2018 IEEE International Conference on Software Quality, Reliability and Security Companion

Research on CodeCache Management Strategy Based on Code Heat in Dynamic Binary Translator

Fei Deng,Feng Gao, Yunqiang Yan,Wei Zou
Institute of Computer Application
China Academy of Engineering Physics
MianYang,China

***Abstract*—Software verification and validation play an important role in the development process of equip embedded software. It is a research hot spot in software testing field to adopt full digital simulation test environment based on a virtual target machine, which has the advantages of not relying on hardware, convenient fault injection etc. CodeCache management strategy is the key factor affecting the speed of the full digital simulation. It has analyzed the deficiency of existing translation caching methods and proposed a translation cache partition management strategy based on code heat. This strategy has considered the temporal and spatial locality of the program execution and the replacement cost of Cache, realizing the efficient management of CodeCache. The test result shows that the strategy has efficiently improved the simulation performance of virtual target machines.**

Keywords—Software Testing;Dynamic Binary Translator; QEMU; CodeCache Management ;Full Digital Simulation;Code Heat

I. INTRODUCTION

With the rapid development of information technology, embedded software plays an increasingly important role in the core control logic of weapon equipment and the complexity of its design has multiplied [1-2]. The quality of embedded software becomes the key factor to determine product reliability and safety [3-5]. In order to improve the quality of embedded software, reliable and effective software testing becomes more and more important. However embedded software is usually developed in the host machine. It runs in the target machine and is closely related target hardware. These makes verification and validation rely too much on hardware system in the condition of lacking adequate technical means. This brings great difficulty to the work. [6] But it can get rid of the dependence on hardware by testing embedded software with full digital simulation test environment based on virtual target machines. It is widely recognized as an effective technique at home and abroad with the advantages of low cost, transparent test environment, convenient fault injection, strong controllability etc . [7]

QEMU is an open source cross-platform dynamic binary translator. It can invent target hardware platforms and various peripherals like SPARC, ARM, MIPS, POWERPC etc. [8-9] And it has the characteristics of cross-platform operation, so it is ot necessary to think about compatibility while designing. It really has a great advantage to achieve the target machine simulation with QEMU.

QEMU translates according to the basic blocks. In order to avoid repetitive translation, QEMU will store existing translation blocks in memory for next use. This memory for the translated code is called CodeCache [8]. At the beginning of translating, QEMU will firstly if there's a target code block which has been translated in CodeCache. If it finds the translated code blocks, it will execute the target code directly. If it doesn't find the translated code blocks, it will execute the translation process and the optimized translation results will be cached in the CodeCache. The purpose of multiple executions in one translation process can be realized through CodeCache, thus improving the simulation performance. The performance of the CodeCache directly influences the speed of binary translation. So it is of great significance to improve the simulation performance of QEMU by studying the strategy used to effectively manage CodeCache.

The advantages and disadvantages of common substitution algorithms are analyzed in the second section of this paper. CodeCache replacement algorithm adopted by QEMU is introduced in the third section. The fourth section describes the ontology and proposes management algorithm in detail. The fifth section carries out the experimental analysis and finally summarizes and makes prospect.

II. COMMON REPLACEMENT ALGORITHM

A. The Feature of CodeCache

CodeCache stores the translated target code in the binary translator. Because the translation is based on the basic blocks, there are different characteristics to the Cache in the CodeCache and computer hardware systems. Therefore the following questions should be considered while designing CodeCach [10-12]:

- CodeCache is in an area of the but not existing independently. So to improve the performance of CodeCache, efficient management strategy and concise data structure need to be adopted.
- The content of CodeCache has no reproducibility. Once a block of code is replaced or the entire Co-

978-1-5386-7839-8/18/$31.00 ©2018 IEEE
DOI 10.1109/QRS-C.2018.00117

deCache is empty, the next time the same block of code need to be re-translated.

- Code blocks in CodeCache are interlinked. The existence and location of the block rely on each other. If a block of code is removed from the CodeCache, all all the pointers associated with this code block need updating. Therefore, frequent changes will result in frequent pointer manipulation, and administrative overhead will increase with it.

