2009 15th International Conference on Parallel and Distributed Systems

A Light-weight Code Cache Design for Dynamic Binary Translation

Wei Chen, Li Shen, Hongyi Lu, Zhiying Wang, Nong Xiao
School of Computer, National University of Defense Technology,
Changsha 410073, Hunan, China
e-mail:{chenwei, lishen, hylu, zywang}@nudt.edu.cn

Abstract—**Interpretation and basic block translation (BBT) are two typical strategies for cold code emulation in a dynamic binary translation (DBT) system. More and more DBT systems employ BBT as the generated native code runs more efficiently than the interpretation routines. We observe that BBT's high efficiency is based on those special hardware assists. With certain simple hardware techniques, interpretation could outperform BBT. In our pervious work, we proposed a hardware interpreted code cache (Pcache) mechanism to speedup interpretation by saving the decoded instruction information during interpretation. This light-weight code cache design could be extended to assist the hotspots translation, thus further reduce the DBT systems' overhead. We add the translation routine entry into the Pcache design thus saving most decoding operations during translation. We use eight SPEC 2000 integer benchmarks on our DBT simulator. Results show that the modified Pcache design causes a speedup of 1.94 according to the referenced DBT with basic interpretation. Furthermore, the interpretation based DBT system assisted by the modified Pcache performs more efficiently than the DBT system which employs BBT for the cold code.**

Keywords-interpretation; basic block translation; decoding; Pcache; performance

I. INTRODUCTION

Dynamic binary translation (DBT) [1] converts codes written for a source instruction set architecture (ISA) into optimized codes for a target ISA, allowing great flexibility for realizing microarchitecture innovations. A number of dynamic binary translation systems have been proposed and developed in recent years.

Because the translation is done dynamically, it usually costs a software-implemented DBT thousands of instructions to translate and optimize a source ISA instruction [2], most sophisticated DBT systems adopt the two-stage translation strategy. Some of them use simple fast interpretation for cold code (not frequently executed codes) emulation and translate/optimize a hotspot/trace (frequently executed code segments). Some use simple basic block translation (BBT) for the cold code emulation. Because the target codes generated by BBT could get native execution which is more efficient than the interpretation routine, more and more DBT systems adopt BBT for the cold code.

Actually, both interpretation and BBT have advantages and disadvantages. The biggest disadvantage of interpretation is that no target binary codes generated for reuse. But it is easy to do the profiling (collecting program execution information) during interpretation as the interpreter and the interpretation routine have the same execution context. The biggest advantage of BBT is the generated target codes. As each basic block ends with a control transfer instruction, chaining should be used to avoid costly context saving and recovering as the basic block execution and lookup table accessing (checking whether the required next block has been translated) have different execution context. Thus, it is difficult to do the profiling when the basic blocks are chained together because profiling is difficult during native execution. For this reason, special hardware assists for profiling should be adopted.

Compared the two cold code emulation approaches, we observe that with certain hardware assists, interpretation may overcome its disadvantage. In our previous work, we proposed a hardware light-weight code cache design Pcache [13, 14] for saving the decoded instruction information during interpretation. We have demonstrated that Pcache can avoid most redecoding operations thus speeding up the interpretation stage.

In fact, Pcache could be extended to assist translation. In this paper, we proposed a modified Pcache design which provides decoded instructions for translation. So, the translation procedure could be accelerated. We run 8 SPEC 2000 integer benchmarks on a DBT system simulator, where the source ISA is IA-32 and the target ISA is a simple VLIW (very long instruction word). Experimental results show that the entire interpretation based DBT system could be accelerated and outperforms the BBT based DBT system.

The rest of the paper is organized as follows. Section 2 introduces some related works. Section 3 summarizes interpretation and BBT first, and then compares the two cold code emulation approaches. Section 4 first briefly reviews the Pcache strategy and then proposes a modified version of Pcache which would assist the translation. Evaluation of the modified Pcache is presented in section 5. Evaluation of the entire DBT system is also presented in this section. Section 6 concludes the paper.

