International Journal of Hybrid Information Technology
Vol.8, No.2 (2015), pp.199-210
http://dx.doi.org/10.14257/ijhit.2015.8.2.18

Register Allocation for QEMU Dynamic Binary Translation Systems

Yi Liang[1], Yuanhua Shao[2], Guowu Yang[2] and Jinzhao Wu[*3]

[1]*College of Information Science and Engineering, Guangxi University for Nationalities, 530006, China*
[2]*School of Computer Science and Engineering, University of Electronic and Technology of China, Chengdu, 611731, China*
[3]*Guangxi Key Laboratory of Hybrid Computation and IC Design Analysis, Guangxi University for Nationalities, 530006, China*
LiangYi1204@163.com[1], YuanhuaShao2010@163.com, GuowuYang03@126.com, hidrwu@sohu.com
**Corresponding authors: Jinzhao Wu*

Abstract

Binary translation is an important step to solve the code migration, QEMU is more advanced and efficient binary translation system. It uses lighter TCG technology to achieve dynamic binary translation but analysis of the TCG internal process, we found that the excessive use of temporary variables meaningless in the TCG intermediate code, the backend generates host code does not take into account the efficient use of registers. Through these two aspects of improvement, especially increased a linear scan register allocation algorithm in the back-end, can be in an acceptable translation time, generates efficient host code. The experimental results show that the optimized program run time significantly reduced and the amount of generated host code reduced by an average of 8%.

Keywords: *Dynamic binary translation, Linear scan register allocation algorithm, spill process, QEMU*

1. Introduction

With the research and development of new architecture, code migration issues more prominent, and the core binary translation virtual machine technology has gained widespread attention and research. Binary translation technology is a direct translation technology, it can translate executable on a processor to another processor. You can begin software development and debugging before the born of the hardware using embedded hardware emulation whose core technology also is binary translation. Binary translation technology has application value on software cross-platform and hardware emulation.

QEMU (Quick EMUlator) system is more advanced multi-source and multi-objective binary translation system, and it supports both virtual process level and system-level virtual two modes of operation, with high-speed, cross-platform, open-source, easy to transplant, etc. [1, 7]. Process virtual machine allows applications developed for other architectures to use, and interact in the same way with the other programs installed on the current host.; System virtual machine can make other architecture operating system can be installed and run on the user host computer, and can be mounted within the operating system to run the application. Compared advantages and disadvantages of these two modes, process-level simulation is often relatively simple and efficient, especially in the source machine operating system and the host operating system is the same series, the more obvious advantages [1]. This paper

ISSN: 1738-9968 IJHIT
Copyright ⓒ 2015 SERSC

mainly discusses the the QEMU process level binary translation system. TCG (Tiny Code Generator) technology was introduced into QEMU after v0.10.0. There always been a key problem, the allocation of the registers. The number of registers in the hardware machine is small but the operation speed is high. So they are precious resources. The original TCG technology has not considered the using condition of the registers in the source code context, the values were not saved and the using ratio of the registers is low, causing repeatedly access memory while using variables. Register allocation algorithm can solve this problem, and we will implement it at the back-end of TCG.

2. About QEMU Translation System

QEMU is a pure software simulation of the integrated development environment in the common linux and windows platform, it can simulate common embedded computer systems.

The working process of QEMU is similar with the process of traditional compiler. The different point is that the front input of QEMU is executable binary code on some platform while the front input of traditional compiler is some kind of advanced language. QEMU translation process is shown in Figure 1. TCG's role as a real compiler backend, primarily responsible for analyzing, optimizing the object code and generates host code. A real CPU, execute process consists of three parts of the instruction fetch, the translated instructions, the execution instruction. The QEMU simulator processor is the same. Key among these is the translated instruction process, which consists of three parts disassemble, the intermediate code analysis program, dynamic code generator to complete. The front end disassembly Used source binary code disassemble generate intermediate code. The rear end is a combination of the analysis procedures and the dynamic code generator, responsible for the activity analysis of the intermediate code and complete the generation of the host binary code.

3. TCG Translation Problem Analysis

3.1. A Problem in the TCG

Different architecture has a different set of registers, the number of registers that they have is not consistent. Host registers mapped to the target machine registers, it is a very effective way in dynamic binary translation. QEMU First intermediate variables introduced by the system translation mechanism mapped to the host register. This article believes QEMU translation mechanism introduced intermediate variables is not necessary, but affect the translation performance. Do this in a later experiment, we will cancel the process of mapping the variables of the object code to the intermediate variables.

