Dynamic Binary Code Translation for Data Prefetch Optimization

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

Abstract

Recently, CPUs with an identical ISA tend to have different microarchitectures, 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. On the other hand, there is a significant gap between processor's operational speed and memory access speed, and currently the gap is increasing. In this paper, we introduce several special prefetch instructions that are suited for memory access patterns that frequently appear in program execution. However, such special instructions are utilized only by compiler's machine-dependent code optimization, and therefore software vendors do not utilize such instructions. To increase opportunities for effectively exploiting the instructions for optimization, we propose dynamic optimization techniques that consist of dynamic code modification and analysis methods of memory references. We evaluate the techniques by using SPEC2000 benchmarks.

1 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 cases, 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, an infrastructure for dynamic optimization, *Hybrid Dynamic Optimization System (HDOS)* which is proposed by our previous work [1], is exploited. 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 or later execution time without optimization overhead. The HDOS is organized with dedicated hardware inside a CPU and operating system support. The dedicated hardware is called *User Definable Trap (UDT)*. Software called by the trap is optimizer routines which is provided by an operating system.

We focus on data prefetch optimizations as application of the HDOS. 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 ex-

978-1-4244-2683-6/08/$25.00 ©2008 IEEE

ecution. If the memory block arrives at the cache memory before the block is actually used, the data prefetch can eliminate the memory access latency. Three data prefetch methods, sequential prefetch, indirect prefetch, and indirect sequential prefetch are proposed in this paper. We introduce three types of instructions to implement the three prefetch techniques. By analyzing memory references and replacing existing load and store instructions with the instructions, the HDOS optimizes target binary codes.

This paper is organized as follows: Section 2 gives an overview of the HDOS. Section 3 describes a method of analysis of memory references and the three prefetch optimizations. Section 4 shows how to reuse optimized binary codes provided by the HDOS. Section 5 shows results of performance evaluations of prefetch optimization by the HDOS. Section 6 describes related works of prefetch methods and dynamic optimizations. This paper is concluded by Section 7.

2 Overview of 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. In this section, an overview of the HDOS is introduced.

The 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.

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 search 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 the 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.

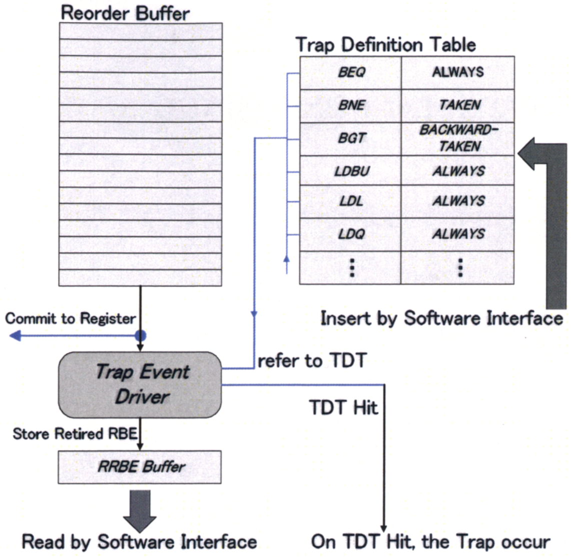


Figure 1. User Definable Trap (UDT) Hardware.

3 Analysis and Optimizations

The HDOS is a client-side optimization method as noted in the Section 1. For this reason, the HDOS makes it easy to exploit new instructions for the optimization, since the optimization is a machine-dependent code optimization. Three types of instructions for the optimization are introduced in this paper. The instructions, sequential prefetch, indirect prefetch, and indirect sequential prefetch instruction, are suited for memory access patterns frequently appearing in a program. The sequential prefetch instruction is effective in sequential accesses such as accesses to array data structure. The instruction is referred to as *Sequential Load or Store Instruction (SLSI)* in this paper. The indirect prefetch instruction is effective in indirect accesses such as accesses via pointer references. The instruction is referred to as *Indirect Load Instruction (ILI)*. The indirect sequential prefetch instruction is effective in indirect accesses such as accesses via pointer-array references. The instruction is *Sequential Indirect Load Instruction (SILI)*. In later subsections, the instructions and analyses which is required to exploit the instructions are described in detail.

3.1 SLSI

One of applications of the HDOS is to utilize SLSIs. This instruction has 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.

