A Novel Chaining Approach to Indirect Control Transfer Instructions

Wei Chen, Zhiying Wang, Qiang Dou, Yongwen Wang

A Novel Chaining Approach to Indirect Control Transfer Instructions

Wei Chen[1] , Zhiying Wang[1], Qiang Dou[1], Yongwen Wang[1]

[1] School of Computer, National University of Defense Technology,
Changsha 410073, Hunan, China
{chenwei}@nudt.edu.cn

Abstract. Both dynamic binary translation systems and optimization systems store the translated or optimized code in the software maintained code cache for reuse. The performance of the code cache is crucial. Translated code is usually organized as code blocks in the code cache and each code block transfer control to the next one through a control transfer instruction. As the target address of a control transfer instruction is in the form of its source program counter, the conventional code cache system has to check the address mapping table for the translated target address to find the required target code block, which will cause considerable performance degradation. Control transfer instructions can be divided into two categories as direct control transfer instructions and indirect control transfer instructions. For indirect control transfer instructions, the target address is hold in the register or memory element whose content can be changed during the execution of the program. It is difficult to chain the indirect control transfer instructions with a fixed translated target address through pure software approaches. A novel indirect control transfer chaining approach is proposed in this paper. The principle of the technique is to insert custom chaining instructions into the translated code block while translating the indirect control transfer instructions and execute those chaining instructions to implement dynamical chaining. Some special hardware and software assists are proposed in this paper. Evaluation of the proposed approach is conducted on a code cache simulator. Experiment results show that our hardware assisted indirect control transfer instruction chaining approach can improve the performance of the code cache system dramatically.

Keywords: code cache, indirect control transfer instruction, dynamic chaining, simulator

1   Introduction

Software maintained Code Cache Systems (CCS) [1] have been widely used by many binary translation or optimization systems (optimization system is a kind of translated system in which translation is performed between the same instruction set architecture; in the rest of the paper, the translation system includes the optimization system if not especially declared) to store the translated or optimized code for reuse. As more and more translation systems have been proposed and developed, the performance of the

code cache systems becomes a hot topic in both the industry and academic fields, especially for those virtual computing environments.

In the code cache, the translated code is organized in terms of code blocks. Each code block ends with a control transfer instruction which switches control flow between different code blocks. When a source code block is translated the first time, its target code block may have not been translated most of the time. Thus, the translated code block always keeps the target address of the control transfer instruction in terms of its original Source Program Counter (SPC). The conventional code cache system always maintains an Address Mapping Table (AMT) [2] for recording the code block's SPC and its corresponding Translated Program Counter (TPC). During the execution, CCS finds the required translated target code block by looking up AMT as Fig. 1(a) [2] shows.
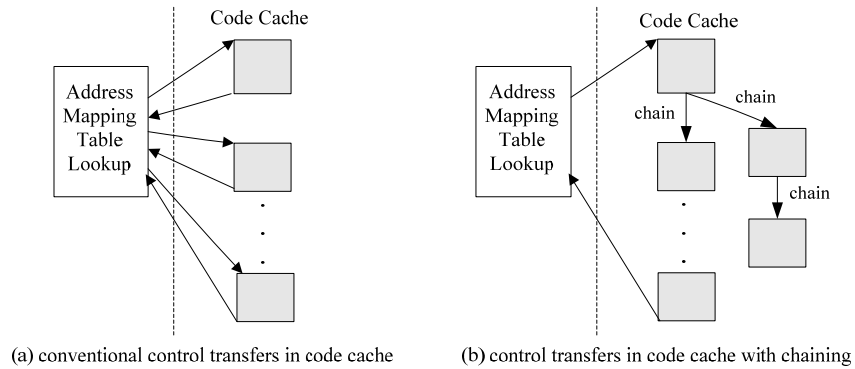

(a) conventional control transfers in code cache   (b) control transfers in code cache with chaining

Fig. 1. Control flow changes among code blocks in the code cache.

When executing within a code block, the execution/control flow is straight-line. However, transitions from one code block to another may cause performance degradation because the control flow is changed and costly address mapping table lookup mechanism must be invoked. In our code cache simulator, an AMT lookup may cost 15~25 local instructions while a context switch may cost about 40 local instructions. Therefore, if the control transfer instruction can be chained directly to the translated target address (as Fig. 1(b) shows), the performance of the code cache system may be dramatically improved as table lookup and context switch operations can be avoided.