Code expansion is unavoidable in the process of translation, a better translation strategies can reduce the code expansion rate. In the process of the memory variable mapping host register, if you use the right register allocation strategy in acceptable translation time, you can greatly reduce the amount of code generated host code, reduce code expansion coefficient, and eliminates unnecessary memory repeat access operation, and to shorten the execution time of the host binary.

3.2. Choice of Register Allocation Strategy

In any compiler register allocation problem is an important work, it's efficiency problems in Sthe past 20 years has been widely studied. Register allocation is NP complete problem, the most widely used is graph coloring algorithm proposed by Chaitin [2, 8] in 1981. Unfortunately, most aggressive global register allocation algorithms are computationally

 Copyright ⓒ 2015 SERSC

expensive due to their use of the graph coloring framework, in which the interference graph can have a worst-case size that is quadratic in the number of live ranges [3, 9].

Taking into account the TCG technology is code block as a translation unit, and the amount of code for each block of an average of 5-7, we choose a simple linear scan register allocation algorithm which is an optimization technology widely researched and used at the back end of compilers. The linear scanning algorithms to simplify the allocation based on graph coloring problem, consider the coloring of an ordered sequence lifetime. The linear scan algorithm can improve the register allocation speed (linear speed), to produce a good quality within a short time for translation target binary code. The complexity of the algorithm is low and the translation speed is high. The running speed of the final generated host binary code is high so the algorithm could well meets the trade-off relationship between distribution effects and allocation efficiency. So Its improved version is favored by LLVM, Java Hotspot and produced great influence.

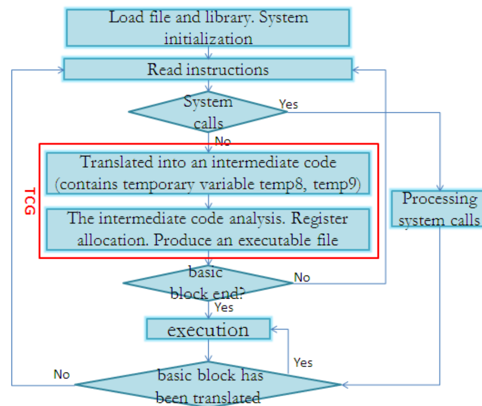


Figure 1. QEMU Translation Process

4. Front End Redundant Code Eliminate

4.1. TCG Redundant Code Analysis

Table I is taken from QEMU log file. Intermediate code representation of the table shows three adjacent instructions.

As can be seen from Table 1, TCG intermediate code generation process, each source binary instruction that will be translated into several tiny operation. It was first mapped the source variable instruction to the temporary variable temp8, temp9, then use a temporary variable for arithmetic operations. Finally, the operating results of temporary variables are mapped to the source variable instruction. Original register variables like r1 and r2 are called global variable while temp8 and temp9 are called temp variables.

QEMU v0.9.0 and before, has not introduced the TCG technology, the basic idea is to cut per a source of instruction divided into several microinstructions, each microinstruction is achieved by a simple piece of C code, and then extract the corresponding target file to generate dynamic code generator the last microinstruction combined into a single function to perform. Operations on uncertain number of global variables are translated into operations on a small number of temporary variables. In such a case, the intermediate code of QEMU these intermediate variables can bring other advantages: can greatly reduce the number of micro-operation [1], reduce the design complexity of C functions.

TCG technology, the use of this temporary variable is not only no longer reflect the advantage, but the intermediate code generation process is more complex, and not conducive

Copyright ⓒ 2015 SERSC

to the optimization of the back-end. And this translation process for a single instruction, so that adjacent no association between instructions, is not conducive to the optimization of the statement. Translating on Single instruction can hardly get efficient optimization, only translating on a hole block with considering the context can effectively analysis code and find optimization space.

4.2. TCG Redundant Code Elimination

Arm instruction r0 ~ r15 as memory variables in the process of translation, so r0 ~ r15 operation does not need to be mapped to the temp temporary variables but direct use of the variable operation.

For some complex instructions, if necessary, still use temp8 temp9 to disassemble multiple micro-ops. Table 2 is optimized by the front-end redundant data in Table 1.

Color figures will be appearing only in online publication. All figures will be black and white graphs in print publication.

