2012 IEEE 14th International Conference on High Performance Computing and Communications

Improve Indirect Branch Prediction with Private Cache in Dynamic Binary Translation

Liao Yin[1], Jiang Haitao[1], Sun Guangzhong[1], Jin Guojie[2], Chen Guoliang[1]
1 (School of Computer Science and Technology, University of Science and Technology of China,
Hefei, Anhui 23007, China)
2 (Institute of Computing Technology, Chinese Academy of Sciences, Beijing, 10080, China)
liaoyin@mail.ustc.edu.cn

Abstract—**Dynamic binary translation (DBT) is a just-in-time technology of compiler. It is used to support the binary translation, dynamic optimization, program instrumentation, virtualization and so on. How to improve performance is the core research issues about dynamic binary translation technology. Many researches show that how to handle the indirect branch instruction has a key impact about the performance of DBT. Some methods are proposed to handle the indirect branch. But there are some limitations for these methods. This paper analyzes the locality of target address of the indirect branch. The experiment indicates that there is a well locality about the distribution of target address. To make use of this feature, we propose a novel algorithm to quickly predict the target address for indirect branch. For a given indirect branch, we add a private buffer to cache its all previous target address. When translator meets an indirect branch, it predicts the target address from the private buffer firstly. This way increases the predication hit rate of the target address for indirect branch and reduce the number of context switching in a DBT system. It also effectively improves the whole performance for the dynamic binary translation technology.**

Keywords-dynamic binary translation; indirect branch; locality; private cache

I. INTRODUCTION

Dynamic Binary translation (DBT) is a just-in-time technology of compiler [1]. This technology can analyze and optimize a program at running time. It is widely applied to port legacy code, analyze the behaviors of program, develop new architecture of CPU and support virtualization for clouding computing in heterogeneous platform. There are many dynamic binary translation system, such as FX!32 [2], QEMU [3] and Pin [4]. But the performance of dynamic binary translation is so low that it is not applied widely in practice. The average running time of application program in DBT is 10 times the time of the native program [5]. One of the most important research issues about DBT is that how to improve performance of this technology.

As noted by many researchers, a significant overhead of the dynamic binary translation is caused by indirect instruction handling [6, 7]. The dynamic binary translation technology is based on a basic block of program to work. A basic block ends with a branch instruction is a translation and execution unit for DBT. Usually, there is a branch instruction in average 4 to 7 instructions for a common program [8]. The dynamic binary translator translates a basic block into native code one by one instruction until it encounters a branch instruction. Then the native code is stored into a large code cache. Finally, control flow of program is transferred to the code cache address and executes the native code. Because the storage of basic blocks is not consecutive in the code cache, the control flow must jump from one block to another one. Target address of basic block is decided by the branch instruction. Usually, there are two types of branch: direct branch and indirect branch. The target address of direct branch is statically decided at compiling time. It will not be changed in the program running time. But the indirect branch is decided at program running time. Its target address may vary greatly each time so that it is difficult for a prediction. If the translator meets a branch, it generally breaks control flow to switch context to look for target address in lookup module. Obviously, this way will destroy the locality of DBT system and lead to a significant slowdown of the performance. It is simple and intuitive for link these basic blocks statically according to jump relationship. As a result, one basic block can jump directly to another basic block without lookup process. Unfortunately, only the direct branch can be made a link statically in DBT. Because the target address of indirect branch is decided at running time and varies each time, it can not be linked statically. The DBT system has to look for target address in lookup module when it encounters an indirect branch. The processing of indirect branch leads to a performance penalties. Many algorithms are proposed to handle the indirect branch instruction, including software method and hardware method.

This paper analyzes the existing algorithms of direct branch and indirect branch processing. It also analyzes the distribution target addresses for indirect branch. The result indicates that target address has a well locality for prediction in DBT. We propose a novel algorithm to deal with the indirect branch instruction with this feature. For a give indirect branch, we add a private buffer to buffer its previous target address. When an indirect branch is executed, it predicts target address in its private buffer first. If the private buffer is larger enough, the prediction hit rate will be high and finally form a similar link of the direct branch. This algorithm can reduce context switch for target address lookup. It will improve the whole performance for correlative dynamic binary translation technology.