Control transfer instructions can be divided into two categories as direct control transfer instructions and indirect control transfer instructions. For direct control transfer instructions, the target address always appears as an immediate value in the instruction itself. So, no matter the immediate value indicates an address or an offset, the target address of a direct control transfer instruction can be calculated from the instruction itself and is fixed during execution. Thus, the SPC of the direct control transfer instruction can be placed with its TPC through software mechanisms. We have proposed an effective solution to this problem in our previous work [2].

For indirect control transfer instructions, the target address is saved in a register or a memory element. The indirect control transfer instruction figures out its target

address by reading the register or the memory element. During execution, the content of the register or memory element can be changed, which means that the target address of the indirect control transfer instruction is not fixed. Therefore, it is difficult to fix the indirect control transfer instruction's target address with a fixed TPC. At present, as we know, there is not a pure software approach which can solve this problem effectively.

In this paper, a hardware assisted Indirect Control Transfer Chaining (IDCTC) method is proposed for the indirect control transfer instruction. The principle of IDCTC is: (1) while translating the source code block, inserting custom chaining instructions into the translated code block which contains indirect control transfer instructions, (2) during execution, executing the inserted chaining instructions to dynamically determine the translated target address. Special hardware/software assists are occupied by IDCTC. The key of IDCTC is a hardware assist called indirect transfer target buffer, which saves most frequently accessed indirect control transfer instructions.

To evaluate the efficiency of IDCTC, we conduct experiments on a code cache simulator and the experiment results show that IDCTC can dynamically chain the indirect control transfer instructions and dramatically improve the performance of the code cache system.

The rest of the paper is organized as follows. Section 2 introduces some related works. Section 3 introduces the hardware and software assists occupied by IDCTC. Section 4 describes the process of our indirect control transfer chaining approach. Section 5 presents the evaluation of IDCTC and section 6 concludes this paper.

2 Related Work

Code cache systems are the critical part in a binary translation system or optimization system. All the translated codes are cached in the code cache system for reuse. Transmeta Crusoe [3] use the Code Morphing Software (CMS) to translate x86 binaries into Very Long Instruction Word (VLIW) forms. The translated VLIW instructions are stored in the code cache as superblocks. IBM DAISY [4] translates PowerPC binary codes into its VLIW version as "tree regions", which are cached in the code cache for reuse. IA-32 EL [5] translates IA-32 instructions into Itanium instructions. It is a two-stage translator that begins with a simple basic block translator and invokes an optimizing translator once hotspot code is detected. Both the basic code blocks and superblocks are stored in the code cache. The well-known optimization system HP Dynamo [6] optimizes the hotspot of the source code and stores the highly optimized code in the code cache in terms of trace. Other successful translation /optimization systems which employ code caches include: FX!32 [7], UQDBT [8], Strata [9], DELI [10], et al.

As the performance of the code cache system has great impact on the translation system and optimization system, quite a few researchers put their interests on the approaches to improving the performance of the code cache system. Kim works on the code cache management schemes and proposed a generational code cache based code cache management algorithm [1]. This algorithm categorizes code traces based on their expected lifetimes and groups traces with similar lifetimes together in separate storage

areas. Using this algorithm, short-lived code traces can easily be removed from a code cache without introducing fragmentation and without suffering the performance penalties associated with evicting long-lived code traces.

A software prediction approach is proposed in [11] to predict the target address of the indirect control transfer instructions. The principle of the prediction based approach is to compare the content of the register of the indirect control transfer instruction with the pre-defined target SPC, which is most likely to be the real target SPC. If the value of the register matches with the predicted SPC, the control flow can be easily transferred to the required TPC which is corresponding to the pre-defined SPC. The software prediction based approach actually provides a kind of chaining of the indirect control transfer instruction and is easy to be implemented. The prediction depth can be configured flexibly. The shortage of this approach is that a large number of instructions should be inserted into the translated codes in order to implement the prediction. This means considerable code expansion which may offset the performance improvement. Especially in the case that the prediction is failed after several times of comparing, the software approach will cause extra overhead.