Table 2 shows that the reduced intermediate code expression eliminated unnecessary temporary variable mapping, the form of expression is more simple. After simplified the expression of intermediate code, intermediate code involved in computing with the style of global variable. A hole block source code is regarded as an analysis unit, research related code, find the life time of variables. This type of life time is no longer in single source code range, it is in a hole basic block range. So better mapping between variables and registers can be achieved and more effective optimization method can be more conveniently implemented.

Table1. Source Log Information

ARM instruction	TCG intermediate code
add r3,r1,r2	mov_i32 tmp8,r1 mov_i32 tmp9,r2 add_i32 tmp8,tmp8,tmp9 mov_i32 r3,tmp8
add r5,r2,r3	mov_i32 tmp8,r2 mov_i32 tmp9,r3 add_i32 tmp8,tmp8,tmp9 mov_i32 r5,tmp8
add r6,r1,#12	mov_i32 tmp8,r1 mov_i32 tmp9, $0xc add_i32 tmp8,tmp8,tmp9 mov_i32 r6,tmp8

Table 2. Source Log Information

ARM instruction	TCG intermediate code	X86 instruction
add r3,r1,r2	add_i32 r3,r1,r2	① mov 0x4(%ebp),%ebx ② mov 0x8 (%ebp),%esi ③ add %esi,%ebx
add r5,r2,r3	add_i32 r5,r2,r3	④ mov 0x8(%ebp),%esi ⑤ mov %ebx,%edi ⑥ add %edi,%esi
add r6,r1,#12	mov_i32 tmp8, $0xc add_i32 r6,r1,tmp8	⑦ mov 0x4(%ebp),%edi ⑧ add $0xc,%edi

 Copyright ⓒ 2015 SERSC

5. Design and Implementation of the Backend Register Allocation Algorithm

Backend does not retain the register which assigned to the source operand. So when neighboring instructions to use the same variable, you need to re-apply for the register, re-read the contents of memory, as Table 2 in the statement ①⑦and②④. QEMU will retain the use of the destination operand register, but when used again, still need to re-apply and produce movement between registers, statement ⑤, such as in Table 2, this mobile has no practical meaning. After allocating registers for variables, register mapping was not implemented according to subsequently variable using condition, only implemented giving up the use of registers or mapping by the inherent rules. While there are not enough registers, could not meets the application, a register is written back into memory to vacate it according to inherent rules. The inherent using condition of the spilled out variable was not considered during the hole procedure and could not determinate the weight of the current spill out based on the analysis of overall intermediate code.

Although QEMU translates in the unit of basic block, implemented code activity analysis of the block source code, it did not determine the allocation of the registers according to its usage range, wasted the optimization space of the block code. The unreasonable allocation and usage of registers caused the generated host code more complex. The subsequent sections of this article will introduce a true register allocation algorithm at the backend of QEMU, fully use the superiority of high access speed of registers, reduce the running time of host code.

5.1. Register Allocation Algorithm

The life time of variable (life interval, also known as active interval): The range between the first definition or use of the variable to the last use of it is called its life time. A register could not be allocated to variables active at the same time. Linear scan register allocation algorithm to perform a reverse scan in the intermediate code, collected variable's life interval information in this process and then accordance with the order of appearance in the instruction assigned to register. Suppose there are R available physical registers and n variable's life interval overlap in a point. If n> R, then there are at least n-R variables must be sent to the memory cache, which is called overflow. The overflow means it's every access must directly manipulate memory. The target of register allocation algorithm is that variable occupy a register as long as possible, avoid be spilled out.

The definition of V is the set of all life interval, two vectors defined below will be used in the process of register allocation [5]

Lives = {Vi | Vi Є V , if start(Vi) < start(Vj) , then Vi < Vj}.

Active = {Vi | Vi Є V , Vi has been assigned register, if end(Vi) < end(Vj) , then Vi < Vj}.

Vector Lives stored the life time of every variable and sorted the by the ascend of its begin time and users can conveniently find the start time of a variable's life time. Linear scan algorithm need maintain an active list which recorded active ranges allocated registers at the currently time during generating the host binary code. The list is the active vector, life time stored in it is sorted by the ascend sequence of stopping point of life times.

