficient instruction fetching, hardware approaches such as branch prediction and trace cache are widely known. One the other hand, this optimization can avoid this inefficient instruction fetching by software approach. It generates software traces and adds them to the original binary code, where frequent taken branch is translated to not-taken branch.

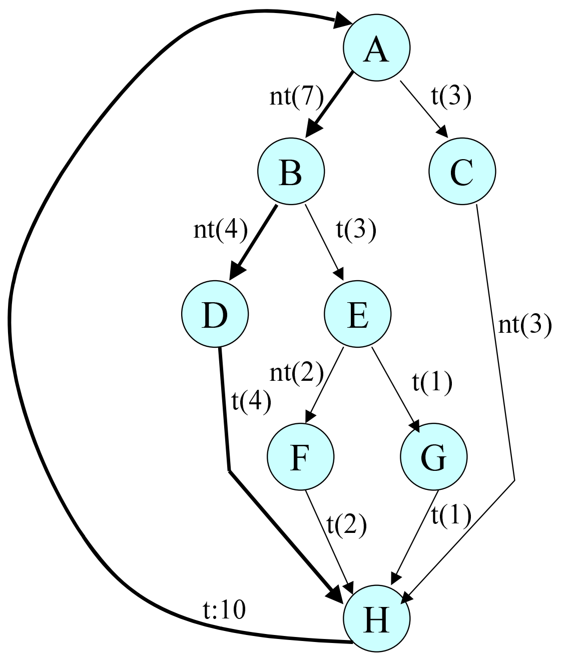

Fig. 3. Weighted digraph.

1) Loop Detection: This optimization generates software traces. However, it is difficult to generate all inherent traces in the program, since the code size would be too large. Therefore, it is important to generate only effective traces that are frequently executed (hot-traces/hot-spots). Loop bodies are one of the examples. At first, the optimization finds loop structure. The HDOS creates TDT entries to generate traps when backward branch is performed. The UDT mechanism calls an optimizer routine every time the CPU executes a backward branch instruction in this TDT configuration. The optimizer routine reads the RRBE, adds the instruction address (PC value) to a list for the loop detection if the branch is not found in the list, and records (increments) the number of executions of the branch instruction in the list. (Either subroutine calls such as JSR (Jump to Sub-Routine) or RET (Return from subroutine) are not added to the list, since they are not related to loop structure. The distinction between such branches and other branches can be done by TDT condition setting described in the section .) When the number of executions exceeds a given threshold, the corresponding backward branch is identified as a loop-back branch instruction.

2) Path Profiling: After a loop-back branch is identified, the TDT configuration is changed for the UDT to generate traps whenever branch instructions are executed. The optimizer routine repeats the analysis of all branches inside the loop structure until the specified number of executions of the loop-back branch instruction is performed.

During the analysis, the optimization generates a weighted digraph for branching direction. Figure 3 illustrates the weighted digraph.

The figure illustrates the weighted digraph after executing ten loop backs. In the figure, nodes (A to F) are depicted for branch instructions inside a loop body, and a loop back branch instruction is represented by H. Edges from a node indicate branch direction. Each edge is accompanied by a direction, `t`(taken) or `nt`(not-taken). The parenthetic numbers denoted near each edge are weight, that is, the number of execution times for the direction.

In the beginning of profiling, there is only an H node. Then, the other nodes are added and the edges are weighted by the optimizer when it is invoked by the UDT. The optimizer searches the digraph for PC of the branch instruction. If hit in the digraph, the number of execution times for the direction is incremented. Otherwise, a new node is created and connected to the previous node. This analysis was continued until the number of execution times of the loop-back edge from the H node became ten.

After the analysis is finished, the optimizer selects a path for generating a software trace by pursuing the weighted digraph from A to H. When an edge have to be selected between taken and not taken, the selection criteria is edge's weight. In the figure, a path, A-B-D-H is selected and the branches and the directions in the path (branch list) are used when generating a software trace, described in the following subsection.

3) Generating a Software Trace: Using the branch list obtained in the path profiling, the optimizer generates a software trace. The generation algorithm is as follows.