In our previous work, we proposed a software implemented Direct Control Transfer instruction Chaining (DCTC) [2] approach to chain the direct control transfer instructions. The principle of DCTC is to replace the target address of the direct control transfer instruction with its TPC, in order to reduce the context switching and the address mapping table lookup operations. DCTC adopts an address mapping table and a special direct control transfer target address mapping table to assist the chaining process. DCTC has been demonstrated its efficiency in chaining the direct control transfer instructions. If the indirect control transfer instructions can be chained effectively, the performance of a code cache system can be further improved.

In this paper, we focus on improving the performance of the code cache through chaining the indirect control transfer instructions.

3   Software and Hardware Support for IDCTC

3.1   Hardware Support for IDCTC

It is difficult to chaining the indirect control transfer instructions through only software approaches. Thus, in this paper, we propose a novel hardware support called Indirect Transfer Target Buffer (ITTB) to support chaining the indirect control transfer instructions to their translated targets.

ITTB is implemented as a high speed buffer on chip which can provide one read/write operation in one clock cycle. ITTB is used to store the frequently accessed target address of the control transfer instructions. Obviously, ITTB provide a copy of AMT. As ITTB is a hardware approach, target address can be looked up quickly. The principle of ITTB is similar to Translation Look-aside Buffer (TLB) [12] used in the management of virtual memory space. Fig. 2 shows the infrastructure of ITTB.
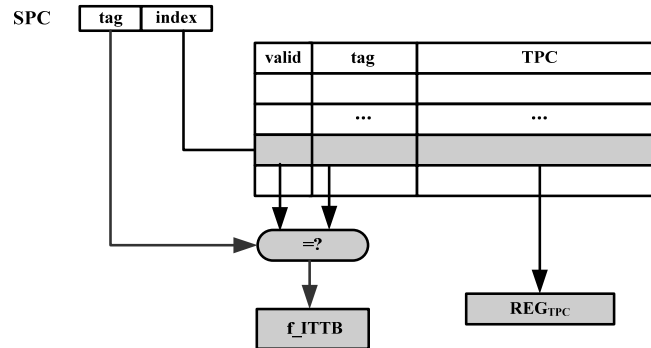
Fig. 2. Infrastructure of indirect transfer target buffer.

ITTB stores the corresponding TPC of indirect control transfer instructions, and is accessed by the SPC of the indirect control transfer instructions. SPC is divided into tag and index, while ITTB use valid bit and tag to check whether the required SPC is matched in ITTB. ITTB can be implemented as a cache, which can be organized with multi-way associativity and replacement policies. The ITTB accessing result will update the flag f_ITTB, if it is hit, this flag will be set as 1, or it will be cleared. If it is hit in ITTB, the required TPC will be written into the special purpose register REG_{TPC}, which is custom designed in the microprocessor for ITTB. Table 1 gives out the description of f_ITTB and $REG_{TPC.}$

Table 1. Special purpose registers custom designed for ITTB.

Register	Description
REG_{TPC}	Register used to store address of the translated code block
f_ITTB	Status flag indicting whether ITTB is hit or not; f_ITTB can be implemented using reserved bit of the status flag register of the microprocessor as most modern microprocessors have reserved status bits and it can also be implemented through custom individual status flag register

As ITTB is custom designed hardware, it should be accessed via custom designed instructions as "search_ITTB, reg" and "store_ITTB, <SPC, TPC>" which are added to the instruction set of the microprocessor.

- Search_ITTB, reg: this instruction looks up ITTB according to the content of the register '"reg", which is corresponding to the register used by the register control transfer instructions. The value of "reg" is the SPC of the target address.
- Store_ITTB, <SPC, TPC>: this instruction update the corresponding ITTB element related to SPC with the value of TPC.

The above two instructions are custom designed for ITTB access. Furthermore, the ITTB checking result will be used by other two instructions which should also be

provided in the microprocessor as "JMP REG$_{TPC}$" and "JITTB, imm". The former transfers control to the address indicated by REG$_{TPC}$ while the latter checks the status of flag f_ITTB. Detailed description of the custom instructions can be found in Table 2.