Register allocation algorithm is as follows [3]: LinearScanRegisterAllocation

Active ← {}

foreach Vi

ExpireOldIntervals(Vi)

Copyright ⓒ 2015 SERSC

```
            if length(Active) = R then
                  SpillAtInterval(Vi)
            else
register[Vi] ←a register removed from pool of free registers,add Vi to Active,sorted by increasing end point
ExpireOldIntervals(Vi)
foreach Vj in Active
            if endpoint[Vj] >= startpoint[Vi] then
                  return
            remove Vj from Active
            add register[j] to pool of free registers
SpillAtInterval(Vi)
spill← last interval in Active
if endpoint[spill] > endpoint[Vi] then
      register[Vi] ← register[spill]
      location[spill] ← new stack location
      remove spill from Active
add Vi to Active,sorted by increasing end point
else
location [Vi] ← new stack location
```

Linear scan algorithm allocate registers according to the begin time of life time in the intermediate code. It need to read every variable's life time one by one to allocate registers. Vector Active was scanned and overdue life times were eliminated before allocating registers to a life time. The method is to compare the ending time of life times in Active with the begin time of new life time, if the end time is smaller than the begin time, it implies that the life time in Active already finished when the new life time begin and can remove the life time from active to release the corresponding register. Life times in Active is sorted by the ascend of their end time and can easily find those to be removed. Successively compare the begin time of the new life time with the end time of life times in Active, you can stop subsequent comparison if the result is not more than, life times ahead could be removed from vector Active.

Above algorithm need to overflow, choose end time late life interval. We can find the life time to be spilled out by comparing the new life time and the last life time in vector Active. This article will improve overflow algorithm. Specifically refer to section IV.B.

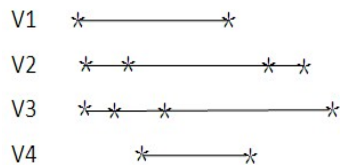


Figure 2. Note How the Caption is Centered in the Column

5.2. Overflow Algorithm Implementation

Shown in Figure 2, the existing life of V1, V2, V3, and V4. The symbol '*' represents the variable is used in this position.

It is assumed that the number of registers R = 3, according to the traditional linear scan algorithm for register allocation. The results are as follows:

 Copyright ⓒ 2015 SERSC

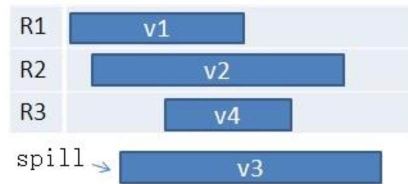


Figure 3. Linear Scanning Algorithm Allocation Results

Life interval of the V3 will be overflow, each operating on V3 will report directly to the memory operation. This increases the memory read.

Improved algorithm, when there is a conflict, separatism life interval and not overflow the whole life interval [4, 6]. If you choose Vj overflow, truncate it in the conflict position. The two parts of life interval were named as Vj_1 and Vj_2. Vj_1 continue to retain the original registers. Vj_2 join Lives vector. The method can reduce the memory read and write times, but to allocate registers Vj_2, still difficult to avoid secondary conflict.

By the analysis shows that, when n> R, select a life interval to overflow or split all want to get more registers idle period, to reduce register pressure. We can record the length of the interval of the life and the frequency of use in the calculation of the range of life. When conflict occurs, select the largest weight do processing. The length of the interval of life time is computed by subtracting the instruction label at the start position of life time by the instruction label at the end position. Weight formula:

weight = Life interval length/reference number

We can blur the weight of the largest range of life, using a minimum density. We realized this approach, called mode1. Overflow algorithm can be described as：

SpillAtInterval(Vi)
spill←the least intensity of use in Active
register[Vi] ← register[spill]
split the spill into the spill_1 and the spill_2 at current point
remove spill_1 from Active
add spill_2 to Lives, sorted by increasing start point
add Vi to active

The most intuitive approach is to select the life interval which next available position furthest away from the point of conflict. So as to appear the largest register idle period. As the previous example, when V4 register allocation conflicts and V2's next use is the furthest away from the conflict position. So, choose V2 to split, to ease the use of the register pressure can be the most efficient. Register allocation results using this method shown in Figure 4:

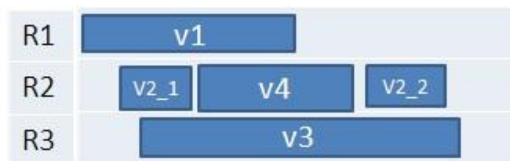


Figure 4. Optimal Allocation Results

The algorithm needs to be able to get the variable's next use position in the conflict. So, you need to record every use of the variable position in the statistics of the life interval of the information. Intermediate code analysis process becomes more complicated. We also realize the method, called mode2. Overflow algorithm can be described as:

