Floating Point Optimization Based on Binary Translation System QEMU

Qiang Shi[1, a], Rongcai Zhao[2, b]

[1] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China

[2] State Key Laboratory of Mathematical Engineering and Advanced Computing, Zhengzhou, 450001, China

[a]email: shiq1006@126.com, [b]email:Rczhao26@126.com

Keywords: Binary Translation; QEMU; Float Point Instruction; Optimization

Abstract. Binary translation system, a technique for translating executable binary program, could transform the code operating on a system structure into the command of another system structure. However, in multi-source translation system, the efficiency in floating point instruction translation realized by relying on software is not high. The floating point computation capacity is the important indicator of computer performance. By taking SW processor as the target platform, translation algorithm for floating point optimization based on QEMU translation system is put forward. It mainly includes the following two stages: firstly, the content of identifying floating point computation invoking helper function, the number obtaining parameter and returned value, data type information; secondly, the floating point operation of changing into local floating point operation and reduce the redundant floating points according to the standard of IEEE. According to the experiment of NPB test set, it manifests that the average speed-up ratio of floating point optimization algorithm can reach 30%.

Introduction

Along with the diversification of computer microprocessor, new-type microprocessor usually can enable it to have high performance ratio with more advanced design concept. However, it is severely restricted due to the application software during promotion. Binary translation [1][2], as the one of research hotspot for solving the problem at the present, can realize the compatibility of new system structure with the application program of the old one by simulating the old one. As a result, the original software designed for certain system structure could be equivalently operated on another system structure without modification. It could well reduce the dependence of application program on base layer hardware so as that the both can be developed and changed relatively dependently [3].

Binary translation can be divided into dynamic, static and dynamic static integrated binary translation [4] according to translation mode. Dynamic binary translation [5] could generate code while executing the target program, but its efficiency is not high for its optimization and execution program occupies translation time. What's more, while executing the generated target codes, dynamic binary translation should finish loading analysis, operating environment setting, real-time translation, code buffer management & code block switch, etc. Though the efficiency of dynamic binary translation system is enhanced through some techniques including hot path optimization, register mapping and multi-thread optimization [6][7], the problem about low efficiency of dynamic binary translation is not solved. Static binary translation statically analyzes executable program, extracts instructions, translates them, and executes after finishing translation. It could generate high-quality code through complicated code analysis and optimization algorithm. It has high execution efficiency. However, the static binary translation cannot solve the problem about code discovery and positioning.

Multi-source translation system uses the intermediate code not relating to the system structure to realize the compatibility of multi-target machine and multi-host machine, but the system structural characteristics of host machine are neglected. However, floating point computation capacity is the

© 2016. The authors - Published by Atlantis Press

important indicator of computer performance. In the field of binary translation, it is necessary to translate the floating point instructions efficiently and correctly out of consideration for function and performance. On the basis of above content, the floating point optimization algorithm for domestic SW platform translation is designed based on dynamic binary translation system QEMU to full utilize the platform characteristics in this paper. Besides, through NPB test set proves that this optimization algorithm can enhance operating efficiency remarkably.

Related Work

This algorithm optimizes the translation of floating point instruction in binary translation system QEMU. QEMU is a dynamic translation system of variable source and target compiled by Fabrice Bellard et al [8]. As shown by the Fig. 1, QEMU conducts translation based on the unit of basic block. Basic block is a program segment only with one entrance and numerous exits. Before translation begins, QEMU will firstly fast search the corresponding translated block of initial program address in cache of translated basic blocks. To enhance searching efficiency, QEMU stores the translated block used recently in hash table. In case of fail to find out the translated block, the translation program will be started to read the code and transform it into the corresponding TCG platform independence intermediate representation and conduct instruction optimization. Then, it will be transformed into the target platform code according to the target platform instruction and also will be switched into execution thread so that the host will execute the program until the basic block is finished [9].