II. RELATED WORKS

Typically, the overhead of an optimized translation for each source ISA instruction is on the order of thousands of instructions. For example, DAISY [3] is reported to take more than four thousand operations to translate and optimize

1521-9097/09 $26.00 © 2009 IEEE
DOI 10.1109/ICPADS.2009.134

one PowerPC instruction for its VLIW engine. The translation of each Alpha instruction to a proposed superscalar-like ILDP ISA takes more than one thousand Alpha instructions [4]. Hence, DBT systems usually employ staged approaches to ISA emulation that simple emulate the cold code and translated/optimized the hotspots of the program. During the cold code emulation stage, the simply basic block translation could be first used, where code is translated based on the basic block without any optimization and is saved in a translated code cache for reuse. The Intel IA-32 EL [5] adopts this approach. [6] also use BBT for the cold code. An alternative to BBT is simple fast interpretation. A part of DBT systems interpret the cold code at the initial stage and translate/optimize the hotspot. DAISY interprets PowerPC instructions before invoking "tree-region" translation [3]. Transmeta Crusoe [7] uses a staged translation strategy combining interpretation and hotspot translation. In our research, we also employ fast interpretation and trace translation/optimization.

III. COLD CODE EMULATION APPROACHES

In this section, we summarize and compare the two cold code emulation approaches, i.e., interpretation and BBT.

A. Interpretation

A software interpreter is a program that reads instructions of the old architecture one at a time, performing each operation in turn on a software maintained version of the old architecture's state [8]. A typical interpreter is always organized as a decode-and-dispatch interpreter, as it is structured around a central loop that decodes and then dispatches it to an interpretation routine based on the type of instruction [9]. In the interpretation routine, one or several target instructions may be used to achieve the same effect as the source instruction. The interpreter and the interpretation routine have the same execution mode/context. Profiling could be performed on each interpreted source instruction. Fig. 1 shows an interpretation routine of a free software interpreter Bochs-2.4.1[10]. This interpretation routine could emulate 32-bit IA-32 add instructions whose operands are general purposes register eax and immediate value, like "ADD eax, $0x4".

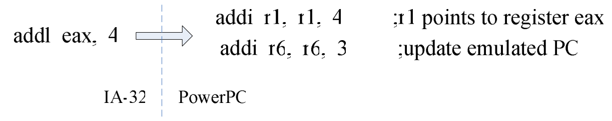
```
void BX_CPP_AttrRegparmN(1) BX_CPU_C::ADC_EAXId(bxInstruction_c *i)
{
    bx_bool temp_CF = getB_CF()
    Bit32u op1_32, op2_32 = i->Id(), sum_32

    op1_32 = EAX
    sum_32 = op1_32 + op2_32 + temp_CF
    RAX = sum_32

    SET_FLAGS_OSZAPC_32(op1_32,op2_32,sum_32,
    BX_LF_INSTR_ADD_ADC32(temp_CF))
}
```

Figure 1. An example of interpretation routine.

B. Basic Block Translation

Basic block translation (BBT) is another popular strategy for the cold code emulation. Source binary codes are recognized and translated to the target codes as their basic blocks. BBT performs simple straightforward translation without any optimizations. The generated binary codes will be saved in a translated code cache in the main memory for reuse.

Fig. 2 shows an example of binary translation from an IA-32 binary to PowerPC binaries in the form of assembly language.



```
addl eax, 4   ==>   addi r1, r1, 4    ;r1 points to register eax
                    addi r6, r6, 3    ;update emulated PC
```

Figure 2. An example of binary translation.

For BBT, each block ends with a control transfer instruction, like a direct jump. The DBT will maintain a lookup table which records the correspondence between the source basic block and the target one. Each time the target block finish the execution, the DBT will search the lookup table to check if the next block has been translated yet. This will cause considerable overhead when control is transferred between target blocks and the DBT software, as they have different execution context. When control is transferred from translated target block to the DBT, processor's special function registers like the status register has to be saved first and will have to be recovered when control is transferred back to the translated code execution mode. One commonly used optimization is to chain the basic blocks together so that immediately jump/branch can be performed at the end of one block to the next. For direct jump instruction, it is easy to chain by modifying the source target address to the target one. For the indirect jump instruction, more complicated methods have to be taken, like software prediction, JLT (Jump Target-Address Lookup Table) [11] and RAS (Return Address Stack) [11] for return instructions.