978-0-7695-4749-7/12 $26.00 © 2012 IEEE
DOI 10.1109/HPCC.2012.45

The rest of this paper is organized as follows. Section 2 gives an overview of the dynamic binary translation. Section 3 discusses the existing algorithms on how to handle direct branch and indirect branch. Section 4 analyzes the distribution of target addresses for indirect branch and evaluates its locality. Section 5 describes our algorithm with private buffer to handle indirect branch handing. Section 6 evaluates the performance of a DBT system with our algorithm. Section 7 concludes this paper.

II. An Overview of Dynamic Binary Translation

Dynamic binary translator can control and manipulate the application program at running time without source code. Generally, the DBT system consists of three modules: translation module, lookup module and execution module. The translation module and lookup module is a front end of the DBT, and the execution module is a back end. The whole flow chart is illustrated in the Fig. 1. Firstly, a source binary program, the front-end code, is input into translation module. The translation module decodes the binary program and translates it into native code that is called back-end code. The native code is stored in the code cache. The code cache is made up of the native code of the basic block unit. Secondly, the control flow is transferred to execute the corresponding native code in the code cache. Thirdly, when the control flow meets a branch instruction in the back-end code, it has to be transferred to lookup module. The lookup module looks for next basic block to be executed through the target address of branch. Because the native code is stored in the code cache, the address space of front-end code is different from the back-end code. There is a mapping of address space between source program counter (SPC) and target program counter (TPC). The relationship of the mapping is maintained usually by a hash table. The lookup module looks for the target basic block in the hash table through the target address (SPC) of branch instruction.

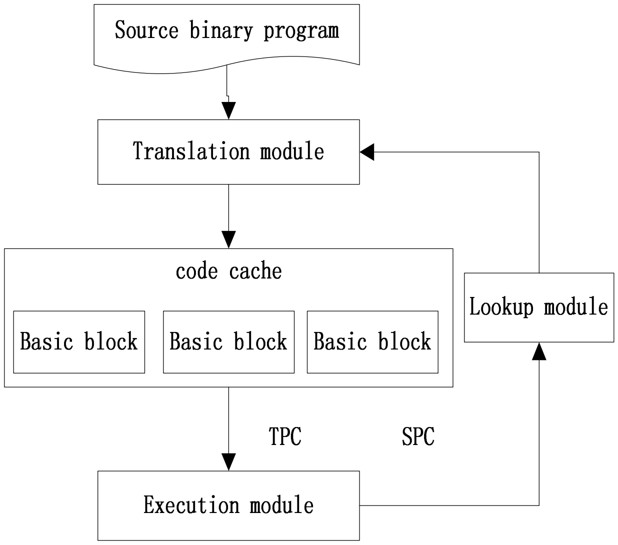

Fig .1. The flow chart of the dynamic binary translation

The branch instruction must be translated specially into a control transfer code (CTC) in a basic block. The control transfer code decided that the control flow is transferred to the lookup module or the next basic block. If the control flow returns back the lookup module, it has to switch context with state saving and recovering. There are a large number of branch instructions in a common program. The context switch will cause a slowdown of the performance of the DBT. How to handle the branch, including direct branch and indirect branch, is key factor of the performance for dynamic binary translation technology.

III. Branch Processing

A. Direct branch Processing

Direct branch includes conditional jump instruction and direct jump instruction. These instructions only contains up to two jump targets. The target address of the direct branch is determined statically at the compile time and is not changed at running time of the program. So it is easy to make a link among the direct branches. When the CTC of the direct branch is executed at first time, it still returns back the lookup module for looking for next basic block. Then the jump target of the CTC is replaced by the address of the next basic block. If the CTC of the same direct branch is executed again, the control flow will be transferred to the next basic block directly rather than the lookup module. At last, this process will form a jump chain among all direct branches. The algorithm is shown in the Fig. 2. This way almost eliminates the context switching of direct branch for target lookup except the first time execution. It executes the back-code as far as possible without the interruption of the control flow. It is a very effectively way for branch processing.