1) Extract basic blocks from the branch list and combine the basic blocks,
2) Invert branch condition if the branch was a taken branch,
3) Decide a location (in memory) where the trace is placed,
4) Resolve relocation problems due to PC-relative branches,
5) Add unconditional branches to the trace to return to the original code, and
6) Link the trace with the original code by inserting an unconditional branch into the original code.

B. Sequential Data Prefetching Optimization

Another application of HDOS is optimization for sequential data prefetching. Data prefetch techniques make CPU issue a non-blocking read request before the memory block is actually used. The memory block read from main memory is loaded into the cache in the background of program execution. If the memory block arrives at the cache memory before the block is actually used, the data prefetch can eliminate memory access latency. In this paper, we propose special instructions for sequential prefetching and an algorithm for memory reference analysis.

The Alpha instruction set, originally, provides non-blocking prefetch functions by using the instructions, LDL, LDS, and LDQ, with a destination register R31 or F31 (zero registers). In addition, the FETCHx instruction is generally used to achieve a wide-area prefetch beyond the cache block size. If such prefetch instructions are inserted into a program binary code by the HDOS, target addresses of branch instructions must be changed, which leads to large overheads. To solve this problem, we add new instructions for sequential prefetching,

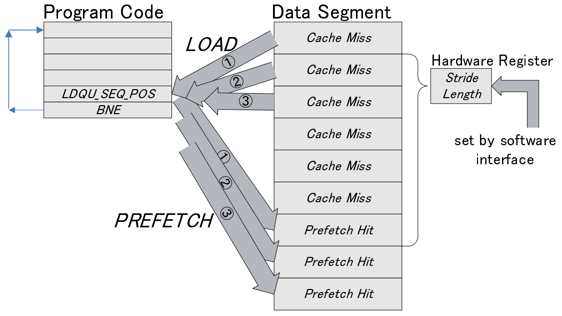

Fig. 4. An example of behavior of SLSI.

sequential load or store instructions (SLSI), to the Alpha instruction set. The instructions have two functions; one is the same as by load or store, and the other is a sequential prefetch function. The two functions are realized in a single instruction. Therefore, the optimization has only to replace a load or store instruction in a program code with the SLSI. The SLSIs can be provided by utilizing miscellaneous instructions in Alpha ISA. Figure 4 illustrates run-time behavior of SLSIs.

In the figure, LDQU_SEQ_POS is one of SLSIs. This instruction executes the following three steps. The first step is to execute the same function as LDQU. The second is to calculate a prefetch address by adding a constant value to the virtual address accessed in the first step. The third is to issue a prefetch using the calculated address. The constant value used in the second step is given by the "Stride Length" register. The stride length is represented by the number of cache blocks. In Figure 4, LDQU_SEQ_POS issues the first prefetch while accessing the first memory block. Then, the second prefetch is issued when the LDQU_SEQ_POS is executed for the second memory block. This is the case in the later execution. Consequently, the LDQU_SEQ_POS constantly prefetches a block ahead by the stride length. In this example, the LDQU_SEQ_POS causes six cache misses. After the sixth miss occurs, the LDQU_SEQ_POS does not generate a cache miss.

1) Memory Reference Analysis and Optimization: There are three steps for the analysis and optimization of a program; Step 1 is for loop detection, which is required since sequential accesses by load or store instructions frequently appear in a loop structure. This step is the same as in the software trace optimization. Therefore, this step is not described again.

Step 2 is to create a memory access history table that holds statistics of memory accesses during the loop execution, examine the table to find load and store instructions that generate sequential accesses, and then list candidates for instructions that could be replaced by SLSI. Step 3, the last step, selects instructions that should be actually replaced out of the candidates and modifies the binary code. The following subsections describe the three steps.

2) Step 2: Creating and Examining Memory Access History Table: After a loop-back branch is identified, a condition is added to the TDT so that the UDT generates traps when load or

PC	Stride	Variable	Stride Length	Last Address	Execute
0x12000	0	0	0	0x200000	1
0x12100	1	0	0	0x210000	30
0x12200	1	0	4	0x220004	100
0x12300	0	1	0	0x230018	50
⋮	⋮	⋮	⋮	⋮	⋮

Fig. 5. The history table of memory references.

store instructions are executed. The optimizer routine repeats the analysis of memory accesses until the specified number of execution of the loop-back branch instruction is performed. In this paper, this job is referred to as an observation phase. Figure 5 illustrates a history table of memory accesses created by the optimizer routine.