C. Interpretation vs. BBT

Though the overhead for translating each source instruction is bigger than interpretation, translation is mostly performed the first time the instruction is executed. The translated codes could get native execution. Thus, more and more DBT systems employ BBT to emulate the cold code.

Actually, the high efficiency of BBT is always achieved at the cost of special hardware assists.

First, in order to avoid the high overhead of control transfer, chaining methods should be employed. The software predication is mainly used for register-indirect jump. Software prediction will expand the size of each basic block and may cause great miss penalty due to mispredictation. A better strategy is to use hardware assists like JLT and RAS, which need additional hardware implementation and need to modify the processor's architecture.

Second, it makes the profiling more difficult if the blocks are chained together, because control is transferred to the native execution. Thus, special hardware assist must be employed to support profiling. For an example, Merten et al. [12] proposed a 4K-entry branch behavior buffer (BBB) located after the instruction-retire-stage to identify dynamic hotspots.

In contrast, as the interpretation does not need to change the execution mode, profiling is easily performed without special hardware assists. Besides, as there is no control transfer during interpretation, context saving and recovering is avoided.

The biggest shortage of interpretation is that no target codes are generated for reuse. This could be compensated through both software and hardware approaches. In our previous work [13, 14], we proposed a novel hardware assist-interpreted code cache (Pcache) for saving the decoded instruction information produced during interpretation.

We believe that with high efficient hardware assist, interpretation may achieve the same performance as BBT, and may even provide better performance than BBT.

IV. A Light-weight Code Cache Design

Pcache is a light-weight code cache design for reducing the interpretation overhead. The details of the simple Pcache design are described in a previous paper [13] where Pcache is used in a cross-ISA DBT system. We also demonstrated its efficiency in a same ISA emulation system [14]. In this section we first summarize the principle of our Pcache strategy and then propose the extended version.

A. *Principle of Pcache*

A typical DBT system combining interpretation and translation consists of five major components, a fast interpreter, a profiler, a translator, a translated code cache and a DBT control center. Each component will introduce different overhead. We have conducted experimental simulations on a DBT system simulator. Results from some SPEC2000 integer benchmarks indicate that interpretation was responsible for 40.84% of the total overhead [13].

For a typical decode-and-dispatch interpreter, the overhead of interpreting a source instruction could be partitioned to overheads of fetching, decoding, dispatching and executing the interpretation routine.

Overhead of fetching the source codes through memory hierarchy is determined by the memory system and are not easy to be changed. Overhead of executing the interpretation routine would always been minimized as the interpretation routines always consists of the most efficient equivalent target instructions. Overhead of decoding a source instruction is always on the order of thousands of clock cycles, specially for those complicated CISC ISAs, like x86. Thus, a feasible way to speedup interpretation is to reduce the decoding overhead.

We have found that most interpreted instructions are reinterpreted many times. Results from some SPEC2000 benchmarks on the simulator shows that instructions interpreted only once is less than 1% of the entire interpreted instructions. Clearly, the reinterpreted instructions would cause redecoding operations, introducing a mass of redundant overhead. This forms the motivation of our previous work to reduce the decoding overhead in interpretation.

Pcache is a special light-weight code cache design for saving the decoded instruction information during interpretation. Each Pcache line contains useful instruction information of interpretation routine entry and the operands' addressing form. With Pcache, the interpreter will first access the Pcache according to the source instruction's address. If the required instruction information is found, an interpretation routine whose entry is in the Pcache could be directly dispatched without decoding the instruction. Decoding is only activated when it is not hit in Pcache.

We have demonstrated that Pcache can dramatically reduce the repeated decoding operations and speedup the interpretation. Actually, it could be easily extended to accelerate the translation procedure and further improve the entire DBT's performance.