SpillAtInterval(Vi)
spill←furthest reference in Active
register[Vi] ← register[spill]
split the spill into the spill_1 and the spill_2 at current point
remove spill_1 from Active
add spill_2 to Lives,sorted by increasing start point
add Vi to active

Neither of these two algorithms required stored Active's life interval of the endpoint from small to large order. But, need to record the frequency of use of the variable or use position in statistical process. This statistical of the life interval process more time-consuming. We need to re-scan the intermediate code, divide the life time of spilled out variable when spilling out occurred. Repeatedly scanning of the intermediate code can increase the binary translation time but it also can reduce the running time of executable code.

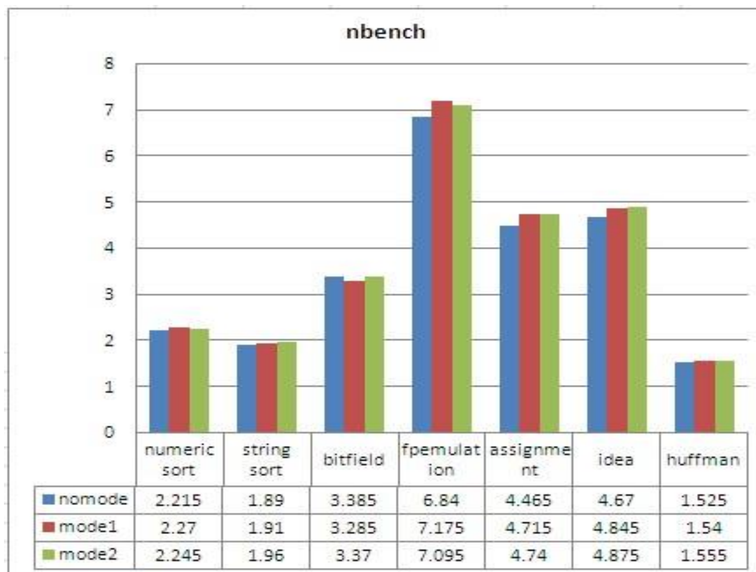
5.3. The Experimental Results

The test was carried out by running the nbench benchmark suite [11] (the performance testing program recommended on QEMU official website) test suit on QEMU. Nbench is a simple basic standard test program to test the performance of the processor and memory. The result is shown as Figure 5:

Compare the running result of nbench on current system and on an AMD K6-233 computer with linux environment we can see that both of the spill out algorithm can improve the performance of QEMU in Various types of calculations. The difference between mode1 and mode2 is not conspicuous while mode2 is better.

The following test results, the blue represents the unmodified QEMU Red said the mode1, Green said mode2.

Run three test examples, respectively statistical the TCG translation time, the host binary run-time, as well as translation and run time sum. As shown in Figure 6:



	numeric sort	string sort	bitfield	fpemulat ion	assignme nt	idea	huffman
nomode	2.215	1.89	3.385	6.84	4.465	4.67	1.525
mode1	2.27	1.91	3.285	7.175	4.715	4.845	1.54
mode2	2.245	1.96	3.37	7.095	4.74	4.875	1.555

