2010 2nd International Conference on Information and Multimedia Technology (ICIMT 2010)
IPCSIT vol. 42 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V42.22

DCC: A Replacement Strategy for DBT System Based on Working Sets

Zhichen Ni, Kai Chen, Ruhui Ma , Hongbo Yong, Yi Zhou and Haibing Guan

Department of Computer Science & Engineering, Shanghai Jiao Tong University, Shanghai, China
{nizhichen,kchen,ruhuima,yanghongbo819,hbguan}@sjtu.edu.cn, Zy_21th@yahoo.com.cn

Abstract—Limited memory resource has always been deemed as the major bottleneck of the program performance, not excepting for dynamic binary translation (DBT) [1] system. However, traditional methods seldom enable this issue mentioned to be solved better due to a fix cache size routinely assigned for codes without considering the program behavior on the fly. Though some prediction methods implemented via LRU [2] histograms can achieve better performance in Operating System, unlike pages in memory, blocks in DBT have unfixed sizes each other, making the prediction imprecise. In this paper, we present a new replace policy named DCC (Dynamic Code Cache) based on working set[3] for bounded code cache, which would dynamically change the size of the code cache and clear the cache according to the information of working set detected. It would sacrifice a small part of the performance to save the space, especially when many great or complicated applications are running on the same physical machine, and the performance might be better under this condition. For Evaluating, we test this strategy on a DBT system named Crossbit [4]. We get around 32% space saved up with about 9% running time sacrificed.

Keywords-dynamic; working set; code cache; DBT; replacement strategy

1. Introduction

As the current application programs could do much more complicated tasks than before, the requirement of hardware resources has accordingly been increased, especially the memory resource. When many resource-consuming applications are running on the same physical machine, the shortage of memory is getting more obvious. As a result, the limited memory resource has become one of the bottlenecks of the program performance. The dynamic binary translation (DBT) systems are facing the same problem.

We try to ease the problem by the unfixed size software code cache and the replacement strategy in DBT systems. Almost all of the DBT systems are implemented with a fixed-size software code cache. However, not all of the blocks(the unit of codes called in DBT) in code cache are highly in use during a certain period of the running time, which means at this period, even if we reduce the code cache's size to the size just holding the highly-used codes might not influent the performance of the application too much. So the key point becomes how to decide the size of code cache at a given time. Working set might probably give us great ideas.

Working set in DBT means the collection of the most recently used blocks of the program run by our DBT system in the software code cache. This concept perfectly matches the point in the last paragraph. As a result, our job is to find the working set at different running time and accordingly adjust our code cache size to the working set's size from time to time. Here, we borrowed some job done by others to decide the transition of working sets, but we make some modification of the method to make it work better for DBT system.

In this paper, we propose a dynamic code cache with a replacement strategy based on working sets. The main purpose of this proposal is to reduce the memory requirement. The code cache does have a bound but it doesn't take the full size all the time. Instead, it just takes part of it at the beginning and the size is dynamically changing, which could grow up to the bounded size at most, decided by the working sets at that

time. When the transition of working set happens or the upper bound of code cache size has been reached. We believe it would not affect the performance too much, as we take the advantage that the working sets are closely relative to the behavior of the program. In a word, it's a dynamic code cache which would adjust its size to the program behavior dynamically, saving part of the memory resource with a little sacrifice in performance.

Some work [5] has already been done to use the memory effectively. However, DBT system is equivalent to neither the operating system nor the system-level virtual machine, the work might not be so effective. Meanwhile, some of those methods cause considerable overhead. On the other hand, our work is quite flexible and low in overhead.

For the rest of the paper, we'll present some relative binary translator (Crossbit), concepts and traditional methods in Section Ⅱ. In Section Ⅲ, we'll discuss the modified replacement strategy based on working sets and the dynamic code cache. Section Ⅳ evaluates the performance and the space saved by our work. Finally, Section Ⅴ will draw the conclusion and talk about the future work.