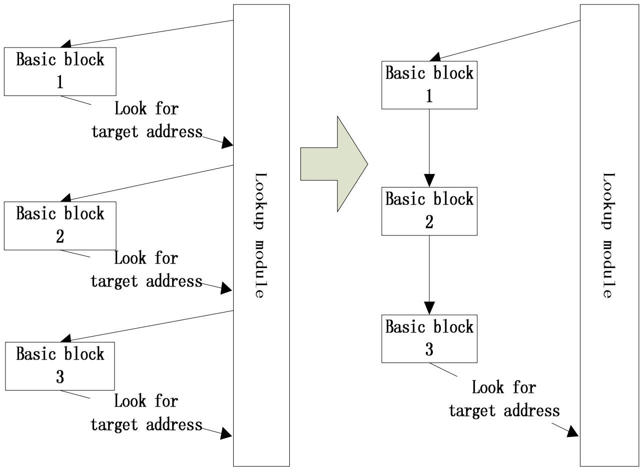

Fig .2. How to make a link for direct branch

B. Indirect branch Processing

Usually, the target address of the indirect branch is stored in a register or a memory location. Their values are determined at program running time and may be changed in each execution. The method to make a link is not feasible for indirect branch. It is a simple and intuitive method to transfer the control flow to the lookup module. Then it consults the lookup module to look for the next basic block by target address. Then the control flow return back the back-end code to continue to execute the next basic block. The switch of the control flow will destroy locality of the program and pollute the cache line. It also cause context saving and recovering for two different contexts between the front end and back end. Therefore, a lot of run time is consumed to look for target block of indirect branch. Many algorithms have been proposed to handle the indirect branch. These algorithms have a basic principle that is to predict the target of indirect branch without introducing time-consuming lookup module.

1) Indirect branch hash table: An indirect branch hash talbe is indexed by the source program counter (SPC), with the entries that are corresponding target program counter (TPC) [6]. An entry contains the source binary program address and corresponding code cache address. The translator looks for a target code cache address in the hash table through the target address of indirect address. The Fig .3 shows the pseudo-code for indirect branch table to look for target code cache address prediction.

The first step, the control transfer code, which is at the end of the basic block with indirect branch, returns back the lookup module from the execution module. The context of the execution module is different from the lookup module, so it has to perform a state saving that is to save all registers of the target CPU into the memory. The second step, the index is calculated by the hash function. Then the translator loads the source binary program address from the corresponding table entry according to the index. The third step, it compares the target address of the indirect branch with the source binary program address. If they match, the prediction of the target code cache address is successful. Then the context of the execution module is recovered from memory. The control flow is transferred the code cache address and execute it from the first instruction in the code cache. If they do not match, the prediction is failure. Thus the translation module begins to translate the basic block into the code cache according to the target address of indirect branch. It also updates translation result into the indirect branch hash table, and then executes the code translated.

This way avoids the re-translation of the target of indirect branch and increases the utilization of the code translated. But it still needs to be switched between the execution and lookup module. The number of context switch is not reduced in this method.

2) Target address inlineg: In order to reduce the number of context switch between the lookup module and execution module, some algorithms are to inline the target address in the back-end code [8, 9, 10]. The possible target code cache address is inserted into control transfer code of the indirect branch. Before the first time encounter of the indirect instruction, the inline address is unknown. When an indirect branch is executed, the translator fills the inline address with new target address and code cache address translated. If the same indirect branch is executed again, it compares those inline the address with the current target address in a register or a memory location. If their values are equal, the prediction is a hit. The control flow can be directly transferred to corresponding code address without lookup. If their values are not equal, the target address continues to be compared with another inline address until the last inline code. If none of them is hit, the prediction is failure. The control flow goes to the translation to translate the target address. The pseudo-code for inline target address is shown in the Fig. 4.

```
save_context();

index = calculate_index(branch_target);

/* check if branch target has been translated */
if(barch_target == lookup_hash_table[index]->SPC){
   TPC=lookuptable[index]->TPC;
  recover_context();
  goto TPC;
}
else{
   goto translate_module(barch_target);
   update_hash_table();
}
```

Fig .3. The pseudo-code for indirect branch table to look for target address

If there is a few of types of the target address of indirect branch, it will effectively reduce the number of the context switch between the lookup module and translation module for target address lookup. However, the target addresses of the indirect branch vary greatly each time execution through the sun’s research [11]. The Fig. 5 shows a statistics of the return address types at running time of the SPCE CPU2000 INT benchmark [12]. There may be handreds of the type of target address. It indicates that the prediction of this way will be failure in most case. If it fails, the process will execute all if statement in the back-end code, which introduces a lot of performance overhead.

```
if(branch_target==address1) {
        goto TPC1;
}else if(branch_target==address2) {
        goto TPC2;
}else if(branch_target==address3) {
        goto TPC3;
}else goto lookup_module( ) ;
```