B. *Extended Design of Pcache*

The basic Pcache strategy saves the decoded source instruction information only for interpretation. Actually, the translation procedure also has a similar decode and dispatch process. The difference is that it dispatches to a translation routine. Thus, the translation routine entry could also be recorded in the Pcache. Fig. 3 shows the modified Pcache line form.

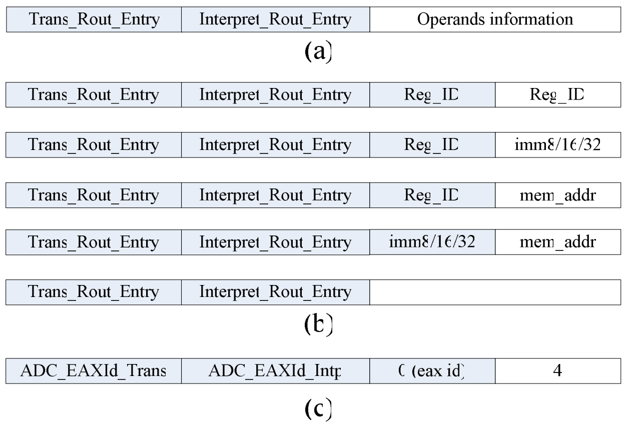


Figure 3. Pcache line format: (a) generic form of Pcache line; (b) Pcache line for IA-32 integer ISA; (c) Pcache line for ADD eax, $0x4.

For IA-32 integer ISA [15] (without IA-32E mode), Pcache lines could be arranged as Fig. 3(b). Each Pcache line contains 16 bytes with 32 bits for each sub-element of the Pcache line. The interpretation function distinguishes the source operand and the destination. Decoded instruction information of ADD eax, 0x4 is shown in Fig. 3(c).

Fig. 4 shows a modified Pcache infrastructure, where PR3 is added for the translation routine entry. PMU (Pcache Management Unit) is the control centre, which convert the source instruction address (Srcpc) and check whether the desired decoded instruction is in Pcache. In our previous work, the hardware Pcache access will affect the zero flag in the status flag register. Actually, this may influence other instructions whose execution depends on the zero flag, like the JNZ instruction. Though we have not found any incorrect cases when using the SPEC 2000 benchmarks, we do consider it is not safe to use the zero flag for the Pcache. It may be better to occupy a new bit for the Pcache in the status flag if there are any reserved bits left. Otherwise, a new status flag could be added into the microprocessor.

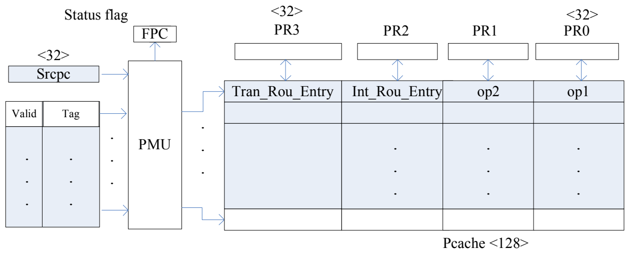


Figure 4. Pcache infrastructure

Conventional cache access is transparent to the user, but access of the Pcache is explicit through several new instructions implemented in the target microprocessor/ISA. Table Ⅰ briefly describes the new instructions. The first two instructions have been already proposed in our previous work, the JPC instruction is added for checking the Pcache status flag FPC.

TABLE I. PCACHE ACCESS INSTRUCTIONS

Instruction	Brief Description
PLD Srcpc	Read Pcache with given source instruction address Srcpc. If hit, send the content of the desired Pcache line into PR0~PR3 and set FPC flag to 1; if not hit, clear FPC flag.
PST Srcpc	Write the content of PR0~PR3 to the Pcache line corresponding to Srcpc.
JPC	Judge the FPC flag to check whether the required instruction is already in Pcache

```
0.Pcahce_Load:
1.PLD Srcpc
2.JPC Decode_Routine
3.JMP [PR2]/[PR3]
;to interpretation/translation routine
```

(a)

```
0.Pcahce_Store:
1.MOV PR0,op1
2.MOV PR1,op2
3.MOV PR2,Interpretation_Routine_Entry
4.MOV PR3,Translation_Routine_Entry
5.PST Srcpc
```

(b)

Figure 5. Code for Pcache access: (a) use PLD to read Pcache; (b) use PST to write Pcache.