B. CodeCache Replacement Algorithm

At present, there are following common CodeCache replacement strategies in dynamic binary translation systems [13-16]:

1) FLUSH strategy

FLUSH strategy is to place CodeCache code blocks in turn according to addresses. While there is not enough free space in CodeCache , all code blocks will be replaced to empty the entire CodeCache. In this strategy, the algorithm is simple, administrative overhead is also small and there's no fragmentation. But the replacement granularity of this method is too large, which would cause a high loss rate. At the same time it doesn't take into account the temporal and spatial locality of the program. It will replace a lot of hot code blocks and cause the repeated substitution of the hot blocks. And the performance of CodeCache will be degraded.

2) LRU strtegy

LRU strategy is to replace least used blocks out while there 's no enough space in CodeCache. If the space is still not enough after replacement, the adjacent code blocks will also be replaced. The process will be repeated until CodeCache has enough space to place new code blocks. This algorithm takes into account the temporal and locality of the program. It predicts its future execution features according to the recent execution features. But its algorithm is complicated, the management overhead is expensive. Sometimes its performance is not as good as FLUSH strategy. So they're not many binary translation systems using LRU strategy.

3) FIFO strategy

FIFO strategy is to regard CodeCache as a FIFO buffer. While CodeCache doesn't have enough space, the first code block that goes into the CodeCache will be replaced. If the CodeCache still doesn't have enough space, the second code block that goes into the CodeCache will be replaced until there 's enough space for new code blocks. FIFO considers the time locality of the program in some extent. But its overhead is expensive. Generally, binary translation systems don't use it independently, but use it together with other algorithms.

4) Other management strategies

Except above CodeCache management strategies, there're some other management strategies, such as all clear strategy based on working sets, no replacement strategy, the largest priority replacement strategy etc. The all clear strategy based on working sets requires a clear phase separation. No replacement strategy requires that CodeCache space is large enough. The largest priority replacement strategy only considers the spatial locality of the program but it doesn't consider the temporal locality and execution characteristics. These policies are generally not used in dynamic binary translation systems.

III. CODECACHE MANAGEMENT STRATEGY IN QEMU

QEMU adopts FLUSH strategy to manage CodeCache. During program execution, QEMU will make a judgment to CodeCache while there are new code blocks to be translated[8]. If the free space of the CodeCache is insufficient to store the new code block, QEMU will recycle all the space of CodeCache and delete all translation results. So the basic blocks of all subsequent execution must be retranslated. The frame of QEMU code management is shown in Fig. 1. QEMU organizes the translation information of the basic blocks into a hash table. Tb_phys_hash[] has saved the translation information of all basic blocks. QEMU system will firstly look for the information according to virtual PC value in Tb_phys_hash[] while executing a piece of code. If it can find the information, that means the basic block has been translated. The system will get the translated target code blocks and execute it. If it can't find the information, the system will translate the basic block in binary with translation function. At the same time the translation information and translation results of basic blocks will be put in Tb_phys_hash[] and CodeCache[].

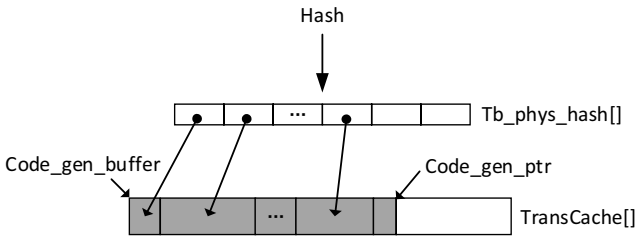


Fig. 1. QEMU code management frame diagram

As is shown in Fig. 1, the grey part shows the used part of CodeCache. QEMU uses the global pointer code_gen_buffer and code_gen_ptr to preserve CodeCache. Code_gen_buffer points to original address. Code_gen_ptr points to target code area. When a new block of code needs to be translated, QEUM will judge if there's enough free space of CodeCache according to the difference value of code_gen_ptr-code_gen_buffer. If there's enough free space, QEMU will allocate cache Spaces, translate basic blocks, put the translation information of basic blocks into Tb_phys_hash[] and put the translation results into CodeCache[]. If there's not enough free space, then empty all content of CodeCache[] and Tb_phys_hash[],reset the code area pointer code_gen_ptr and make it equals to code_gen_buffer. After translate new code blocks, update tb_phys_hash[]、CodeCache[] and code_gen_ptr.