Table 2. Custom designed instructions for using ITTB.

Instruction	Description
Search_ ITTB, reg	Lookup ITTB according to the value of register "reg"; if it is hit, write TPC into REG$_{TPC}$, if not, clear flag f_ITTB
Store_ITTB, <SPC, TPC>	Write TPC into the ITTB element according to the SPC
JMP REG$_{TPC}$	Jump to the address indicated by REG$_{TPC}$
JITTB, imm	Check if ITTB access is hit. If it is hit, transfer control to the instruction whose address has an offset of "imm" with the current instruction, if not, then execute the next instruction.

In some microprocessors, the instruction set architecture may not support the instruction format that one instruction consists of two immediate values. In that case, "strore_ITTB" can be implemented through several register-register instructions. First, write SPC and TPC into two registers, and then execute "strore_ITTB, reg1, reg2" like instruction. How to implement "store_ITTB, <SPC, TPC>" depends on the instruction set architecture of the target microprocessor.

3.2   Software Support for IDCTC

Besides the hardware support, DBT systems should also maintain some software data structures to assist the chaining of the indirect control transfer instructions. One is the Address Mapping Table [2] which stores the SPC and TPC of the code blocks. The other is called T-SPC, which is a hash table used to save any possible source programming counter (SPC) of the target address of the indirect control transfer instructions. T-SPC will be updated under two situations:

- During profiling, any time an indirect control transfer instruction is executed, the value (SPC of the target address) of the register or memory used by the indirect control transfer instruction should be written into T-SPC.
- When a translated code block transfers the execution to VMM because of the fact that the indirect control transfer instruction at the end of the code block has

not been chained, VMM should write the target SPC of the indirect control transfer instruction into T-SPC.

T-SPC is used to update ITTB when a new code block is translated.

4 Indirect Control Transfer Instruction Chaining

Based on the hardware and software assists introduced in section 3, we propose a novel indirect control transfer instruction chaining approach. The basic principle is: 1) to insert specific instructions which will directly chain the indirect control transfer instructions when translating the source code block; 2) to execute these specific chaining instructions before executing the indirect control transfer instructions during the execution of the translated code block.

4.1 Insert Chaining Instructions

The first step is to insert custom chaining instructions and update the AMT, ITTB and T-SPC, when generating the translated code block which contains indirect control transfer instructions.

Insert custom chaining instructions during translation

---
1. Update AMT, write SPC and TPC of the current code block into AMT;
2. Insert chaining instructions into the current translated code block at the place where the indirect control instruction should appear;
3. Lookup T-SPC according to the SPC of the current code block;
4. if the required SPC exists in T-SPC
5.   Execute "Store_ITTB, <SPC, TPC>" instruction to update ITTB;
6.   Update T-SPC, delete the SPC from T-SPC;
7. endif

Fig. 3. Insert custom chaining instructions and update related software/hardware assists.

Though, different indirect control transfer instructions may require different sequence of chaining instructions, the translating process or the inserting process is similar. Thus, we take the indirect jump instruction for example. The indirect jump instruction stores the target address in a register. When translating the source indirect jump instruction, IDCTC has to insert those custom chaining instructions at the place where the translated indirect jump instruction should appear. Fig. 4 indicates how to insert the chaining instructions in details. TPC_{cur} represents the address of the indirect jump instruction itself in the code cache. LEN_{JITTB} represents the length/bytes of the JITTB instruction. Similarly, $LEN_{save\ register\ content}$ represents the total length/bytes of the instructions which are used to save the content of the register in the indirect jump instruction.

1. **Generate "Search_ITTB, reg" instruction at the address of TPC_{cur} (reg points to the register used by the indirect jump instruction)**
2. **Generate "JITTB, $LEN_{JITTB}+LEN_{save register content}+LEN_{stub code}$" instruction**
3. **Generate the instructions that save the content of the register used by the indirect jump, i.e. save the target SPC**
4. **Generate stub code**
5. **Generate "JMP REG_{TPC}" instruction**
6. **Return**

Fig. 4. Insert chaining instructions for indirect jump instruction.