The history table in the figure consists of, from left to right, the *PC* value for the load or store instructions, binary value of a flag *Stride* indicating occurrence of stride accesses, binary value of another flag *Variable* indicating occurrence of non-stride (=irregular) pattern accesses, the *Stride Length*, the *Last Address* indicating the address the instruction accesses last time, and *Execute* that is the number of executions of the instruction.

When a UDT trap occurs, the optimizer routine searches the history table by using the *PC* value corresponding to the UDT trap occurrence, found in the RRBE register. If a matching entry is not found, the optimizer routine creates a new entry with initial values for Stride, Variable, and Stride Length, as shown in the first row. Otherwise, it updates the history table by the following four steps.

1) Subtract the *Last Address* from the address referenced by the load or store instruction that caused the trap. Then the *Stride Length (SL)* is obtained.
2) The obtained *SL* is compared with the old *SL* ; if *Variable* is 0 and the obtained *SL* is equal to the old *SL* , then *Stride* is set to 1, *Variable* is to 0, and the *SL* is updated by the obtained value. Otherwise, *Stride* is set to 0 and *Variable* is to 1.
3) The *Last Address* is replaced with the new referenced address.
4) *Execution* is incremented.

The Optimizer routine repeats the steps of (1) to (4) until the end of the observation phase is reached.

After the observation phase, the optimizer routine walks thorough the history table, and finds load or store instructions that generated sequential accesses. The load or store instructions that performed sequential referencing during the observation phase have the *Stride* being 1 and the *Stride Length* being not 0, as shown in the third row of Figure 5.

3) Step 3: Modifying a Binary Code: The optimizer routine replaces the instructions found in the step 2 by SLSIs. If, however, there are too many candidates for the instruction replacement in a loop, replacing all the candidates may lead to unacceptable performance degradation; When a group of SLSIs prefetch more data sets than the number of cache ways (associativity), there is a possibility of cache index conflicts between the prefetch requests, and in the worst-case scenario, it puts the cache at risk for thrashing.

To avoid this problem, the candidates for the instruction replacement are sorted in decreasing order of product of *Stride Length* and *Execute*, and the top N candidates are replaced with SLSIs, where N is the number of cache ways.

TABLE III
SIMULATION PARAMETERS FOR MIPS64 SIMULATOR

issue width	4
branch prediction	ideal
ruu size	infinite

IV. PERFORMANCE EVALUATION

We evaluate performance of two types of the optimized binary codes, software trace optimization and sequential prefetching optimization, compared with non-optimized binary codes by simulation.

A. Simulation Methodology

We used two simulators to evaluate two types of optimizations, respectively. The performance of fetching in software trace optimization is evaluated by the MIPS 64 ISA superscalar CPU simulator we developed. The simulator focuses on instruction fetching. The simulation parameters are shown in Table III.

The sequential prefetching optimization is evaluated by SimpleScalar 3.0 with Alpha ISA [7]. The simulator evaluates total performance of a program execution including cache miss ratio and IPC. We modified the SimpleScalar to model a memory system with prefetch function, and to add SLSIs to the Alpha instruction set. The SimpleScalar simulation parameters are shown in Table IV.

In both simulation models, the optimizer routine were directly implemented by simulator codes.

We used eight applications form SPEC95 INT benchmark suite [8] to evaluate the software trace optimization, and ten applications from SPEC2000 INT benchmark suite [8] to evaluate the sequential prefetching optimization. (The pre-compiled binaries of the SPEC2000 available from [9] were used in the experiment.) All benchmark programs were run until two-billion instructions were committed.

B. Simulation Results of Software Trace Optimization

Figure 7 shows instruction fetching throughput.

In `go`, `m88ksim`, `gcc`, `compress`, `ijpeg` and `perl`, the instruction fetching throughput was improved. However, the `li` and `vortex` got worse than non-optimized binary code.

The degradation happens when target addresses of most branches are variable, that is register jumps are often used, or directions of conditional branches are impartial. The results show the software trace optimization is effective in six of the eight SPEC95 INT applications.