Fig .4. The pseudo-code for for inline target address

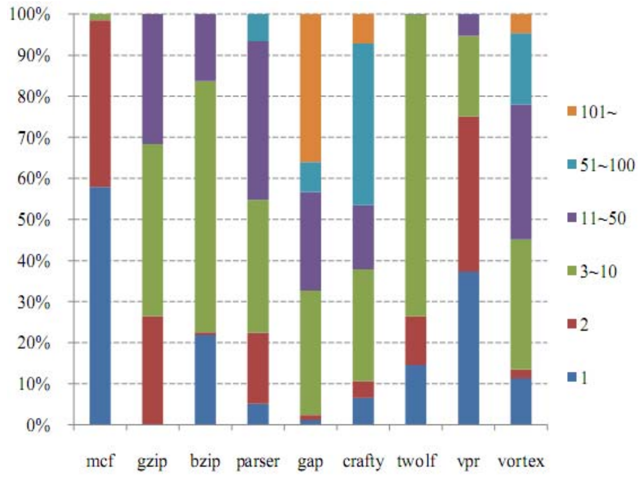

Fig .5. A statistics of the return address types of the benchark[11]

3) Shadow stack: Return instruction is a very important indirect branch instruction in a common program. Shadow stack is proposed to prediction the return address of return instruction in DBT system [2,13]. It is similar with the traditional return address stack (RAS) [14] that is applied widely in the most modern hardware architecture. The RAS maintains call relationship of the function. It can exactly predict the return address of the return instruction. However, this mechanism of the RAS can not be used directly in the DBT technology. The return address is a SPC, but it needs a TPC in dynamic binary translator. In order to use RAS mechanism to predict the return address in a DBT system, the shadow stack is proposed to handle the call and return instruction in FX!32 [2]. FX!32 is a binary translation system that allows x86 binary programs to be executed on Alpha-based systems. It uses the native Alpha stack to hold the x86 return addresses and the corresponding Alpha return addresses when a call instruction is executed. If the translator encouters a return instruction, it will consult the shadow stack to get the native return address. The method is compare the return address in shadow stack with expected the translated return address of current return instruction. If they match, the control flow is transferred directly to execute the corresponding native code cache address stored in the shadow stack. If the mismathc, it only goes to the translation module to translated new code. The Fig. 6 shows the pseudo-code of shadow stack.

If the application does not modify their return address, the shadow stack has a high hit rate with low overhead. But this method need that all call and return instruction is paired. Once there is a mismatch, the shadow stack is invalid for next prediction.

```
push (SPC, TPC);
…………….
pop( SPC, TPC);
if (SPC == return_address)
{
  /* hit */
  goto TPC;
}else
{
  /* miss */
   goto translation_module(return_address);
}
```

Fig .6. The pseudo-code of shadow stack

The above algorithms are proposed to solve the problem that how to predict code cache address of the indirect branch. They can improve the prediction rate and reduce the number of the context switch between the lookup module and execution module. However, there are some limitations in these ways. In the next section, our experiment indicate that there a well locality of target address distribute for an indirect branch. These ways do not make use of this feature. With the locality, this paper proposes a novel algorithm to deal with the indirect branch that is similar with processing of the direct branch.

IV. Locality Analysis

The existence of spatial locality and time locality is common in modern processor architecture [15]. Although the target address of indirect address is computed at run time, our experimental result shows that there is a well locality about the target address distribution of indirect branch.

Our experiment is implemented in the QEMU binary translator [3]. For a given indirect branch instruction, we record a hit rate that target address keeps the same when the same indirect branch is executed consecutively. The front-code is SPEC CPU2000 INT benchmark with the x86 instruction set. The benchmark is translated into the MIPS code through the QEMU translator. The Fig. 7 illustrates the experimental result. There is an average of 50% high hit rate for all indirect branches in the benchmark. Therefore, there is a 50% possible that the current target address is the same as the last execution of the same indirect branch. If the previous target addresses is cached in a larger size of buffer, the hit rate will be higher. It means that the locality of the branch address maybe is enough well for prediction in this way. In order to exploit the locality of the indirect branch, we propose an algorithm that is to use a fit size of private buffer to cache the target address of indirect branch. The high hit rate will make the translator execute the back-code as far as possible without breaking the control flow.