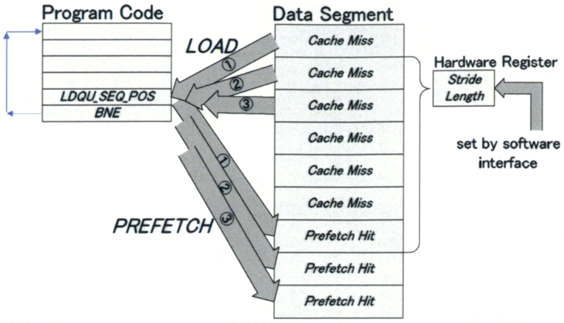


Figure 2. An example of behavior of SLSI.

Therefore, the optimization has only to replace a load or store instruction in a program code with the SLSI.

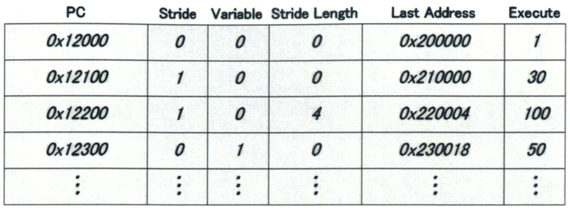
3.1.1 Behavior of SLSI

Figure 2 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 (LDQU is one of load instruction in Alpha[2] instruction set). 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 2, 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 cache misses.

3.1.2 Analysis and Optimization for SLSI

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. 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.



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
⋮	⋮	⋮	⋮	⋮	⋮

Figure 3. The history table of memory references.

In the step 1, the optimizer routine finds loop structure as follows. The HDOS creates TDT entries to generate traps when a 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. When the number of executions exceeds a given threshold, the corresponding backward branch is identified as a loop-back branch instruction.

In the step 2, after a loop-back branch is identified, a condition is added to the TDT so that the UDT generates traps when load or 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 the observation phase. Figure 3 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 new *Stride Length (SL)* is obtained.

2. The obtained *SL* is compared with the old *SL* in the table; if the flag, *Variable* is equal to 0, and the obtained *SL* is 0 (initial value), or 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 3.

In the step 3, 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 *Stride Length*, and the top *N* candidates are replaced with SLSIs, where *N* is the number of cache ways.

3.2 ILI

In high level programming languages such as C language, programmers often use pointers to cut searching and handling time for list and tree data structure. When pointer variables are accessed, virtual addresses which is held by the pointer variables are stored in cache memory. To utilize the characteristic, ILI is added to instruction set. The instruction has two functions; one is the same as by load, and the other is an indirect prefetch function. The two functions are realized in a single instruction.

3.2.1 Behavior of ILI

The indirect prefetch function is provided by a cache memory system. The cache memory has additional control bits which indicates whether the block includes memory addresses (pointer values). When a cache block is loaded, the P-Bits are set to 0. The execution of the ILI sets the control bit to 1 before it performs the load function. In this paper, the control bit is called Pointer-indicating Bit (P-Bit), and the cache system is called pointer chasing cache (PC-Cache). Figure 4 and Figure 5 show meanings of P-bits and a block diagram of PC-Cache, respectively.

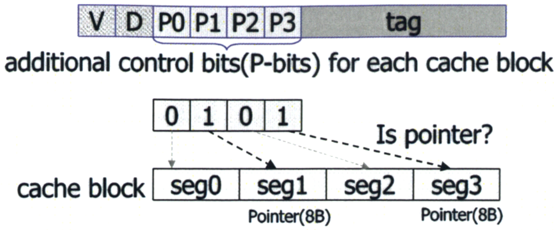
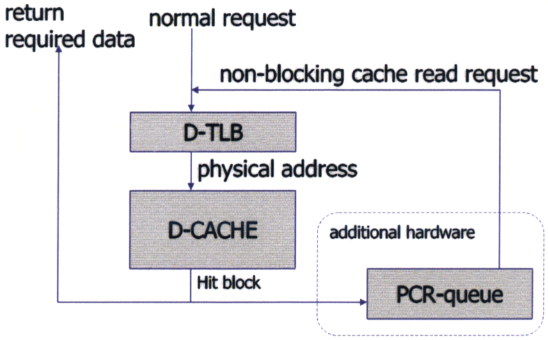


Figure 4. Meanings of P-Bits.



If the required data's P-bit is one, enter the PCR-queue.

Figure 5. Block diagram of PC-Cache.