TABLE IV
SIMULATION PARAMETERS FOR SIMPLESCALAR

issue width	4
ruu size	16
dl1 size	64KB (4 WAY/32 byte block)
dl1 latency	1 cycle
il1 size	64KB (4 WAY/32 byte block)
il1 latency	1 cycle
ul2 size	1MB (8 WAY/64 byte block)
ul2 latency	6 cycles
memory access latency	[first]:120 [inter]:12 cycles
memory access bus width	8 bytes

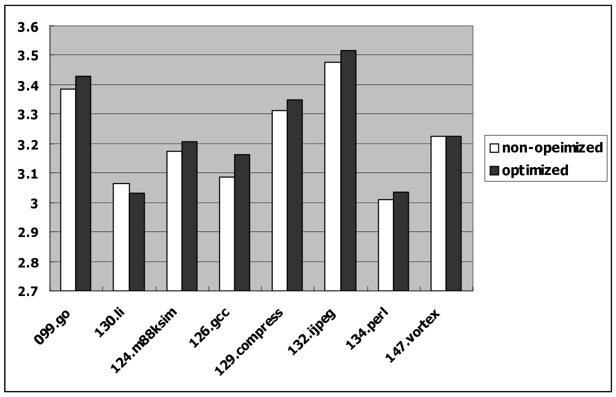

Fig. 6. Instruction fetching throughput for SPEC95 INT.

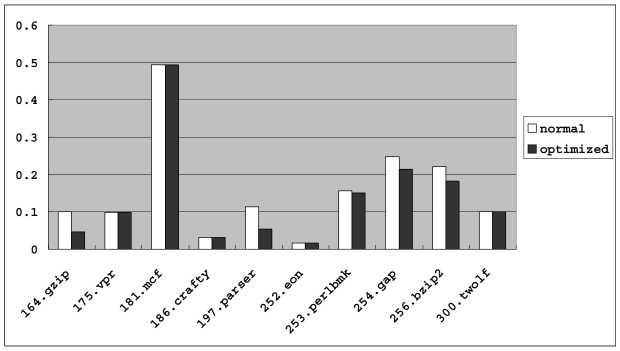

Fig. 7. L2 Cache Miss Rates of SPEC2000 INT.

C. Simulation Results of Sequential Prefetching Optimization

Figure 7 shows cache-miss rates through the program execution. The cache misses in these results do not include misses on blocks for which a prefetch had been already issued.

For all the INT applications, the L2 cache-miss rate was decreased. Especially, in `gzip` and `parser`, the rate was significantly decreased. The `gzip` was most improved, by 53 percent, in all the INT applications. In the `vpr`, `mcf`, `crafty` and `eon`, the L2 cache-miss rate was slightly decreased.

Figure 8 shows IPC performance through the program execution. In five applications, `gzip`, `parser`, `perlbmk`, `gap`, and `bzip2`, the IPC performance was increased.

Especially, the `gzip` was most improved, by 28 percent, in all the INT applications. This is because the performance of

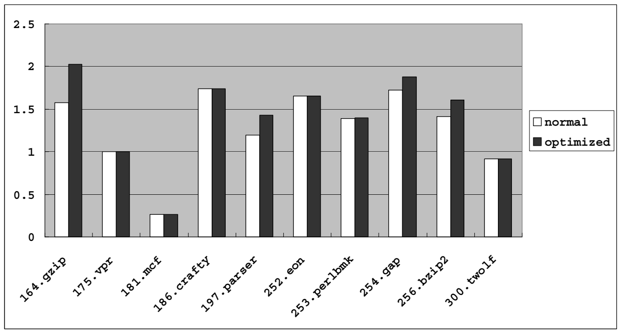

Fig. 8. IPCs of SPEC2000 INT.

the `gzip` depends heavily on accessing to array data structure. In addition, there is non-negligible performance improvement for the `parser`, `gap` and `bzip2`. However, in `vpr`, `mcf`, `crafty`, `eon`, and `twolf`, there is no performance improvement, which means the sequential prefetching optimization is not effective in those applications. The results show the sequential prefetching optimization is effective in five of ten SPEC2000 INT applications.