QEMU adopts FLUSH strategy to manage CodeCache. The implementation is simple and does not produce fragments. And the administrative overhead is low. But FLUSH doesn't consider the temporal locality of the program. The replacement granularity is large. Every time it is emptied, the source code to be executed must be retranslated. And that makes the loss rate of the entire CodeCache too high and will degrade the performance of CodeCache.

IV. PBMS STRATEGY

A. The Principle of The Algorithm

The execution of the program obeys the 80/20 rule according to the temporal locality principle of program. That means in 80% of the time CPU are executing the 20% of the hot code, and 80% of the code is not hot code. Such as 181.mcf in SPECint2000, execute mcf once and 1653 target code blocks will be produced. The most common target code block is executed more than 2 million times. But most target code blocks will turn to waste and wait to be recycled after CPU execute them. But FLUSH replacement strategy adopted by QEMU doesn't consider the execution characteristics of the program. For this feature of the program, this paper will use the replacement algorithm based on Profile to try to keep hot code in CodeCache.

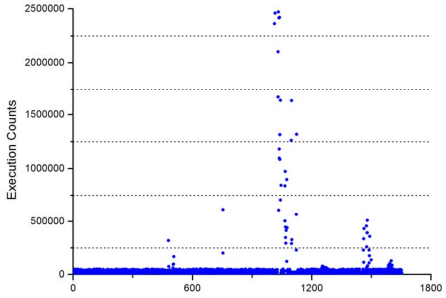


Fig. 2. The distribution of the execution number of 1653 target code blocks of 181.mcf

This paper presents BPMS((Profile-based management strategy)) CodeCache management strategy. The main idea is to divide the code blocks in CodeCache according to their number of executions. And then separate hot code from common code, forming a multilayered CodeCache in logic. At the same time, choose different management strategies according to the characteristics of the two kinds of code. The principle is to divide CodeCache into L1 SubCache and L2 HotCache, store the new translated block of code in the SubCache during program execution and collect Profile information of the code at the same time. When the code block's heat reaches a certain threshold Limen, it is transferred from SubCache to HotCache. CodeCache in QEMU is simulated in memory. So it's different from the multi-level Cache of the hardware. One code block has only one copy in the entire CodeCache. QEMU saves the translation information of all basic blocks through tb_phys_hash[]. Before translating a code block, QEMU will look for tb_phys_hash[] first. If QEMU finds it, that means this code block has already been translated. Then extract the corresponding target system code and take execution. So the execution performances are the same, no matter when it's QEMU in SubCache or HotCache in Cache. Therefore it directly replaces code blocks when the replacement is in SubCache and it replace the code blocks to SubCache when the replacement is in HotCache. This allows the hot code to remain in CodeCache for a long time. Because it's normal code, SubCache can use the simple and non-fragmented code replacement algorithm such as all clear algorithm, coarse-grained FIFO and other algorithms. An it's all hot code in HotCache. In order to minimize thermal code replacement frequency, it adopts FIFO replacement algorithm. The two-level Cache structure brings a combination of various management styles. Therefore, it is possible to select a simple and efficient combination of algorithms based on the characteristics of various levels of Cache.

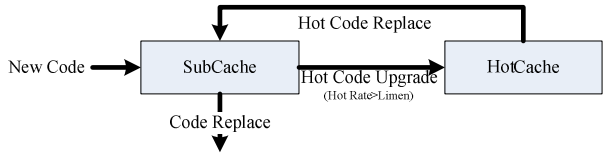


Fig. 3. BPMS translation cache management diagram

B. The Size Selection and Allocation of CodeCache in PBMS Algorithm

The performance of CodeCache can be measured from Time performance and spatial performance. Spatial performance refers to the size of memory space occupied by the CodeCache, and the measurement is simple and unique. Time performance is measured by Cache Access Time.

$$CodeAccessTime = HitTime + MissRate * MissPenalty \quad (2)$$

Table 1 summerizes the relationship among Hit time, Miss rate and Miss penalty. It shows that if the capacity of CodeCache is higher, the Hit rate will be higher, the Miss rate will be lower and the Hit time will increase. The organization of code blocks also affects Hit time and part of Miss penalty. The impact of replacement algorithms on Miss rate and Miss penalty is also significant. In general, replacement algorithms with low Miss rate tend to mean higher Miss penalty. So we need to compromise among these factors.