Figure 5. nbench Test Results

Copyright ⓒ 2015 SERSC

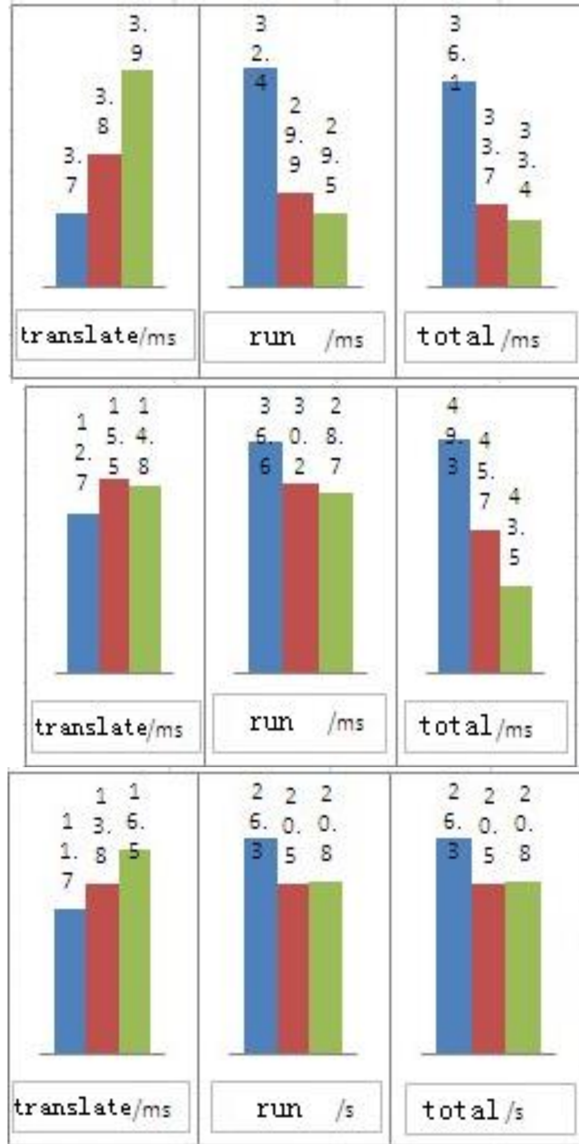


Figure 6. Translate, Run, Total Time Statistics

The height of the column line graph in Figure 6 indicates time. The figures upper the columnar Article denote translation/run/full time of cases in the QEMU version, time units are given in the corresponding figure.

The modified QEMU translation process will be used more time. From the running time, mode1 and mode2 running time is shorter than the unmodified QEMU, and the mode2 running time is slightly better than the mode1. By the sum of translation and run time, this algorithm can indeed reduce the overall program run time. Register allocation algorithm can improve the quality of reduced code, reduce the amount of code. There were repeated accessing of memory in reduced host code. The reducing of such operations can cut down time use of code running after translation because the access of memory can waste lots of time. In the third example, the translation time to the millisecond, the running time for

Copyright ⓒ 2015 SERSC

seconds, that there is more duplication of code procedures, saving time more significantly, to 23% here.

The following run seven test examples, and statistical the number of instructions of the target platform. Both algorithms are able to certain procedures to reduce the final target platform number of instructions. Specific circumstances as shown in Figure 7:

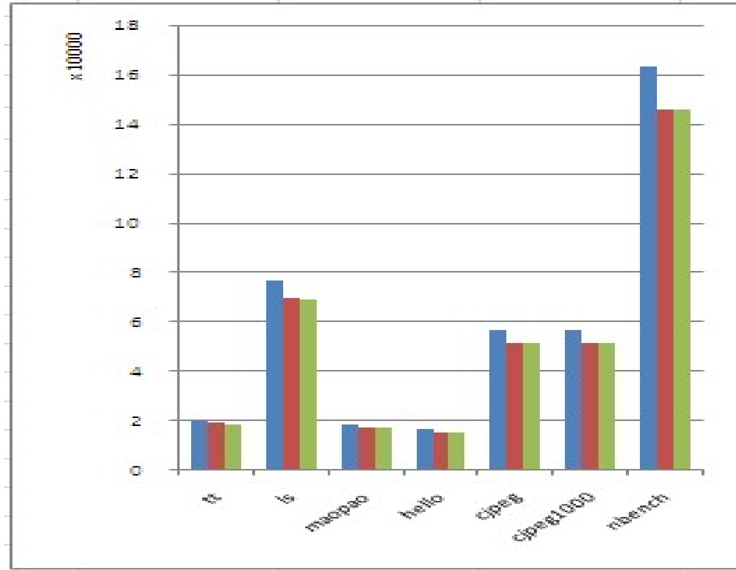


Figure 7. Hosts the Amount of Code Statistics

From the result we can see that register allocation algorithm can improve the use ratio of registers, reduce the repeated memory access operation and data move operations among registers. Both algorithms could effectively reduce the number of eventual instructions of the host. The reduced amount of instructions respectively achieved 10.42 percent and 10.60 percent in nbench case. Mode2 chose the spill out with the farthest use distance, more accurate than the vague weight computing of mode1. So mode2 can get more register free time, eased the use pressure of registers as much as possible and the quality of generated code by mode2 is higher than by mode1.