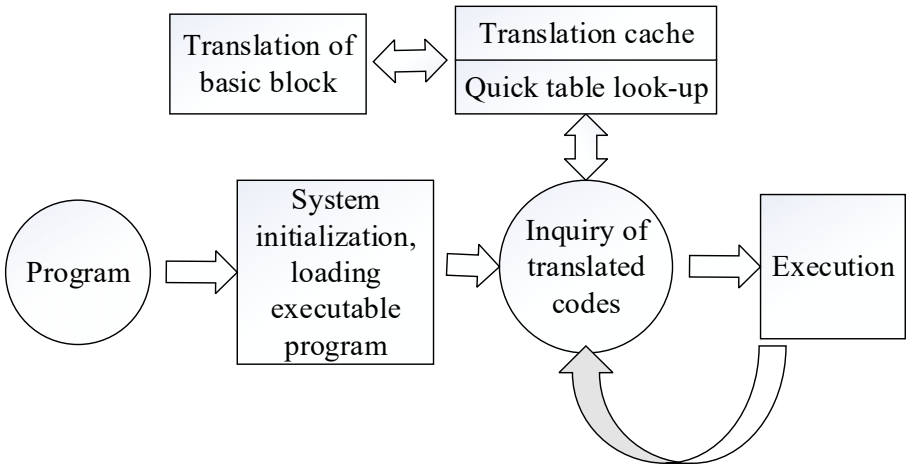

Fig. 1 Work Flow Diagram of QEMU

The support for floating point computation is an important module of dynamic binary translation system. It has great influences on performance of the whole system to deal with floating point computation.

For the difference in architecture characteristics of source and target platform, QEMU translator only simulates these characteristics through software when processing the floating point instructions [10]. That is, each floating point instruction is processed into a function reference of corresponding functions. In this way, for the translation result cannot be utilized repeatedly. When the floating point instruction is executed, a function reference will be generated. This waste a lot of time. Besides, the translation of instruction's code block will also be influenced. When the floating point instruction is translated, the code block will be finished [11]. Then, the function reference corresponding to the floating instruction will be executed once. The code block after floating point instruction will be executed. Thus, the segmentation of code block will be caused and the efficiency will also be inevitably influenced.

According to analysis on the processing of QEMU for floating point instructions, it is time-consuming to simulate all contents about floating computation with the function helper. For instance, the index, extraction of decimal part, overflow judgment and execution operation are simulated with c programming languages [12]. Therefore, the algorithm for simplifying floating

point algorithm according to local floating point operation of SW should be proposed. Firstly, it is necessary to find out floating point processing code in QEMU, revise the analogue function into the local version of SW according to the standard of IEEE and then conduct testing of procedure set.

Formalization and Equivalence of Binary Translation

The equivalence of binary translation and optimization will be expressed and formalized model will be built in this chapter.

For formalized model of binary translation, some important theoretical representations are as below: $M(S,I,\gamma)$ indicates computer, wherein S: indicates state of computer, I: indicates the instruction set of computer, γ: The mapping of $I \times S \to S$, indicating the explanation of certain instruction under certain machine state. Assuming $M\mathrm{s}(S,I,\gamma)$ indicates the source platform computer and $M\mathrm{t}(S,I,\gamma)$ indicates the target platform computer, based on which, binary translation can be represented by: Search a map φ so as that the source platform can explain the instruction I under current state S. There will be corresponding target platform state S' to execute $I' = \varphi(I)$. This map φ is one of core work of binary translation.

Program equivalence transformation indicates the transformation of structure, execution mode, storage mode and programming mode of the program under the premise of keeping the program's functions unchanged designated to enhancing the execution efficiency, safety and transportability of the transformed program, including the transformation from low-level language to low-level language, like the performance optimization of binary translation and math library, from high-level language to low-level language, like typical advanced language encoder, transformation from serial program to parallel program including parallelized transplantation of the high performance software. Binary translation indicates program equivalence transformation of instruction layer. Its related theories are also applicable to binary translation. According to binary translation mode, the definition of equivalence is given to the equivalence of binary translation in terms of instruction and semantic layers.

$$\gamma(S,I) = \gamma'(\phi(S),\varphi(I)) = \gamma'(S',\varphi(I)) \quad (1)$$

Definition 1 instruction level equalization: source platform Ms& target platforms Mt. When meeting the above equation, the mapping and instruction state mapping of binary translation is equivalent.

This definition explains strong equivalence of simulation at instruction level and strong equivalence model. In some binary translators, the instruction-level equivalence requirements could be satisfied if the instruction of source platform are explained one by one. However, such definition of strong equivalence restricts the optimized function layer. It only could optimize each instruction but cannot optimize it according to the basic block or function semantics of the instruction.