Fig. 5 illustrates the kernel codes used by the DBT system to access Pcache both during interpretation and translation. Srcpc is the implementation register holding the source instruction PC value. When reading the Pcache, a source instruction is fetched by a PLD (Pcache load) instruction from the Pcache. If it is hit in Pcache, the content of the corresponding Pcache line will be send to PR0~PR3 and the FPC flag will be set to 1. If not hit, the FPC flag will be cleared. The instruction JPC, check the FPC flag to determine whether the decoding operation could be avoided. Codes in Fig. 5(b) are used to write the decoded instruction into the hardware Pcache. Related information is first MOV to PR0~PR3. Then, PST instruction is performed to write the content in PR0~PR3 into the hardware Pcache.

A modified DBT system assisted by Pcache is shown in Fig. 6. Decoded source instructions are fetched from Pcache first. If it is hit, the interpreter/translator directly executes the related interpretation/translation routine using information in the related Pcache line. If not hit, the interpreter/translator fetches and decodes source instructions from data cache, as it normally does. Decoded instruction is finally written back to Pcache.

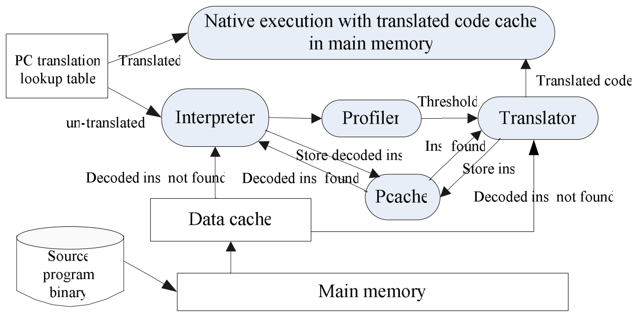


Figure 6. DBT system assisted by Pcache

V. EVALUATION

A. *Evaluation Environment*

The experimental infrastructure is based on a DBT simulator we construct to evaluate the related DBT performance. The DBT simulator is a two-stage DBT system which combines fast interpretation and trace translation/ optimization. The hot trace formation method is a slightly modified version of Cao's adaptive trace/hotspots detection and construction technique [16] with a threshold of 16. The source ISA is IA-32 integer, and the target is a simple VLIW simulator we construct from SimpleScalar 3.0 [17]. Other primary components are shown in Fig. 7 and only the main control/data flow is depicted.

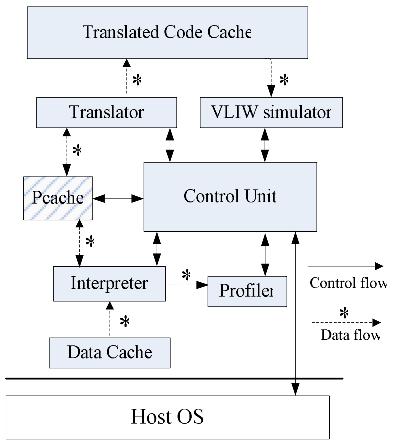


Figure 7. Infrastructure of DBT simulator

The simulator supports two working modes, with Pcache or not, controlled by a controller. Related configuration settings are provided in Table Ⅱ. Eight benchmarks (illustrated in Fig. 8, Fig. 9) of SPEC2000 integer benchmark set have been successfully executed on the DBT simulator. Other SPEC2000 benchmarks contain some complicate floating point instructions we can not handle at the moment as only IA-32 integer instructions will be translated.