6. Concluding Remarks

The QEMU with TCG technology to achieve a dynamic binary translation. It uses the secondary translation strategies, the source binary code is translated into intermediate code, and then translated into the target binary code. Twice translation will generate some redundant instruction and greatly increasing the degree of expansion of the code, an average of 6%. TCG simply delete the dead instruction in the intermediate code analysis process, but with little success. In this paper, through downsizing intermediate code, and successfully realized the linear sweep register allocation algorithm in TCG back end. Greatly reduces code bloat. By verifying the increased translation time overhead is less than the time register allocation algorithm overhead. The algorithm has realistic feasibility.

7. Acknowledgment

This work is partly supported by the NSF of Guangxi No.2011GXNSFA018154, 2012GXNSFGA060003 and 2013GXNSFAA019342, the Science and Technology

 Copyright ⓒ 2015 SERSC

Foundation of Guangxi No.10169-1, Guangxi Scientific Research Project No.201012MS274, the Bagui scholarship project, Guangxi University for Nationalities Project, No.2012QD017, the National Natural Science Foundation of China under Grant No. 11371003 and No. 11461006.

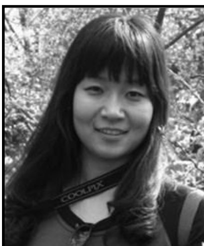
8. References

8.1. Journal Article

[1] F. Bellard and Usenix, "QEMU, a fast and portable dynamic translator [C]//", USENIX Association Proceedings of the FREENIX/Open Source Track, (**2005**), pp. 41-46.
[2] G. J. Chaitin, "Register Allocation & Spilling via Graph Coloring", In Proceedings of the ACM SIGPLAN '82 Symposium on Compiler Construction, vol. 06, (**1982**), pp. 98-105.
[3] M. Poletto and V. Sarkar, "Linear Scan Register Allocation [J]", ACM Transactions on Programming Languages and Systems, vol. 21, no. 5, (**1999**), pp. 895-913.
[4] F. Bellard, "QEMU, a fast and portable dynamic translator", USENIX Annual Technical Conference, APR 10-15, 2005, USENIX ASSOCIATION PROCEEDINGS OF THE FREENIX/OPEN SOURCE TRACK: (**2005**), pp. 41-46.
[5] Z.-l. Liu, W. Guo and J.-z. Wei, "TTA Compiler Optimization Based on Linear Scan Algorithm [J]", Computer Engineering, vol. 36, no. 11, (**2010**), pp. 58-60.
[6] O. Traub, G. Holloway and M. D. Smith, "Quality and Speed in Linear-scan Register Allocation[C]//", Proc. Of ACM SIGPLAN'98. [S.1.]: ACM Press, (**1998**), pp. 142-151.
[7] D. Ung and C. Cifuentes, "Machine-adaptable dynamic binary translation [J]", ACM SIGPLAN Notices, vol. 35, no. 7 , (**2000**), pp. 41-51.
[8] G. J. Chaitin, M. A. Auslander and A. K. Chandra, "Register allocation via coloring [J]", Computer languages, vol. 6, no. 1, (**1981**), pp. 47-57.
[9] L. George and A. W. Appel, "Iterated register coalescing [J]", ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 18, no. 3, (**1996**), pp. 300-324.
[10] C. Wimmer and H. Mössenböck, "Optimized interval splitting in a linear scan register allocator[C]//", Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments, ACM, (**2005**), pp. 132-141.
[11] F. Bellard, http://www.qemu. org, (**2005**) September.

Authors

Yi Liang, MS, engineer, her research interest includes algorithm optimization, Guangxi University for Nationalities. Her current research focuses on evaluation and expectation of performance of complex systems, her research interests include quantum computing, reversible logic, hardware formal verification, floor planning, and routing.



Yuanhua Shao, she is currently an Engineering student at Baidu Company. She received the M. S. degree in computer science from University of Science and Technology of China. Her research interests include embedded systems, algorithms.

Guowu Yang, He received the B.S. degree in mathematics from the University of Science and Technology, China, in 1989, and the Ph.D. degree in electrical and computer engineering from Portland State University, Portland, OR, in 2005.He is currently a Post-Doctoral Researcher in the Department of Computer Science, Portland State University. His research interests include quantum computing, reversible logic, hardware formal verification, floor planning, and routing.

Jinzhao Wu, He is a professor of School of Information Science and Engineering, Guangxi University for Nationalities. His research interests include formal specification and verification of software/hardware system, formal semantics, etc. His current research focuses on evaluation and expectation of performance of complex systems.

 Copyright ⓒ 2015 SERSC