In the Figure 4, it assumes that a cache block size is 32 bytes and an address is 64 bits (8 bytes). In this cache configuration, the four P-Bits for a cache block are required. Individual bit indicates whether the corresponding segment holds memory address (pointer value) or not. When the cache is accessed, the cache checks P-Bits of a cache block. Values in the segments for which the corresponding P-bit is 1 are used as an address for prefetching.

In the Figure 5, Pointer Chasing Request queue (PCR-queue) is added to a conventional data cache. If the hit block includes pointer values, all the pointer values are inserted into the PCR-queue while the cache returns the requested data. When the PCR-queue finds idle cycles for TLB, it issues a pointer value in the top entry to TLB. If the pointer value can be translated to physical address, the physical address is issued to the data cache as a non-blocking read request. If the request causes a cache miss, the request becomes a prefetch request. On the other hand, if the request hits in the cache, P-Bits of the hit block are checked and then the pointer values, if any, are issued again. This behavior means “pointer chasing”.

3.2.2 Analysis and Optimization for ILI

To exploit the ILI on the HDOS, references to pointer variables have to be found in program execution. If the references are found, the HDOS simply replaces the load instructions for the references with ILI. To find the load instructions, the HDOS can use the history table described in the section 3.1. First, load instructions that have 1 for the *Variable* , such as in the fourth row of Figure 3 are listed. Then, the optimizer routine analyzes the binary code. If a base register of the load instruction is a destination register of the other (preceding) 8-byte load instruction [1], the latter instruction can be replaced with ILI.

3.3 SILI

The SILI has both SLSI and ILI functionalities. In the first step of SILI execution, the SILI executes the same function as ILI except that it does not set P-Bits of the target block. At the same time, it executes the same function as SLSI for 8-byte load. Then when the prefetched block arrives at cache memory, the block with all P-Bits asserted is stored in the cache memory and all 8-byte values in the block are inserted into the PCR-queue. Using the SILI, pointer-array data can be prefetched with data that are referenced by the pointers, in more advance than using ILI.

3.3.1 Analysis and Optimization for ILI

To exploit the SILI on the HDOS, the analysis is based on SLSI analysis. At the same time as analysis for SLSI, the HDOS checks possibility of SILI. If the candidates for replacement with 8-byte SLSI are found, the optimizer routine analyzes a binary code. If a destination register of the load instruction is used for a base address of the other load instructions, the former instruction can be replace with SILI.

4 Reusing Optimized Binary Codes

The HDOS can transparently optimize target binary codes, since the optimization does not require any run-time software environment such as virtual machines and interpreters. Moreover, the optimization does not require any advance modification of target binary codes, such as code augmentation. However, overheads of the optimization are not negligible. For example, cost for executing the optimization routine and inefficient pipeline execution due to generating UDT trap can diminish the effect of the optimization. Though, optimized binary codes can be reused because of the characteristic of “translation from native binary to native binary”. In this section we describe how to reuse optimized binary codes.

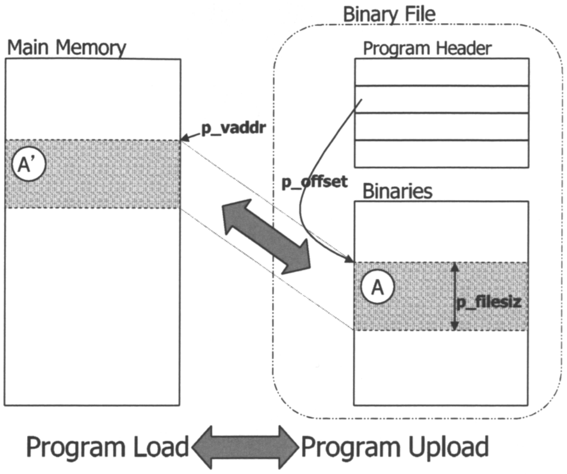


Figure 6. An Example of Program Uploader on ELF Executable Binary Format.

To reuse the results of the optimization, operating system support is needed. We propose a binary code dump method which reorganizes an executable file from a program binary code located in main memory. This method is called a program uploader.

Figure 6 shows a program upload example of the case of ELF32 executable binary format in UNIX system.