2. Related Works

2.1 About DBT System and Crossbit

Binary translation (BT) system translates the binary codes of one platform to the other (x86 to MIPS e.g.).It is a translator or simulator mainly solves the incompatible problem between different platforms.

Dynamic binary translation (DBT) system is kind of BT system which does the translation job while executing. Almost Every DBT system holds a code cache to store the blocks already have been translated to improve performance in case these blocks might be used later.

Crossbit is a Dynamic Binary Translator. It's a multi-source and multi-target DBT system.

2.2 Traditional Replacement Strategies

The traditional replacement strategies, such as FIFO [6], LRU and FWF, have been widely used in operating systems. However, due to the unequal size of every block in BT, the traditional strategies might encounter some problems which would not happen in OS, such as the fragment, de-linking*. As a result, they may not achieve their expected performance in BT.

2.3 Original Replacement Strategy Based on Working Sets

The concept of working sets is raised as "the collection of segments recently referenced" at first. In this paper, we would rather define it as the "set of blocks that run recently". Taking the loop circles of the program into account, for a certain period of time, we may regard the program is running among only several blocks. So ideally speaking, during this period of time, even we move other blocks not belonging to this set out of the code cache, the performance of the program would not drop. Now how to determine the working sets correctly has become quite important.

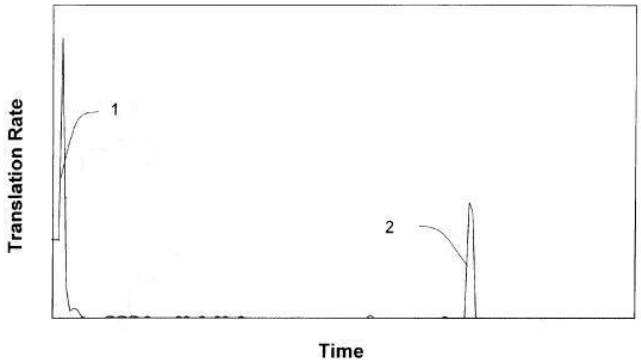

Figure 1. The two WorkingSets Showed by Translation Rate

Actually, this problem has been solved already, using the translation rate and the misprediction rate. In

this paper, we use the translation rate to decide the working set since the latter one would record all the races and cause great overhead. As we could see in Figure 1, the two waves are the two different working sets in different running time. Before the second working set is formed, even if we adjust our code cache size to just holding the first working set, the performance would not be affected.

The way to define the second working set has come could be divided into two steps:

- We set the first threshold as Line 1 in the figure, when the translation rate drops below this threshold, we assume the first working set has almost formed and go to step 2.
- We set the second threshold as Line 2 in the figure, when the translation rate rises over this threshold; we believe the second threshold has come. We could flush the code cache and go to step 1 again.

This approach allows us to do some preemptive job before the code cache is full, and it avoids the de-linking job brought by LRU and other traditional strategy.

However, it is obvious that the existing method has some disadvantage. The code cache does not always work with full size under this strategy since the working sets may migrate at any time. This could hardly promise a better performance than FWF (flush when full) Strategy, so this strategy needs improvement.

As is mentioned above, the traditional strategies may not be able to do the job in BT as well as they do in OS. Meanwhile, the strategy based on working set might conquer part of the problem. It seems that if we fix the disadvantage of the original working set approach, we may find a better solution.

3. Replacement Policy and Dynamic Code Cache

3.1 Overview

As described above, generally speaking, our code cache replacement strategy would have the characters as follows:

- work with a bounded code cache
- take part of its bounded size at the beginning
- replace code cache based on working set preemptive replacement strategy
- adjust its size to the working set from time to time

In this part, we discuss some key points of our strategy and take experiment on Crossbit to determine the key parameters in our strategy. At the end of this section, we'll present our strategy step by step. The points to be discussed are: the value of thresholds, code cache initial size, dynamic-size code cache, self-adjust thresholds.

3.2 The Value of Thresholds

We've already introduced the existing method to detect working sets in Section 2. However, the key point is to find the value of the two thresholds. These two values could determine whether we could get the right working set or not.