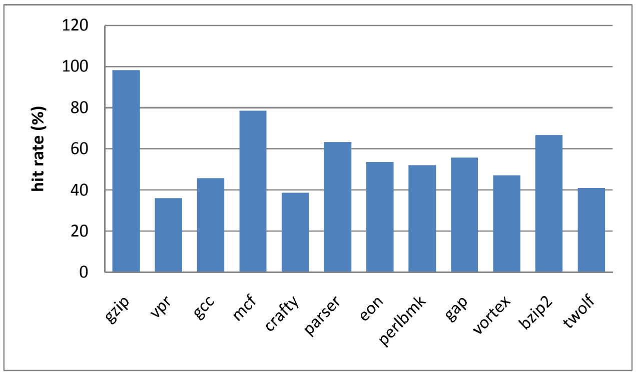

Fig .7. A hit rate that target address keeps the same when the same indirect branch is executed consecutively.

V. The Algorithm with private buffer

Inspired by the experimental results, we add a fit size of private buffer for a given indirect branch. The private buffer is a small hash table, indexed by the source binary instruction address, with the entry contain the previous target Translation Block of the current indirect branch. The Translation Block (TB), which contains the first source binary instruction address of the current basic block and corresponding code cache address, is a data structure to descript the basic block code in the QEMU translator. The Fig. 8(a) shows the flow chart of the algorithm with private buffer to predict the target code cache address.

In the execution module, current TB.1 with an indirect branch is executed at first time. First, it calculates the index through a hash function, which is target address of the indirect branch modulo the size of private buffer. Second, the corresponding entry of the private buffer is loaded from memory, called TB.2. It begins to compare the target address of the current indirect branch with the first source binary instruction address of the TB.2. If they are equal, the prediction is successful. The control flow can jump to the code cache address of TB.2 and continue to execute from the first instruction. If they are not equal, the prediction is failure. The control flow is transferred to the lookup module for looking for next basic block. The pseudo-code of the control transfer code is shown in the Fig. 8(b).

In the lookup module, it can find or translate the next basic block TB.3 through the target address of the indirect branch. It means that the TB.3 is the target translation block of the TB.1. At this time, the translator wants to update the private buffer of the TB.1. The pseudo-code to update the private buffer is in the Fig. 8(c). The process of update is similar with the process of the prediction in the control transfer code in the Fig. 8(b). We get an index that is the result of the first binary instruction address of TB.3 modulo the size of the private buffer. Then the TB.3 is stored into the private buffer of the TB.1 according to the index. If the entry is not empty, it causes a conflict. Our current strategy is simple replace the previous value directly.

An option is to be considered is that how to choose the fit size of the private buffer. The big size buffer will improve the hit rate of indirect branch prediction. The next section verifies this result. The time complexity of the hash lookup algorithm in the private buffer is *O(1)*.

The main overhead of this algorithm is to add a few of instructions in the control transfer code of indirect branch in the Fig. 8(b), and the hash lookup algorithm is so simple in the private buffer. To take advantage of locality, it has a very high hit rate of the prediction. Therefore, this method will reduce the number of the context switch between lookup and execution module.

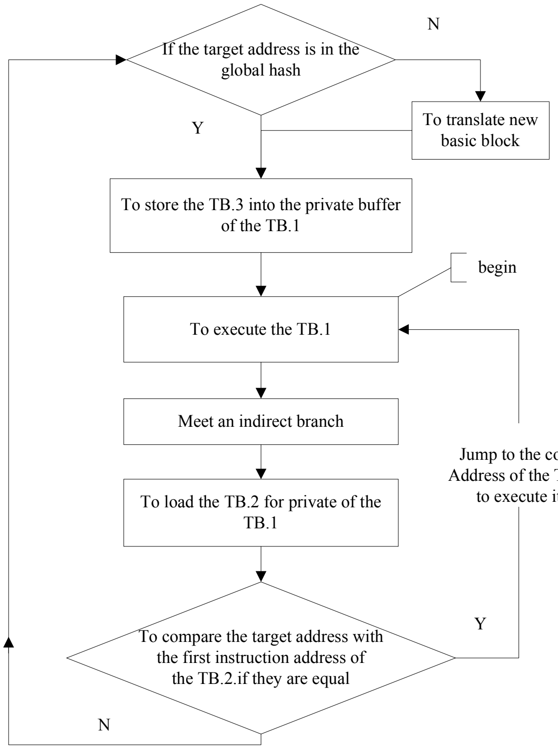
If the target address is in the global hash
N
Y
To translate new basic block
To store the TB.3 into the private buffer of the TB.1
begin
To execute the TB.1
Meet an indirect branch
Jump to the code Address of the TB.2 to execute it
To load the TB.2 for private of the TB.1
Y
To compare the target address with the first instruction address of the TB.2.if they are equal
N