TABLE I. DECISIVE FACTORS OF HIT TIME, MISS RATE AND MISS PENALTY

Item	*Decisive factors*
Hit time	CodeCache size and the data organization of the target code block in CodeCache
Miss rate	CodeCache size and replacement algorithm
Miss penalty	Translation speed，replacement algorithm and the data organization of the target code block

In BPSP algorithm, the Selection and distribution of CodeCache size and the selection of threshold Limen can directly affect the performance of the algorithm. The size of CodeCache is different so the block of code that can be stored is different. The bigger CodeCache is, the more blocks of code can be restored, then QEMU is more likely to be hit by doing dynamic binary translation and the number of substitutions is less. This can improve the execution speed of the program. However, the size of CodeCache cannot be increased indefinitely because the storage capacity of the hardware is limited. At the same time, as CodeCache is bigger, overhead of managing the whole thing is also going to scale up and the Hit time of the block of code also increases. Therefore many factors should be considered to select the size of CodeCache, such as hardware memory capacity, the managing overhead unit cost of

CodeCache, the expansion rate of code translation, the size of the translated code, the managing strategy of CodeCache etc. Generally, the management cost unit cost of CodeCache is a constant. That leads to:

$$ManagementCost = F*UnitCost*(CodeSize*ExpansionRate - CacheSize) + CacheSize*UnitCost \quad (2)$$

F is a rapidly increasing function with code size increasing. Its increasing speed is related to the management strategy. The more efficient the management strategy is, the more slowly f increases. In order to improve the efficiency to manage CodeCache, on the one hand, the size of CodeCache needs to be increased to reduce the first cost. But as TrasCache increases, Hit time also increases. On the other hand, the size of CodeCache needs to be decreased to reduce the second cost. But as CodeCache decreased, Miss rate increases. Therefore, different f functions must be obtained according to different management policies of CodeCache. At the same time, according to code size and the size of the code expansion rate, the minimum cost of managing the CodeCache is obtained. And this is the size of CodeCache .

When the size of CodeCache is fixed, the size of SubCache and HotCache will directly affect the execution efficiency of the program. When SubCache is too big and H-tCache is too small, hot code can just stay in CodeCache for a short time. According to the 80-20 rule of execution of the program, the Hit rate of CodeCache will redue with it and the Miss rate will increases with it. So the replacement cost will increase. In BPSP strategy, the code blocks in HotCache will firstly be replaced into SubCache. And they will be replaced out from SubCache when they need replace new code blocks. So the cost of code replacement in HotCache is higher than that in SubCache. When HotCache is too big, the replacement cost of the entire CodeCache will increase. If the size of code block heat threshold Limen is different, the number of code blocks needs transferring from SubCache to HotCache is different. If the Limen value is set too high, no code block will be promoted to HotCache in extreme cases. That only uses the SubCache part of CodeCache. On the contrary, if the Limen value is set too low, large number of code blocks in SubCache will be replaced in to HotCache. The code blocks that are replaced from HotCache are easily replaced by hot code that is repeatedly replaced with HotCache and the replacement number will be hugely increased. So the distribution of CodeCache is related the size of CodeCache, the management strategy of CodeCache, code block heat threshold Limen and other factors. It can directly affects the management strategy efficiency of the entire CodeCache. Therefore, it is very important to distribute CodeCache and choose Limen value reasonably. If the management strategy and size of CodeCache are fixed, the distribution of CodeCache will only be related to the selection of Limens. So when the management strategy and size of CodeCache are fixed, the following relations can be obtained:

$$HotCache = \Pr ortion * \frac{Code\ Size * Expansion\ Rate}{Limen} * Cache\ Size \quad (3)$$

Proportion is a scaling factor. It is related to management strategies. So when the Limen is bigger, the HotCache is smaller. We can select the appropriate Limen value and HotCache value based on the features of the translated code to make the performance of the algorithm optimal.