In general, a program loader which is included in an operating system function such as UNIX `exec` system call locates a binary code on main memory by using information on a program header table in an executable file. The each program header entry has the following information.

```
typedef struct {
  uint32_t  p_type;
  Elf32_Off p_offset;
  Elf32_Addr p_vaddr;
  Elf32_Addr p_paddr;
  uint32_t  p_filesz;
  uint32_t  p_memsz;
  uint32_t  p_flags;
  uint32_t  p_alin;
} Elf32_Phdr;
```

An important information of ELF32 header entry is a set of `p_offset`, `p_vaddr` and `p_filesz`. The `p_offset` represets an offset from head of the executable file. The `p_vaddr` represents memory address that will be loaded program binary. The `p_filesz` represets size of program binary that is copied onto main memory. Before

[1] In 64-bit architecture, pointer values are represented in 8-byte data.

starting program execution, as in the figure, the operating system sets the file region A on the segment of main memory A' by using the infomations.

In our proposal, when program execution is over, the program uploader replaces a binary code of the read only segment in the executable file with the optimized binary code existing in main memory, referring to information on the program header. The p_type holds segment information such as read only segment. Program text and read only data segment should be set by read only. Therefore the program uploader checks p_type information of program header and finds segments for program text at first. Then the program uploader replaces file region (A) with memory segment (A') by using p_offset,p_vaddr and p_filesz information related to the text segment.

This method should be provided by a system call. In general UNIX system, if the shell uses the system call, the HDOS and the optimizations can be transparent for system user and application programmer.

5 Performance Evaluation

In this section we evaluate performance of binary codes optimized by the HDOS.

5.1 Simulation Methodology

The simulation environment is based on SimpleScalar 3.0 using Alpha binaries[3]. We modified the SimpleScalar to model in detail a memory system with prefetch functions and PC-Cache and to add SLSIs, ILI and SILI to the Alpha instruction set. The instructions can be provided by utilizing miscellaneous instructions in Alpha ISA.

The UDT hardware was not implemented in the simulation model. Instead, the UDT trap detection and the optimizer routine were implemented as simulator codes. The sim-outorder was simulated with the architecture and functional resources shown in Table 1. The HDOS parameters are shown in Table 2.

We used twenty-one applications from SPEC2000 benchmark suite [4]. The pre-compiled binaries of the SPEC2000 available from [5] were used in the experiments. All the benchmarks were run until two-billion instructions were committed.

5.2 Performance Evaluations for SLSI

Figure 7 and 8 respectively show cache-miss rates and IPC performance through the program execution when the HDOS utilizes the SLSI, ILI, SILI. The optimizations are indicated as individual bars, from left to right, SLSIs, ILI, SILI, and ALL of the three optimization at the same run time. Note that the cache misses in these results do not include misses on blocks for which a prefetch had been already issued.

Table 1. Simulation Parameters of SimpleScalar

issue width	4
data-l1 cache	64KB 4WAY 32B-BLOCK
data-l1 access latency	1 cycle
inst-l1 cache	64KB 4WAY 32B-BLOCK
inst-l1 access latency	1 cycle
unified-l2 cache	2MB 8WAY 64B-BLOCK
unified-l2 access latency	6 cycles
memory access latency	[first]:120 [inter]:12 cycles
memory access bus width	8 bytes

Table 2. Parameters of the HDOS

threshold of backward loop count	100
loop-back count for mem-access observation	100
stride length(for SLSI,SILI)	20

Using SLSI optimization, in the applications except art, the L2 cache-miss rate was decreased. Especially, in gzip, mcf, wupwise and equake, the miss rate was significantly decreased. In the applications except art, the IPC performance was increased. Especially, the gzip was most improved. This is because the performance of the gzip depends heavily on accessing to array data structure. Only for the art, the miss rate and the IPC performance were got worse, since a prefetch by SLSI caused inefficiency of the cache usage. The results show the SLSI optimization is effective in eleven of twenty-one SPEC2000 applications.

The ILI could not be applied in almost all FP benchmarks, and the performances were the same as non-optimized binary codes. Using ILI, in all of the INT applications, the rate was decreased. Especially, in mcf, significant decreasement was observed. This is because the miss rate in the mcf depends heavily on accesses using pointer values. Especially, in mcf, perlbmk, and gap, nonnegligible performance improvement was observed. In the applications, the performance depends on indirect accessing. The performance degradation is not observed in the ILI results.