Fig. 5 (a) shows the status of the source code block ends with an indirect jump. Fig. 5(b) shows the status of the corresponding translated code block. It shows that the translated code block has been inserted with the custom chaining instructions. The source indirect jump instruction has not been translated to a local indirect jump in the translated code.
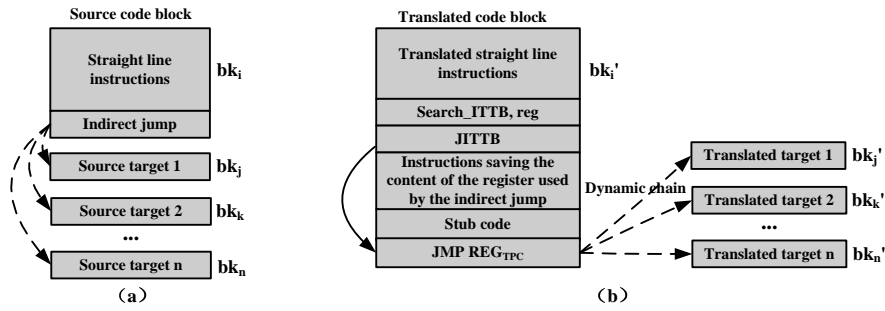

Fig. 5. The status of the source code block ends with an indirect jump and the status of the corresponding translated code block.

In this paper, we only introduce register indirect control transfer instructions, for memory indirect control transfer instructions, the chaining approach is the similar. The difference is that the content of the memory element should be first moved into a register so as to execute the "Search_ITTB, reg" instruction. Thus, this paper focuses on the chaining of the register indirect control transfer instructions. Moreover, the source target address may need to be calculated by adding the value of the register/memory with a certain offset. In this case, we only need to calculate the source target address by following the source target address calculation rule and then write the source target address into a certain register and then execute the "Search_ITTB, reg" instruction.

Stub code is used when the chaining of the indirect control transfer instruction is failed. The stub code saves the execution context and turns the control back to VMM that the corresponding VMM routine looks up the AMT according to the value hold in

the register of the indirect control transfer instruction. If it is hit in AMT, write the related SPC and TPC into ITTB. This is based on the fact that ITTB is a copy of AMT, as the capacity of ITTB is limited, there may be some indirect control transfer instructions whose SPC and TPC have not been saved in ITTB. Hence, if it is hit in AMT, the SPC and TPC of the current indirect control transfer instruction should be written into ITTB.

Other indirect control transfer instructions include indirect branch, indirect subroutine call, and return. Similar operations can be performed on these instructions to insert chaining instructions.

4.2   Executing Chaining Instructions

Though the chaining instructions are inserted into the translated code block, the chaining of the indirect control transfer instruction is finally finished during execution of the translated code block.

The "Search_ ITTB, reg" instruction looks up ITTB according to the content of the register (SPC) of the indirect control transfer instruction, in order to quickly check whether the required TPC is already existed. If the TPC is existed, which means the corresponding TPC has been automatically written into RRG_{TPC}, the "JMP REG_{TPC}" instruction will be executed so as to transfer the control to the target TPC. If the required TPC is not existed, stub code will be executed that the control will be transferred to VMM so as to lookup AMT or perform the interpretation or translation.

Obviously, the novel chaining approach we proposed is a dynamically chaining method based on ITTB.

5   Evaluation

5.1   Evaluation Environment

The experimental infrastructure is based on a Code Cache Simulator (CCS) [2]. We construct CCS to evaluate the control transfer instruction chaining performance. CCS is a two-stage simulator which consists of a profiler, a trace constructor, a code cache and a control unit. In the first stage, CCS runs the source code and the profiler collects the execution information. The trace constructor reorganizes the frequently executing source code into traces and stores the reorganized code blocks in the code cache. In the second stage, CCS runs the code blocks in the code cache instead of the source code while executing the hotspots. Detailed information of the simulator can be found in [2].

The benchmarks (gzip, gap, parser, vortex) we use are selected from SPEC INT 2000 [13]. In [2], we have demonstrated that for these benchmarks, a small number of code blocks may cause huge number of control transfers as hot code blocks are frequently executed. Obviously, without chaining, control transfers are costly because of the large number of AMT looking up operations and context switching overhead.