It could be easily told that if threshold 1 is set too low, the whole program might just be only one working set as a whole; if it's set too high, as threshold 2 must be higher than it, the next working set might never come. Threshold 2 would accordingly has the same problem. Moreover, the gap between the two values is also quite important.

We have done some experiments to decide the two thresholds' value. We take one benchmark from SPEC2000-INT: MCF_REF, as the test program. Firstly, we record the block ID of the first 1000 blocks of MCF_REF. (FIG. 2)

For example, the ID of the first block to be executed in DBT systems is: 194720.

Through FIG. 2, we could tell that the first 10000 blocks could be divided into 2 working sets, and the transition happens around block NO.3000.

We test whether we could detect the transition with the translation rate's thresholds given below. Since the two thresholds could be neither too high nor too low, we set the range between 0.2-0.6. On the other hand, the gap between the two thresholds should not be too wide, so we take 5%-10% as two choices. The results are shown in Table 1.

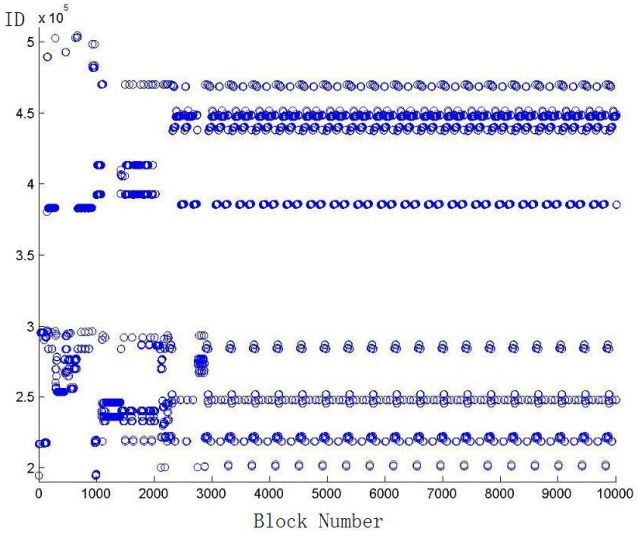

Figure 2. The IDs of the First 10000 Blocks of MCF_REF

TABLE I. TRANSITION OF WORKING SETS DETECTED WITH DIFFERENT THRESHOLDS

Thre.1(%)	Thre.2(%)	Trans.detected
20	25	2
20	30	2
30	35	1
30	40	1
40	45	0
40	50	0
50	55	0
50	60	0

As a result, although 2 transitions of working sets could also make sense, we believe that 1 is more reasonable. So we choose 30% and 35% as the two thresholds.

3.3 Code Cache Initial Size

On the other hand, as the transition of working set might happen at any time, it is quite a waste if we flush the code cache when just a little part of it has been used, say, 10% or 20%, even working set already forms. So since we meant to take part of the size as the initial size, it's a good idea to make it a rule in our strategy that if the initial size has not been full-filled, we would not do the flush job.

It is quite obvious that if we check whether to flush the code cache after it's been used more than a certain percent, the performance should improve for two reasons: on one hand, the flush times could be reduced, so the overhead of this part could be avoid; on the other hand, since the code cache could contain more than one working sets, the formal ones could be reused before they're cleaned, which could save the overhead of translating them again.

The percentage could neither be too small nor too big as the extreme situation may make either size initialization (too small) or working set detection (too big) meaningless. Here, we take the middle value as the result of this point, which is 50%.

3.4 Dynamic-size Code Cache

When we check whether the transition of working set happens after the initial size has been fully taken, if so, a flush job should be done, if not, what should we do to adjust our code cache to working set?

The answer could easily be found as increasing the code cache size since we did not fully take the whole size at the beginning, till the transition happens. But how much should we add as we'll never know when the transition would happen.