Definition 2 semantic-level equivalence: source platform Ms and target platform Mt, for any instruction sequence $t = \{i_1, i_2, i_3 ...\}$, when equation is satisfied, mapping of binary translation instruction and state is equivalent.

$$\gamma(s,t) = \gamma'(\varphi(s),\varphi(t)) = \gamma'(s',t') \quad (2)$$

Semantic-level equivalence requires the final state should correspond to the state obtained by execution after Mt execute instruction sequence $\varphi(t)$ of the target platform of translation. It doesn't require that there should be an intermediate state corresponding to the one for the execution of each instruction so that the optimization based basic & super block and function level could be carried out.

Definition of equivalence, similar to mathematical isomorphism concept, could manifest the conformity of source and target platform with computability.

Realization of Floating Point Optimization

QEMU translation system simulates floating point instructions through software method. Each

floating point instruction of X86 is realized with a corresponding helper c function. One instruction will be changed into a lot after translated. In this way, the efficiency of virtual machine will be inevitably influenced. Such practice is a general one. It has good expandability at the cost of sacrificing performance. To realize floating point optimization algorithm, it is necessary to enhance the efficiency of translation floating point instructions, mainly including the following two stages: firstly, it is about identifying the content of floating point computation helper c function to obtain parameter information; secondly, it is about revising local floating point operation.

To complete computation of floating point instructions, it is necessary to input parameter information into helper c function, including operand, data type of operand and returned value and then transmit to helper c function, and then helper function will extract the decimal part, conduct overflow judgment and execute operation result according to the contracted parameters. In the end, it will write the result into corresponding location of source platform CPU State. The Fig. 2 manifests the example of helper code of invoking float64_add.

```
//Function：Achieve 64-bit floating point of Added functionality
//Import: operand, data type, state parameter
//output: the result of float add

Float float64_add (float64 a,  float64 b  STATUS_PARAM)
{
   Extract the operand index, the decimal, type, and Flag.
   Incoming parameters, calls the addition function.
   Return result.
}
```

Fig.2. Float64_add Code of Floating Point Reference

Obtain floating point operation parameter. As for the original processing mode, there will be lots of code generated due to operations such as judgment of a lot of decimals and decimal extraction. However, it's found that many operations are not useful to the computation of floating point result, but the computation efficiency will be reduced. Therefore, we simplify it into the local version.

```
//Function：Achieve 64-bit floating point of Added functionality
//Import: operand, data type, state parameter
//output: the result of float add

Float float64_add (float64 a,  float64 b  STATUS_PARAM)
{
   Define double precision type variables afloat bfloat and Float64 variable
auint as return.
   The parameters of a and b type casts for double precision.
   Completes the add operation, the result put in afloat.
   afloat cast for float64 type and assignment to auint.
   Return result auint.
}
```

Fig.3. Float64_add Code of Revised Local Version

Experiment and Result

The experiment is aimed at verifying the correctness and efficiency of test set of QEMU translation operation NPB (NAS Parallel Benchmark) through floating point optimization.

Table.1. Binary Translation Environment

Translation environment	Target platform Ms	Host computer Mt
CPU	X86_64	SW64
OS	Fedora2.6.27.5-117.fc10.i686	NeoKylin 3.8.0
Translator	gcc-4.3.2	gcc-4.5.3
QEMU version	QEMU1.7.2	
Test program	NPB program test set	

The performance test methods for optimization algorithm are given out below, and it is compared to QEMU. The performance of CPU, FPU and memory system before and after realization of floating point optimization algorithm through qemu-1.7.2 is tested with NPB. Assuming T_{QEMU} and T_{FQEMU} the time for QEMU to executable program after optimization of QEMU and floating point respectively, and is the speedup ratio of optimized QEMU compared to the original one, define:

$$S_{speedup} = \frac{T_{QEMU} - T_{FQEMU}}{T_{QEMU}} = 1 - \frac{T_{FQEMU}}{T_{QEMU}} \tag{3}$$

NPB test set contains 8 programs, as shown by the following table:

Table.2. Binary Translation Environment

Items	**Test specification**
IS	Basic testing of integer sequence
FT	Benchmark test of quick Fourier transformation
MG	Multi-grid benchmark test
CG	Benchmark test of conjugate gradient
EP	Benchmark test of complex paralleling
LU	Symmetric relaxation method for solving equation set of block sparse
BT	Solve 5×5 tri-diagonal equation set
SP	Solve five-diagonal line equation set

It could be seen from the Fig. 4, NPB test item averagely reaches 30% speedup ratio after optimizing the local algorithm with floating point. For the test items, the programs of floating point computation are frequently adopted, like FT program. This algorithm can greatly enhance the execution efficiency. According to the result of test set, it indicates that the algorithm of items with floating point computation has large speed ratio.