V. RELATED WORK

In this section, we describe the other dynamic optimization environments.

The recently and relevant work is a data prefetch system that was proposed by Jean et al. [5]. The system does not require any additional hardware such as performance counters or specific hardware. Rather, it only requires software modification of target binaries in advance of execution. For this reason, the system can be implemented on any instruction set architecture which has prefetch instructions. An optimizer routine which monitors the program behavior and modifies a target binary is implemented by a signal handler and a thread that is invoked periodically.

There are two advantages of our system over the above system. One is that our system can use precise informations such as virtual addresses actually issued by using the specific hardware (UDT). The other is simple binary modification by introducing new instructions dedicated to sequential accesses (SLSI). Using the SLSIs to modify a binary code, overheads of fetching conventional prefetch instructions that would be added to the code can be avoided.

There is a disadvantage of our system. Our system requires modifications to conventional hardware and operating system software. However, the UDT hardware can be used for not only the dynamic optimization but also the other purposes such as performance analysis of programs. For example, the hardware can be used for the same purpose as the PMU (Performance Monitoring Unit) [3] system which is implemented on Intel Itanium2.

Dynamo[1] and DynamoRIO[2] are a transparent dynamic optimization environment. They operate on unmodified native binaries and require no special hardware or operating system support. Making a binary code run on the interpreter, the

Dynamo can obtain informations to optimize the binary code and timing for optimization. Then, using the informations, the Dynamo generates hot traces that would be frequently executed and therefore can reduce cache misses while the hot traces are executed. However, the overhead of executing the original code on the interpreter can diminish the effect of the optimization. The DynamoRIO is a framework for dynamic code modification systems that are based on Dynamo and can be used to profile and optimize programs. UMI[4] is a lightweight dynamic optimization system built on the DynamoRIO. The UMI and DynamoRIO can bring the same benefits as Dynamo without heavy overhead. There is our advantage over their works. They have to use the dynamic optimization environment every time a program is executed, which incurs the overhead constantly. In our system, on the other hand, the optimization overhead is incurred only at the first execution, since our system directly modifies binary code and can easily disable the dynamic optimization functions at the second run-time or later.

VI. Conclusion

In this paper, we proposed a new dynamic optimization system which can implement several optimization algorithms. This system is called HDOS. The HDOS is organized by simple trap hardware which is called UDT and a trap handler, a part of operating system codes, which is used as the optimizer routine, and creates an optimized binary code. As examples of applications of the HDOS, we showed two binary code optimization algorithms. One is software trace optimization, and the other is sequential prefetching optimization. The software trace optimization can improve efficiency of instruction fetching. The sequential prefetching optimization can reduce cache misses in sequential memory accesses.

The two optimization algorithms were evalueated in simulation. The simulation results of SPEC95 benchmarks showed that the software trace optimization was effective in six of eight applications. The simulation results of SPEC2000 benchmarks showed that the sequential prefetching optimization was effective in five of ten applications. In the other five applications, the sequential prefetching optimization did not much degrade the performance.

References

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

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

[3] Intel Itanium 2 Processor Reference Manual for Software Development and Optimization. http://download.intel.com/design/Itanium2/manuals/25111003.pdf

[4] Q.Zhao, R.Rabbah, S.Amarasinghe, L.Rudolph, and W.-F.Wong. Ubiquitous memory introspection. In international Symposium on Code Generation and Optimization 2007, Washington, DC, USA, March 2007.

[5] Jean Christophe Beyler and Philippe Clauss. Performance driven data cache prefetching in a dynamic software optimization system. International Conference on Supercomputing archive Proceedings of the 21st annual international conference on Supercomputing. Pages: 202 - 209.

[6] Alpha Architecture Reference Manual, THIRD EDITION. ALPHA ARCHITECTURE COMMITTEE, Digital Press, ISBN 1-55558-202-8.

[7] D.Burger, T.Austin, and S.Bennett. Evaluating future microprocessors: The simplescalar toolset. Tech Report CSTR-96-1308, Univ. of Wisconsin, CS Dept., July 1996.

[8] http://www.spec.org

[9] http://www.simplescalar.com