TABLE II. MACHINE CONFIGURATIONS AND SIMULATOR SETTINGS

Execution Engine	
Host OS	Linux Red Hat Enterprise3, 2.4.21-4.EL
GCC	3.2.3
Optimization	-O2 –msoft-float
Simulator Settings	
Data Cache	64KB,64B lines,4-way,LRU
Source/target ISA	IA-32 Integer/VLIW
Trace Threshold	16
Pcache	8KB/16KB, 1/2/4-way, LRU

B. Performance Evaluation of Pcache

To measure the performance of Pcache, we first evaluate the Pcache hit rate for several cache sizes and associativities with the least-recently used (LRU) replacement. Fig. 8 shows the experimental results. With the assist of Pcache, DBT has to access the Pcache first for interpreting each source instruction or for translating the source instruction if it has not been translated, decoding only occurs when there is a Pcache miss. Thus, the reduced decoding operation ratio is equal to the Pcache hit rate.

Because these benchmarks have different program characterizations, the hit rate ranges from 90.86% to 99.72%. We have evaluated the Pcache hit rate during interpretation, which are from 90.80% to 99.69% [13]. Results show that more decoding operations have been omitted as Pcache is used for the translation. Among the 6 Pcache configurations, the average Pcache hit rate reaches the highest 98.73% with a 16KB, 4-way, LRU Pcache.

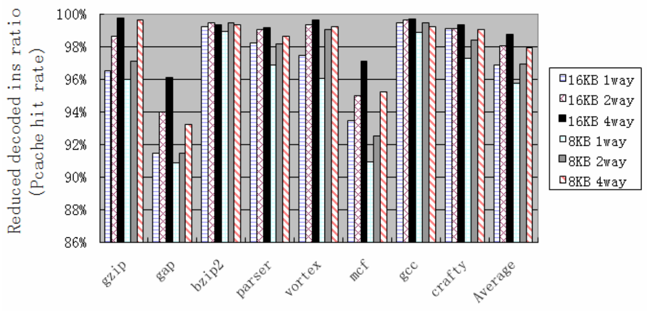


Figure 8. Reduced decoding ratio/Pcache hit rate

C. Performance Evaluation of DBT

Obviously, Pcache can significantly reduce the number of redecoding operations, thus improve the performance of the entire DBT system.

In our DBT simulator, a normal interpretation for a simple IA-32 instruction costs about 2700 CPU cycles on average, about 2500 cycles for decoding a simple IA-32 instruction and 150 cycles for executing the interpretation routine. The average overhead of translating a simple IA-32 instruction is about 47055 cycles, consists of overheads of profiling, source instruction decoding, trace constructing, target codes generating and optimization. As the translator will access Pcache first, most decoding overhead of translation could be omitted.

In our previous work, we have already observed that a 16KB, 4-way, LRU hardware Pcache has a best average speedup of 17.72 for the interpreter. In this paper, we would evaluate the performance improvement of the entire DBT.

Fig. 9 shows the performance of different DBT systems. Ref. is the referenced DBT system combines interpretation and trace translation without Pcache. DBT-P is the reference system with assist of a 16KB, 4-way, LRU Pcache. DBT-BBT is a DBT system combines BBT and trace translation.

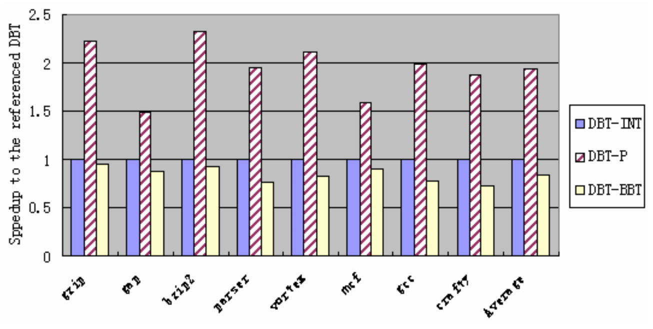


Figure 9. Performance evaluation of DBT systems

Fig. 9 shows the speedup rate of DBT-P to the referenced DBT is 1.94 on average. This indicates that Pcache could significantly speedup the entire DBT system. Fig. 9 also shows the relative performance of DBT-BBT. Here, we do not adopt any special hardware assists for profiling and chaining. Results show that without the special hardware assists, BBT has not outperformed the interpreter and even consumes more overhead.