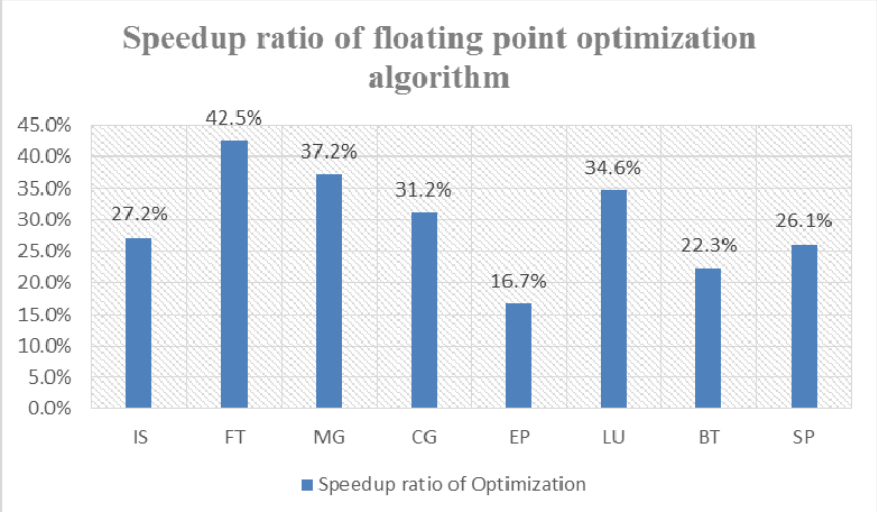

Fig.4. Speedup Ratio of Floating Point Optimization Algorithm QEMU for Items of NPB Test Set

Conclusions

The deficiencies in floating point instruction translation of existing system are firstly analyzed in this paper. Equivalence definition of binary translation as formalized way of expression and translation of program transformation is given out simultaneously. The floating point optimization algorithm for substituting the local floating point operation with original translation processing is put forward. By realizing algorithm in binary translator QEMU system and adopting NPB test set, the speedup efficiency is tested. According to the experiment, it manifests that the algorithm can enhance the efficiency of translation system remarkably for the applications with floating point computation.

References

[1] Erik R Airman, David Knelt and Yaron Sheffer, ”Welcome to the Opportunities of Binary Translation” Computer, Vol 33, No 3, March 2000, IEEE Computer Society Press, PP40-45.

[2] Bellard F. QEMU, a Fast and Portable Dynamic Translator[C]//USENIX Annual Technical Conference, FREENIX Track. 2005: 41-46.

[3] HISER J D, WILLIAMS D, HU W, et al.Evaluating Indirect Branch Handling Mechallisms in Software Dynamic Translation Systems [C]//Proceedings of the international Symposium on Code Generation and 0ptimization, 2007:61-73.

[4] V. Chipounov and G. Candea, “Dynamically Translating x86 to LLVM using QEMU,” Tech. Rep., 2010.

[5] WEI C, ZHIYING, M ZHONG Z, et al. TransARM: An efficient instruction set architecture emulator[J]. Chinese Joumal of Electronics, 2011, 20(1):6-10.

[6] Whelan R, Leek T, Kaeli D. Architecture-Independent Dynamic Information Flow Tracking[C].Proceeding of the 22th international conference on Compiler Construction ,Springer Berlin Heidelberg, 2013:144-163.

[7] Tran Van Dung. Function profiling for embedded software by utilizing QEMU and analyzer tool[C]. Proceedings-the 56th International Midwest Symposium on Circuits and Systems, 2013: 1251 – 1254.

[8] Probst M.. Dynamic binary translation[C] UKUUG Linux Developer’s Conference，2002.

[9] A. Jeffery, “Using the LLVM Compiler Infrastructure for Optimised, Asynchronous Dynamic Translation in Qemu,” University of Adelaide Honors Thesis, 2009.

[10] Shen B Y, You J Y, Yang W, et al. An LLVM-based hybrid binary translation system[C]//Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on. IEEE, 2012: 229-236.

[11] Lyn Y H, Hong D Y, Wu T Y, et al. DBILL: An Efficient and Retargetable Dynamic Binary Instrumentation Framework Using LLVM Backend [C]. Proceeding of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, ACM, 2014:141-152.

[12] Intel Corporation. Intel®64 and IA-32 Architectures Software Developer’s Manual[M]. USA: Intel Corporation, 2008.