5.2 Performance Evaluation of IDCTC

In the experiments, we first let CCS run the benchmarks without chaining, then use IDCTC to chain the code blocks which contain the indirect control transfer instructions.

Chaining efficiency of IDCTC

Indirect control transfer instructions realize fast dynamic chaining through accessing ITTB. The target code block may not have been translated when the indirect control transfer instruction is translated the first time and some of the target address is untranslatable instructions [14], this will cause the IDCTC failed. But, the capacity limit of ITTB is the main reason which causes the failure of the chaining of the indirect control transfer instruction. However, because of the principle of locality [12], most of the time, when indirect control transfer instruction accesses ITTB, the required information can be found in ITTB. Fig. 6 shows the chaining success rate of different benchmarks when using direct mapping ITTB with 64 elements, 128 elements and 256 elements. For the ITTB which contains 256 elements, the average success chaining rate achieves 92.5%. Obviously, the hardware ITTB can dynamically assists to chain the indirect control transfer instructions high efficiently. As a result, the chaining efficiency indicates that most of the table lookup operation and context switching overhead can be avoided.
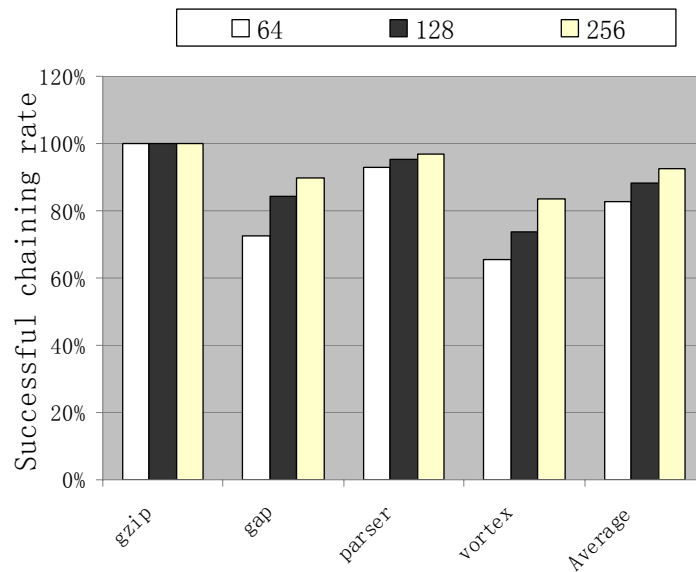

Fig. 6. Successful chaining rate of indirect control transfer instructions.

Fig. 6 indicates that the successful chaining rate is sensitive to the size of the ITTB. For gap and vortex, the chaining rate is lower, because the locality of the target address of the indirect control transfer instruction is not so centralized in these two benchmarks. However, the increase of the ITTB size can fit the address changing character of these benchmarks.

Though, there are still some instructions can not be chained, they only occupy quite a few portion of all the (dynamic) indirect control transfer instructions.

Performance improvement of the code cache

Because most of the indirect control transfer instructions can be chained, the performance of the code cache could be improved. Fig. 7 shows the performance improvement of each benchmark. The y-axis is the performance speedup caused by IDCTC compared to the original execution without chaining.
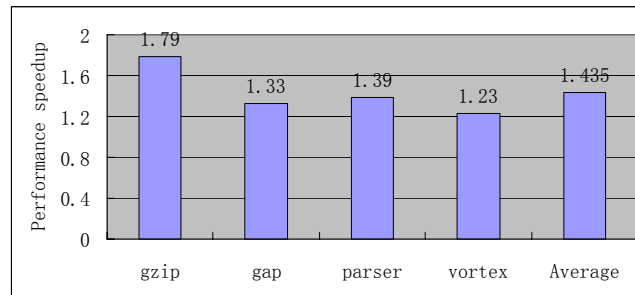

Fig. 7. Performance speedup caused by IDCTC.

The average speed up is 1.435. This demonstrates the efficiency of our indirect control transfer instruction chaining approach. The performance improvement is determined by the chaining efficiency. As a result, IDCTC performs most powerfully for gzip, but still shows dramatic improvement for other benchmarks.