VI. CONCLUSIONS

Though more and more dynamic binary translation systems employ the basic block translation for the cold code emulation, we believe that with certain hardware assist, interpretation could outperforms the BBT.

Pcache is a light weight code cache design which save the decoded instruction information during interpretation and translation, thus avoid most redundant instruction decoding. We conduct experiments on a DBT simulator. Results show that Pcache could dramatically reduce the DBT system's overhead. Furthermore, Pcache is easy to be implemented than those special hardware techniques that used to assist BBT. With Pcache, the interpretation based DBT may achieve better performance than BBT based DBT.

ACKNOWLEDGMENTS

This work is supported by National Basic Research Program (973) under grant No.2007CB310901, National Nature Science of China under grant No.60803041.

REFERENCES

[1] K. Ebcioglu et al., "Dynamic Binary Translation and Optimization," IEEE Transactions on Computers, vol. 50, No. 6, 2001, pp. 529-548, June 2001.

[2] R. J. Hookway and M. A. Herdeg, "Digital FX!32: Combining Emulation and Binary Translation," Digital Technical Journal, vol. 9, No. 1, 1997.

[3] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," Proc. of 24th Int'l Symp. on Computer Architecture, Denver, USA, pp. 26-37, 1997.

[4] H.-S. Kim and J. E. Smith, "Dynamic Binary Translation for Accumulator-Oriented Architectures," Proc. of 1st Int'l Symp. on Code Generation and Optimization, San Francisco, pp. 25-35, 2003.

[5] L. Baraz, et al., "IA-32 Execution Layer: a two phase dynamic translator designed to support IA-32 applications on Itanium®-based systems," Proc. of the 36th Int' l Symp. on Microarchitecture, pp. 191-204, Dec. 2003.

[6] Shiliang Hu and J. E. Smith, "Reducing Startup Time in Co-designed Virtual Machines," Proc. of 33th Inte'l Symp. on Computer Architecture, Boston, USA, 2006.

[7] A. Klaiber, "The Technology behind Crusoe Processors," Transmeta Technical Brief, 2000.

[8] Richard L Sites, et al., "Binary Translation," Digital Technical Journal, vol. 4, No. 4, 1992, pp. 1-16.

[9] James E. Smith and Ravi Nair, Virtual Machines: Versatile Platforms for Systems and Processer. Publishing House of Electronics industry, pp.29-218, 2006.

[10] K. Lawton, "The BOCHS Open Source IA-32 Emulation Project," http://bochs.sourceforge.net

[11] H.-S. Kim, J. E. Smith, "Hardware Support for Control Transfers in Code Cache," Proc. of the 36th Int' l Symp. on Microarchitecture pp. 253-264, Dec. 2003

[12] M. C. Merten, et al., "An Architectural Framework for Runtime Optimization," IEEE transactions on Computers, Vol. 50, No.6, pp. 567-589, Jun. 2001.

[13] Wei Chen, Hongyi Lu, Li Shen, Zhiying Wang and Nong Xiao, "Using Pcache to Speedup Interpretaion in Dyanamic Binary Translation," Proc. of Inte'l Symp. on Parallel and Distributed Processing with Applications (ISPA 2009) Chengdu, China, August 2009.

[14] Wei Chen, Hongyi Lu, Li Shen, Zhiying Wang and Nong Xiao, "A Hardware Approach for Reducing Interpretaion Overhead," in press of the Proc. of 9th International Conference on Computer and information Technology, Xiamen, China, Oct. 2009.

[15] Intel Corporation, IA-32 Intel Architecture Software Developer's Manual, vol.2: Instruction Set Reference, 2003.

[16] Hongjia Cao, "Research on Dynamic Binary Translation Technology for Microprocessor Design," Ph.D thesis, National University of Defense Technology, 2005.

[17] SimpleScalar 3.0, http://www.simplescalar.com/