In QEMU, the optimized module after translation is based on the source code execution process, forming a block of code. SubCache and HotCache of PBMS algorithm are organized according to CodeCache address from low to high. At the same time, the basic blocks are organized according to the address from high to low inside SubCache and HotCache. This logic ensures that the flow of the program is consistent across the space. It can be seen from the substitution strategy, the hot code does not replace CodeCache directly when replacing it. Only uncommonly used code blocks are replaced directly. In this way, the time locality of the program is guaranteed.

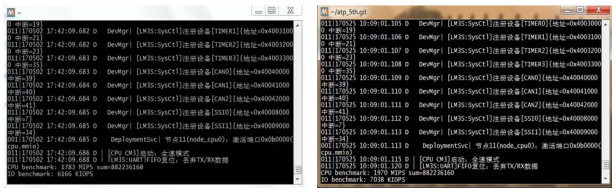
The improved algorithm strategy does not need to traverse the entire CodeCache every time it is replaced. Compared with LRU algorithm, it saves time and is not as complicated as LRU, which is crucial to improve translation efficiency. In addition, the PBMS strategy places the heat to a certain threshold of code in a separate HotCache. And code blocks in HotCache need to be replaced without being replaced directly. This strategy improves the retention time of the hot code in the CodeCache as far as possible and enhances the Hit rate of CodeCache. This is the flaw of FLUSH strategy. For strict FIFO algorithm, the execution features of the program isn't considered and code blocks are purely replaced according to First In First Out. This may replace the hottest code out. The improved algorithm can avoid this problem very well and has a certain improvement in the performance compared with FIFO algorithm.

The shortcoming of this algorithm is that the code block size is not considered. If you want to allocate a large block, you might want to replace multiple blocks. This requires a cumbersome pointer operation. If the block that is to be allocated is smaller than the remaining space to be allocated, the fragment will appear, resulting in the waste of the CodeCache space.

V. EXPERIMENTAL ANALYSIS AND EVALUATION

In order to test the performance of PBMS algorithm, we wrote an ARM test program, CPU benchmark and IO benchmark to test the simulation performance of QEMU(ARM Cotex-M3 on X86). CPU benchmark includes addition, subtraction, multiply, division, circulating, skipping and md5 algorithm, used to test the simulation speed of the ARM cortex-m3 kernel. The test method calculates the assembly instructions for the verification program first. Run the validation program through QEMU and record the start time and end time. Finally the simulation performance is calculated: the number of assembly instructions for the test program/ （start time - end time）. IO benchmark has ARM Cortex-M3 kernel, serial interface and GPIO used to test the simulation performance of the test kernel and peripherals co-simulation. The test method is to calculate the number of times and the start time of the external set status register. Finally calculate the co-simulation

performance: the number of visits to the status registers/（start time - end time）. PBMS adopts and collects profile information to identify hot code. Threshold Limen is set to 200. When the execution is over 200 times, code blocks will be moved from SubCache to HotCache. The default CodeCache size in QEMU is 32KB. To facilitate testing, we changed the value of CodeCache to 12KB, SubCache to 8KB and HotCaache to 4KB. This ensures that a certain number of substitutions occur during the benchmark test run. The test result is shown in Fig. 4.



a）QEMU original strategy　　b)PBMS strategy

Fig. 4. Comparison of QEMU in different management strategies

The comparison test shows that, while simulating ARM Cotex-M3 in X86, the QEMU kernel simulation speed with FLUSH management strategy is 1783MIPS and the kernel peripheral simulation speed is 6166KIOPS. The QEMU kernel simulation speed of the PBMS strategy proposed in this paper is 1970MIPS, and the simulation speed of kernel peripherals is 7038KIOPS. According to the test results, QEMU simulation performance of BPMS strategy is significantly better than that of unoptimized QEMU. The improved ratio of BPMS strategy compared with QEMU original strategy is shown in table 2.