6   Conclusions

Code Cache systems are the key element of the binary translation system and the optimization system. Translated/optimized codes are saved in the code cache in terms of code blocks which ends with control transfer instructions. Conventional code cache systems look up the address mapping table to find the translated target code block according to the source target address. The table lookup operation is considerably costly. Chaining the control transfer instruction with their translated target address can avoided most of the table lookup overhead. The target address of the indirect control transfer instructions are saved in the register or memory and can be changed during execution. It is difficult to chain the indirect control transfer instructions through pure software approaches. In this paper, we propose an indirect transfer target buffer which is implemented as a high speed buffer on chip. ITTB is used to store the frequently accessed target address of the control transfer instructions. Based on ITTB, we propose a novel indirect control transfer instruction chaining method which inserts custom chaining instructions into the translated code block instead of translating the indirect control transfer instructions. During execution, the chaining instructions access ITTB to find the required TPC of the target address. Because of the principle of locality, most of the required target of the indirect control transfer instructions can be found in ITTB. IDCTC is a dynamic chaining approach, in which chaining is

performed during execution instead of translation. Evaluation of IDCTC is conducted on a code cache simulator. The experiment results show that IDCTC can bring dramatic performance improvement for the code cache system.

In the future work, we will use CCS to run more benchmarks to further evaluate the efficiency of IDCTC. Further more, in the future work, we will try to merge the chaining approaches of both direct control transfer instructions and indirect control transfer instructions so as to further improve the performance of the code cache systems as well as those binary translation systems and optimization systems.

References

1. Kim Hazelwood Cettei: Code Cache Management in Dynamic Optimization Systems. Phd. Thesis. Harvard University, Cambridge, Massachusetts (2004)
2. Wexia Xu, Wei Chen, Qiang Dou: A Novel Chaining Approach for Direct Control Transfer Instructions. In: the IEEE International Workshop on Digital Computing Infrastructure and Applications, in conjunction with the 16th International Conference on Parallel and Distributed Systems, pp. 664--669. IEEE Press, New York (2010)
3. J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, J. Mattson: The Transmeta Code MorphingTM Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges. In: 1st Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO'03), pp.15--24. IEEE Press, New York (2003)
4. K. Ebcioglu, E. R. Altman: DAISY: Dynamic Compilation for 100% Architectural Compatibility. In: 24th International Symposium on Computer Architecture (ISCA'97), pp. 26--37. ACM Press, New York (1997)
5. L. Baraz L, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, Y. Zemach: IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems. In: 36th International Symposium on Microarchitecture (MICRO'03), pp.191--204. IEEE Press, Washington, DC (2003)
6. V. Bala, E. Duesterwald, S. Banerjia: Dynamo: A Transparent Dynamic Optimization System. In: the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), pp. 1--12. ACM Press, New York (2000)
7. R. J. Hookway, M. A. Herdeg: Digital FX!32: Combinning Emulation and Binary Translation. In: Digital Techncal Journal, Vol. 9, No. 1, pp. 3--12 (1997)
8. D. Ung, C. Cifuentes: Machine-Adaptable Dynamic Binary Translation. In: the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO'00), pp. 41--51. ACM Press, New York (2000)
9. Kevin Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, M L. Soffa: Retargetable and Reconfigurable Software Dynamic Translation. In: the International Symposium on Code Generation and Optimization, pp. 36--47. IEEE Press, New York (2003)
10. Giuseppe Desoli, Nikolay Mateev, Evelyn Duesterwald, Paolo Faraboschi, Joseph A. Fisher: DELI: A New Run-Time Control Point. In: the 35th International Symposium on Microarchitecture, pp. 257--268. IEEE Press, New York (2002)

11. Ho-Seop Kim: A Co-Designed Virtual Machine for Instruction Level Distributed Processing. Ph.D thesis. USA: University of Wisconsin Madison (2004)
12. Hennessy J L, Patterson D A: Computer Architecture: A Quantitative Approach. Morgan Kaufman Publishers, 3rd edition (2002)
13. SPEC CPU2000, http://www.spec.org/cpu
14. Wei Chen: Research on Dynamic Binary Translation Based Co-Designed Virtual Machine. Ph.D thesis. National University of Defense Technology (2010)