In the results of SILI, both of the results are similar to the ILI results. Main differences from the ILI results are

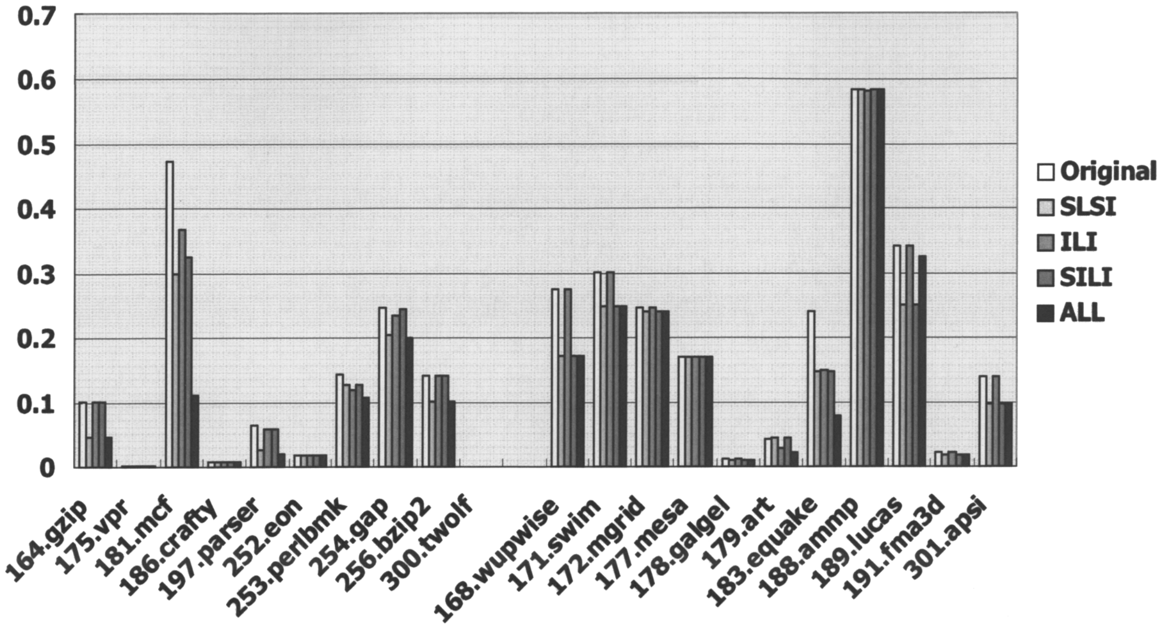


Figure 7. L2 Cache Miss Rate for SPEC2000.

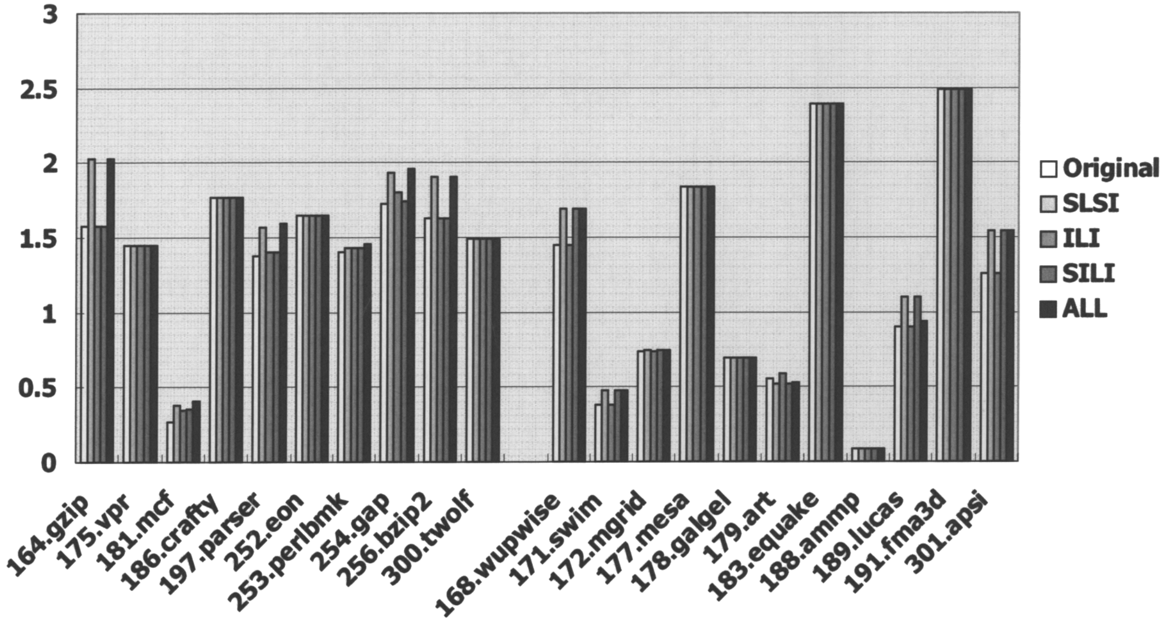


Figure 8. IPC Performance for SPEC2000.