Fig .8(a). The flow chart of the algorithm with private buffer to predict the target code cache address

```
case an indirect branch:
    1, load private buffer of current TB.1;
    2, calculate the index by the target address of the
    indirect branch;

    3, load TB.2 from private buffer  of current TB.1by
    the index;

    4, if (target address == first source binary instruction
    address of the TB.2 ){
            go to the code cache address of TB.2;
    }else{
            go to the lookup module.;
    }
```

Fig .8(b). The pseudo-code of the control transfer code of the indirect branch.

```
#define PRIVATE_BUFFER _MASK \
                            (PRIVATE_BUFFER_SIZE - 1)
/* hash function */
void tb_lookup_hash(TB->pc){
  return (TB->pc >> 2) & PRIVATE_BUFFER _MASK;
}

/* translation and execution loop */
void cpu-exec(){

/* find and translate new TB */
  current_TB = tb_find_fast();

  /* update private buffer of last_TB */
  if (last_TB ends with a indirect branch){
   last_TB ->private_bufer \
  [private_buffer_hash(current_TB->pc)] =current_TB;
  }

/* execute the current TB */
last_TB = tcg_qemu_tb_exec(the code cache address of
the current TB);
}
```

Fig .8(c). The pseudo-code of to update the private buffer in the QEMU system.

VI. Evaluation

In this paper, our experiment is built in the QEMU system. The QEMU is a multi-source and multi-target dynamic translator [3]. The x86 source binary program is input as the front-end code and is translated into MIPS code executed on the Loongson CPU [16]. The prediction hit rate and performance of our algorithm are evaluated in this section.

A. Prediction hit rate of indirect branch

In the last section, it describes that the main overhead of the algorithm is to add several instructions in the control transfer code of indirect instruction. There are different implementations of the control transfer code in different hardware architectures. The indirect instruction is a kind of high-frequency instruction in a common program. If the prediction hit rat is low, the algorithm will add the whole overhead of the DBT system.

With the different size of the private buffer, we measure the prediction hit rate of the algorithm to the SPEC CPU2000 INT benchmark. The result is shown the Fig. 9. With the larger size of the buffer, the hit rate is higher. The reason is that it is use a simple hash lookup algorithm in the private buffer. It gets almost 90% prediction hit rate for all benchmarks with 16 entries buffer. The result proves once again that there is well locality about the target address distribution of the indirect branch.

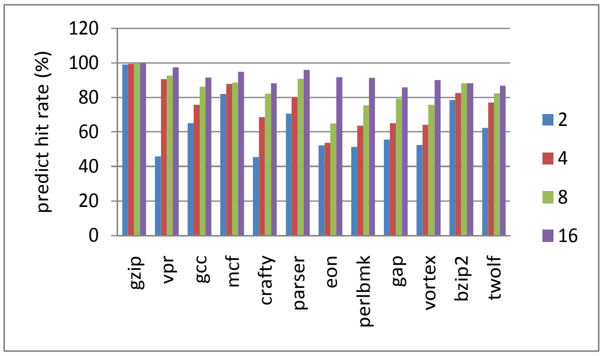

Fig .9. The prediction hit rate of the algorithm with different size of buffer

B. Performanc evaluation

With the high prediction hit rate, this algorithm will significantly reduce the number of the context switch between the lookup and execution module. With the 16 entries buffer, the 80% of the average number of context switching is eliminated for the benchmark in the Fig.10. It also reduces the running time of the x86 benchmarks that is executed on the MIPS platform through the QEMU system. There is an obvious performance improvement for some benchmarks, such as the gzip, mcf and parser. These program has the better locality than other programs.