Here, we take an experiment in Crossbit, which is tested with three benchmarks in SPEC2000-INT: MCF, BZIP2, GZIP. We use the two conclusions from the two points discussed above: taking two thresholds as 30% & 35%, use the 50% size as the initial one. We run the three benchmarks on Crossbit with three different strategies as we increase the cache size by 5%, 10% and 20% (as 20%, 20%, 10%) each time and test their performance. Compared with the FWF strategy's running time, we get the result as below. (FIG.3)

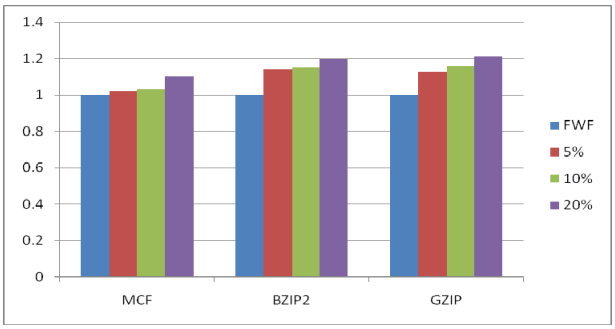

Figure 3. The running time with different increasing grains strategy (Normalized by FWF strategy)

FIG.3 has shown us that the fine-grain-increase choice achieves better results. So in our strategy, we would like to choose 5% to increase our size each time.

3.5 Self-adjust Thresholds

However, different programs have different behaviors; a fixed threshold could not match all the programs. As a result, we try to make our strategy adjustable to the program run on BT system, which processing as follows:

- When we continually flush our code cache 10 times with fully bounded size, meanwhile threshold 1 is reached but no working set transition detected, we believe our threshold is a little bit too high for this program and we minus 2% from both two thresholds.
- When we continually flush our code cache 10 times with fully bounded size, meanwhile threshold is never reached within these 10 times, we add 2% to both two thresholds.

3.6 Replacement Policy

After discussing the points above, our policy becomes more and more clear:

- We initialize our code cache with 50% of the given size in DBT systems.
- As program running on, we keep on recording the translation rate. If it drops below threshold 1, we begin to watch it whether would rise over threshold 2, if so, we set the flag of working set transition true.
- When the initial size is fully taken by blocks, we check the flag in the second step, if it's true, we flush code cache, start over again to record translation rate and do the second step; if not, we apply 5% more of the code cache size, when it's full, we do this step again, until we reach the bound of code cache size.
- If the bound of the size has been reached, we flush the code cache and restart to record translation rate.
- During these processes above, if we continually flush our code cache 10 times with fully bounded size, meanwhile threshold 1 has never been reached, or threshold 1 is reached but threshold 2 is never touched, we would accordingly drop or rise our two threshold by 2%.

4. Evaluation

We evaluate our job by comparing our strategy DCC with the two traditional strategies: FWF and LRU; meanwhile we take the DCC policy without thresholds self-adjusted (named DCC-) as the fourth situation.

The experiment environment is like this: CPU-Intel Core I5(2.66GHz * 4). For DBT system we still choose Crossbit, the programs ran on Crossbit are from SPEC2000-INT: MCF, BZIP2, GZIP, GCC, GAP and TWOLF.

The running time of our experiment is shown in Fig 4. With one of the benchmarks (MCF) is even better than FWF by using DCC, the other benchmarks do not act as good as FWF, not to mention LRU. The time increased is up to 15% at most. These data is still within our imagination; one thing is strange enough: with

the self-adjust thresholds, the running time is even longer. This part of job does not seem work in the right way, the reason is that our strategy is a little bit too sensitive to the running environment, the thresholds change too frequently. We shall put it as our future work.