the `mcf` and `gap`. In the `mcf`, cache-miss rate was more decreased than the ILI. It indicates that the `mcf` has many or large pointer-array data structures. In the `gap`, cache-miss rate was less decreased than the ILI. It indicates that the `gap` has pointer data structures where the data structures are not pointer-array data structures but tree or list data structure.

The three optimizations, SLSI, ILI, and SILI optimization, can be exploited simultaneously. Finaly, we evaluated total performance of the multiple optimizations. In many applications, cache-miss rates and IPC performance are improved more than individual optimization. Especially, in the `mcf`, cache-miss rate was greatly decreased. We confirmed that the both of SLSI and SILI optimizations were effective in this program.

6 Related Work

In this section, we describe other prefetch techniques and dynamic optimization techniques as related works. There are three approaches for prefetching. software prefetch, hardware prefetch, and dynamic prefetch optimization.

The software approach requires the compiler to insert prefetch instructions at the most appropriate places in an assembly language code of the program [6, 7]. Advantageous points of software approaches are flexibility in handling memory access trends which differ from program to program and unnecessity of any hardware assistance.

Hardware approach uses additional hardware in the CPU. The hardware issues a prefetch in the background of program execution, which can reduce memory access latency. Several hardware prefetch techniques have been proposed [8, 9, 10], which find relatively simple memory accesses, such as stride memory accesses, through address analysis, and apply prefetching to the accesses. The techniques allow prefetch to be issued with high accuracy by using the hardware dedicated to the memory access analysis.

The dynamic prefetch optimization has advantages of both hardware and software techniques. One of dynamic prefetch optimization techniques for a data prefetch was proposed by Jean et al. [11]. The system does not require any additional hardware. Rather, it only requires software modification of target binaries in advance. An optimizer routine which monitors the program behavior and modifies a target binary is implemented by a signal handler and a thread which is invoked periodically. There are two advantages of our system over the 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 several accesses patterns (SLSI, ILI, SILI). Using the instructions, overheads of fetching additional prefetch instructions can be avoided.

7 Conclusions

In this paper, new instructions, SLSIs, ILI and SILI are proposed. The instructions are exploited by dynamic optimization system which is called HDOS, and are effective in memory access patterns that frequently appear in program execution. We proposed an analysis technique which can find instructions related to sequential, indirect, and sequential-indirect access patterns by using the UDT support. Those instructions can be replaced with the proposed instructions. The simulation results of SPEC2000 benchmarks showed that the optimization was effective in most applications, especially when the three optimizations were combined.

References

[1] T.Ukezono, K.Tanaka, Dynamic Binary Optimization Using Hardware Analysis System, IPSJ SIG Technical Reports, ARC, Vol.2005, No.120, pp.7–12, 2005 (in Japanese).

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

[3] 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.

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

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

[6] V.Santhanam, E.H.Gornish and W.C.Hsu, Data Prefetching on the HP PA-8000, Proc. of 24th annual International Symposium on Computer Architecture, pp.264–273, 1997.

[7] K.C.Yeager, The MIPS R10000 Superscalar Microprocessor, IEEE Micro, Vol.16, No.2, pp.28–41, 1996.

[8] J-L.Baer and T-F.Chen, Effective Hardware-Based Data Prefetching for High Performance Processors, IEEE Transactions on Comuters, Vol.44, No.5, pp.609–623, 1995.

[9] F.Dahlqren, M.Dubois, and P.Stenstrom, Fixed and Adaptive Sequential Prefetching in Shared Memory Multiprocessors, Proc. of International Conference on Parallel Processing, pp.I-56–63, 1993.

[10] J.W.C.Fu, J.H.Patel and B.L.Janssens, Stride Directed Prefetching in Scalar Processors, Proc. of 25th International Symposium on Microarchitecture, pp.102–110, 1992.

[11] J.C.Beyler and P.Clauss, Performance Driven Data Cache Prefetching in a Dynamic Software Optimization System, Proc. of the 21st International Conference on Supercomputing, pp.202–209, 2007.