TABLE II. COMPARISON BETWEEN BPMS AND QEMU ORIGINAL STRATEGY

Item	*QEMU original strategy*	*PBMS strategy*	*Performance improvement rate*
CPU simulation speed	1783MIPS	1970MIPS	10.49%
IO simulation speed	6166KIOPS	7038KIOPS	14.14%

VI. SUMMARY AND DISCUSS

The management efficiency of CodeCache plays an important role in the simulation performance of dynamic binary translator. Reducing Hit time and improving Hit rate can effectively improve the management efficiency of CodeCache. This paper analyzed the advantages and disadvantages of the commonly used CodeCache replacement algorithms. Then BPMS management strategy based on profile information is presented according to CodeCache management features in QEMU. And CodeCache is divided into SubCache and HotCache. During the translation process, collect the heat information of the code block and store new code block in SubCache. The code blocks whose heat reaches a certain threshold Limen are put in HotCache. At the same time, the code block in the SubCache will be replaced directly while replacing. And code blocks in HotCache will be replaced in SubCache. This will allow the hot code to stay in the field for a long time and improve the hit rate. BPMS algorithm does not need to traverse the entire CodeCache every time it is replaced. This can efficiently reduce Hit time. The two-level Cache structure brings together a variety of management methods, which can be used to select a simple and efficient combination of algorithms based on the characteristics of all levels of the Cache, thus improving the simulation performance. However, the BPMS management strategy does not take into account the size of the block of code, which may result in the fragmentation of the storage space and the waste of CodeCache space. At the same time, how to set the size is still a problem to be solved. In the future, we hope to solve the above problems through research and more detailed quantitative analysis to build a more perfect mathematical model of the two-level Cache.

REFERENCES

[1] Gai, Paolo, and M. Violante. "Automotive embedded software architecture in the multi-core age." Test Symposium IEEE, 2016:1-8.

[2] Enoiu, Eduard P., et al. "A Controlled Experiment in Testing of Safety-Critical Embedded Software." IEEE International Conference on Software Testing, Verification and Validation IEEE, 2016:1-11.

[3] Iqbal, Muhammad Zohaib, A. Arcuri, and L. Briand. "Environment modeling and simulation for automated testing of soft real-time embedded software." Software & Systems Modeling 14.1(2015):483-524.

[4] Jie, Jason Lee Hua. "Industrial Case Study of Transition from V-Model into Agile SCRUM in Embedded Software Testing Industries." Acm Sigsoft Software Engineering Notes 41.2(2016):1-3.

[5] Qu, Mingcheng, et al. "An Embedded Software Testing Requirements Modeling Tool Describing Static and Dynamic Characteristics."International Symposium on Computers and Informatics 2015.

[6] Ze, X. U. "Digital Simulation of Full Scale Static Test of Aircraft." Chinese Journal of Aeronautics 18.2(2005):138-141.

[7] He, Yingjie, and F. Zhang. "Research on Embedded Software Testing Technology Based on Full Digital Simulation." Science Mosaic (2015).

[8] Bartholomew D. QEMU: a multihost, multitarget emulator[M]. Belltown Media, 2006.

[9] Höller, Andrea, et al. "QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks." Digital System Design IEEE, 2015:530-533.

[10] He, Yunchao, et al. "A New Approach to Reorganize Code Layout of Software Cache in Dynamic Binary Translator." International Symposium on Parallel Architectures IEEE, 2010:175-182.

[11] Hm, Igor, et al. "Generalized just-in-time trace compilation using a parallel task farm in a dynamic binary translator." ACM Sigplan Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, Ca, Usa, June DBLP, 2011:74-85.

[12] Xu, Chaohao, et al. "Metadata driven memory optimizations in dynamic binary translator." International Conference on Virtual Execution Environments, VEE 2007, San Diego, California, Usa, June DBLP, 2007:148-157.

[13] Yin, Jinbiao. "TCACHE MANAGEMENT SCHEMES FOR DYNAMIC BINARY TRANSLATOR QEMU." Computer Applications & Software (2012).

[14] Hazelwood, Kim, and M. D. Smith. "Generational Cache Management of Code Traces in Dynamic Optimization Systems." Ieee/acm International Symposium on Microarchitecture, 2003. Micro-36. Proceedings IEEE, 2003:169-179.

[15] Ma, Ruhui, et al. "Code cache management based on working set in dynamic binary translator." Computer Science & Information Systems8.3(2011):653-671.

[16] Yue, Feng, et al. "An Improved Code Cache Management Scheme from I386 to Alpha In Dynamic Binary Translation." International Conference on Computer Modeling & Simulation IEEE Computer Society, 2010:321-324.