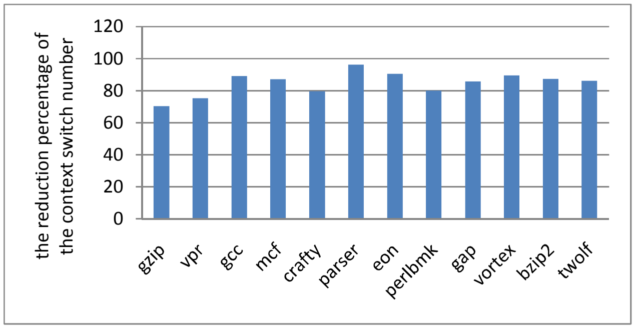

Fig .9. The reduction percentage of the context switch number

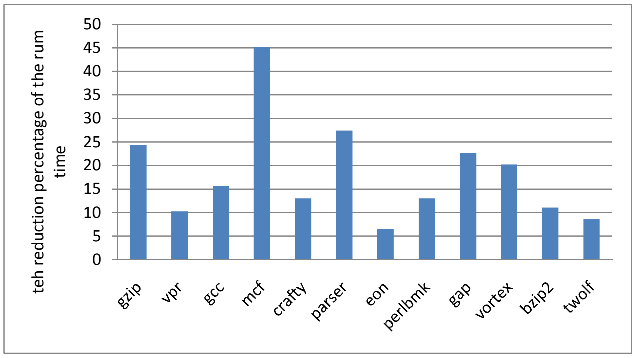

Fig .10. The reduction percentage of the running time

VII. Conclusion

This paper proposed a novel algorithm to handle the indirect branch in dynamic binary translation. It uses a private buffer to predict the code cache address for indirect branch. To take advantage of the locality of the indirect instruction, our algorithm reduces the context switching for DBT system. It is suitable for other similar dynamic binary translation technology, such as dynamic optimization and program dynamic analysis. With low overhead, the experiment result has shown this way can effectively improve the whole performance in dynamic binary translation technology. In the future work, the update strategy will be improved. Only those hot target translation block is added into the private buffer. This strategy will further improve the prediction hit rate with low overhead. The future work also includes dynamic binary parallelization.

Acknowledgment

This work was supported by The National Natural Science Foundation of China (Grant No.61033009).

References

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

[2] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, *"FX!32: A profile directed binary translator"*, IEEE Micro, 18(2), Mar, 1998.

[3] Fabrice Bellard. "*Qemu, a fast and portable dynamic translator,"* In Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track, pages 41–46, 2005.

[4] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney,S. Wallace, V.Reddi, and K. Hazelwood. *"Pin: Building customized program analysisools with dynamic instrumentation,"* In PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design.

[5] Liao Yin, Sun Guangzhong,et al. "*All registers mapping method in dynamic binary translation,"* Computer Applications and Software, 2011.11.

[6] J. D. Hiser, D. Williams, J. Mars, B. R. Childers, W. Hu, and J. W. Davidson. *"Evaluating indirect branch hadling mechanisms in software dynamic translation systems,"* In Intl. Symp. on Code Generation and Optimization, pages61-73, San Jose, California, 2007.

[7] WU Hao LIANG Al-ei LI Xiao-yong, *"Transferring optimize technology in dynamic binary translation,"* Journal of SiChuan University Nature Scienct and Edittion , 2007, 44(6).

[8] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, *"Dynamo: A Transparent Dynamic Optimization System,"* in the Conf. Programming Language Design and Implementation, pp. 1-12, Jun. 2000.

[9] Derek Bruening, Timothy Garnett, Saman Amarasinghe, *"An Infrastructure for Adaptive Dynamic Optimization,"* Int. Symp. Code Generation and Optimization, pp. 265-275, Mar. 2003.

[10] Derek Bruening, Evelyn Duesterwald, Saman Amarasinghe, *"Design and Implementation of a Dynamic Optimization Framework for Windows,"* The 4th Workshop on Feedback-Directed and Dynamic Optimization, Dec. 2001.

[11] Sun Tingtao, Yang Yindong, et al. "*Return Instruction Analysis and Optimization in Dynamic Binary Translation*," Fourth International Conference on Frontier of Computer Science and Technology, 2009. FCST '09

[12] http://www.spec.org/cpu2000/

[13] Michael Gschwind, *"Method and Apparatus for Rapid Return Address Computation in Binary Translation,"* IBM Disclosures YOR819980410, Sep. 1998.

[14] David Kaeli, P. G. Emma, *"Branch History Table Prediction of Moving Target Branches Due to Subroutine Returns,"* The 18th Int. Symp. Computer Architecture, pp. 34-42, Jun. 1991.

[15] D. Patterson and J. Hennessy.*Computer Architecture: A Quantitative Approach*, 4th edition, Morgan Kaufmann, 2006.

[16] W. Hu, J.Wang, X. Gao, Y. Chen, Q. Liu, and G. Li. *"Godson-3: A Scalable Multicore RISC Processor with x86 Emulation,"* IEEE Micro, Vol. 29, No. 2, 2009.