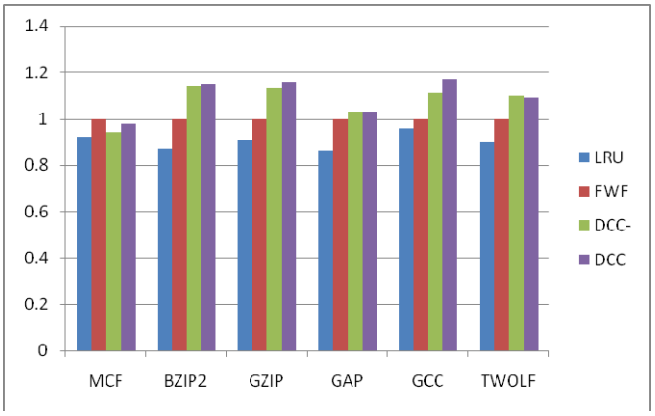

Figure 4. The running time with different replacement strategy (Normalized by FWF strategy)

Though seeing from most of the benchmarks, the performance of running time is not as good as the traditional strategy, the space saved is quite considerable:

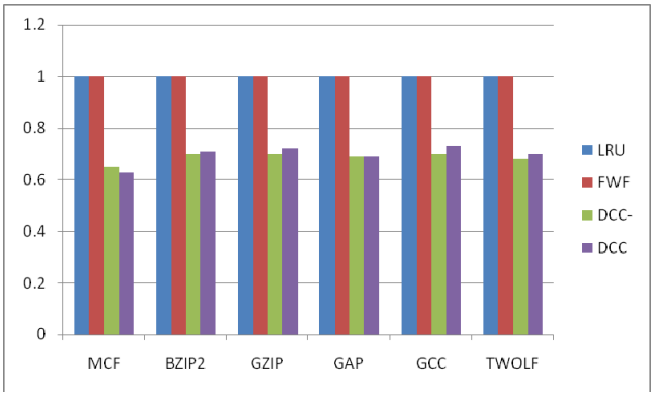

Figure 5. Space Used by Different Strategy (Normalized by FWF Strategy)

These data is calculated by the space used when we flush our code cache, say, 80%, then we count the total times when the code cache is flushed with 80% full. By adding all the situations together (100%, 90%, etc.), we divided by the total flush times, then we get the value above.

Though scarifying 9%time on average, we save 32% of space. Suppose our job is done in a space-consuming environment, our running time would possibly be better than the current situation.

5. Conclusion and Future work

As we could see from the last section, we scarify 10%-15% of the running time to get 28%-32% of the space. We could imagine that our experiment is done by a very free environment, no other processes are demanding for the memory, too. If this situation happens, we might get better running time compare to other strategy.

It's worth mentioning that for MCF, we could even run faster than FWF strategy. But it's quite strange our self-adjusting threshold does not work.

LRU does not get the great result as expected just because in DBT systems, LRU should be carried out with de-linking job, which is quite time-consuming.

Our future job is to:

- Simulate the memory-lacking situation and see how our job works
- Optimize the self-adjusting threshold

6. Acknowledgment

This work was supported by The National Natural Science Foundation of China (Grant No.60773093, 60873209, 60970108, 60970107, 61073151), Shanghai Municipal Natural Science Foundation（Grant No. 10ZR1416400），IBM SUR Funding and IBM Research-China JP Funding. We would particularly thanks Hanbo Xiao for giving important ideas for our job.

7. References

[1] Ebcioglu, K., Altman, E., Gschwind, M., Sathaye, S., “Dynamic binary translation and optimization” Computers, IEEE Transactions on P.529 - 548

[2] W. A. Wong and J.-L. Baer. Modified LRU policies for improving second-level cache behavior. In Proceedings of the 6th International Symposium on High Performance Computer Architecture, 2000.

[3] Peter J. Denning, The working set model for program behavior, Communications of the ACM, v.11 n.5, p.323-333, May 1968

[4] B Yuncheng, L Alei, G Haibing, “Design and implementation of crossBit: dynamic binary translation infrastructure”, Computer Science 2007

[5] Sanjeev Banerjia, Vasanth Bala, Evelyn Duesterwald, “Preemptive replacement strategy for a cacheing dynamic traslator” USA Patent, No.US6,237,065 B1

[6] Asit Dan , Don Towsley, An approximate analysis of the LRU and FIFO buffer replacement schemes, ACM SIGMETRICS Performance Evaluation Review, v.18 n.1, p.143-152, May 1990