DOI: 10.2298/CSIS100327022M

Code Cache Management Based on Working Set in Dynamic Binary Translator

Ruhui Ma, Haibing Guan*, Erzhou Zhu, Yongqiang Gao, and Alei Liang

Shanghai key laboratory of scalable computing and systems,
Department of Computer Science and Engineering,
Shanghai Jiao Tong University, Shanghai, 200240,China,
{ruhuima, hbguan, ezzhu, yongqianggao, liangalei}@sjtu.edu.cn

Abstract. Software code cache employed to store translated or optimized codes, amortizes the overhead of dynamic binary translation via reusing of stored-altered copies of original program instructions. Though many conventional code cache managements, such as Flush, Least-Recently Used (LRU), have been applied on some classic dynamic binary translators, actually they are so unsophisticated yet unadaptable that it not only brings additional unnecessary overhead, but also wastes much cache space, since there exist several noticeable features in software code cache, unlike pages in memory. Consequently, this paper presents two novel alternative cache schemes—SCC (Static Code Cache) and DCC (Dynamic Code Cache) based on working set. In these new schemes, we utilize translation rate to judge working set. To evaluate these new replacement policies, we implement them on dynamic binary translator—CrossBit with several commonplace code cache managements. Through the experiment results based on benchmark SPECint 2000, we achieve better performance improvement and cache space utilization ratio.

Keywords: Code cache management, Working set, Replacement strategy, Code block, Bounded code cache.

1. Introduction

Dynamic binary translation system, as a relocation tool for executable code, has been applied both in electronic commodity and in research domain for decades. The ability to manipulate the instruction stream of an executing program enabled by [1] these systems has had numerous implications in program performance, security, and portability [1]. In order to relieve the system overall overhead, researchers have attempted many algorithms or methods, where a significant improvement method| making use of code cache [26] is able to upgrade better performance. Indeed, it caches lots of

[*]Corresponding Author

translated or optimized code to be reused for system, yet this amortizes the cost of expensive retranslation time over the entire system program execution time. Substantively, this process increases the locality of stored code and code specialization, so program execution speed can be achieved remarkably.

Software code cache is virtually a segment of sequential memory space to store altered copies of original program instructions from low address to high address in order, which should have low overhead, good temporal locality, and minimal fragmentation. Some commercial virtual machines [14-16] still employ unbounded code cache to cache translated basic block or optimized code (i.e. superblocks) to extremely relieve extra retranslation overhead without any management overhead. In [13], Kim has explained that as the size of new huge software released grows, the size of corresponding unbounded code cache [9] will grow proportionately or exponentially. That is to say, unbounded code cache is out of state. Especially, in embedded system, the memory space must be reasonably utilized, rather than be wasted. Currently, though many dynamic binary translators, such as Strata [2], Walkabout [3], and UQDBT [22], have already employed bounded code cache [25], the utilization rate of code cache space is so low that more space cannot be fully used. In DBT system, 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, it 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.

Unlike pages in memory, blocks in code cache have unfixed sizes each other, and linking between them is also considered, making the prediction imprecise. So LRU algorithm, as the conventional replacement policy, causes so many fragmentations that more code cache space will be wasted, if it is applied on software code cache. Furthermore, the simple code cache management, like pure Flush replacement policy, is widely employed in code cache, especially leading no fragmentation. Attentively, Flush algorithm only clears all blocks when cache is full, without considering the program behavior. This causes that extra retranslation overhead would exponentially increase, that is, thrashing [5] usually rises, since some frequent-executed codes are repeatedly flushed and regenerated in code cache. So when many resource-consuming applications are running on the same physical machine, to promise high utilization rate of code cache space and system overall performance, a novel code cache management is needed.

Working set gives us another chance. In DBT system, working set means the collection of the most recently used blocks of the program in the software code cache, which reflects the program behavior as well. In this paper, we present a novel code cache management---SCC (Static Code Cache) based on working set, which outperforms Flush policy since the new one executes alternative policy according to working set. Although some performance improvement can be achieved, associated with SCC policy, extra unused code cache space is vacant. Especially, in memory constrained embedded devices such as cellular phones, global positioning units and medical devices, vacant code cache space must be avoided. So we also propose a new dynamic replacement policy---DCC (Dynamic Code Cache) based on SCC policy. Indeed, DCC policy mainly inclines to reduce the memory requirement, enhance the utilization rate of code cache space with little or no performance sacrifice. 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. 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, the flushing operation is trigged to clear code cache. 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. In particular, the novel contributions of our work are:

In this paper, a working-set-based replacement policy---SCC is proposed firstly. This one in the aspect of performance outperforms Flush policy, due to execution flow following program behavior. In addition, self-adjusting threshold used to decide working set is analyzed.

To efficiently utilize code cache space, another novel replacement policy---DCC based on SCC policy is presented, associated with little performance victimization that can be ignored.

The remainder of this paper is organized as follows. Section 2 reviews the related work. Then section 3 describes the features of code cache and working set in DBT system. In addition, section 4 introduces SCC policy, the method how to judge working set in DBT system, and gives qualitative analysis of SCC. In addition, in section 5, we introduce DCC policy with several key points. Section 6 shows the experimental results about SCC and DCC, and gives the detailed analysis. Finally, we conclude the paper in section 7.

2. Related work

In this section, we will introduce some conventional dynamic translation or optimization systems how to take advantage of code cache, and discuss existing cache management utilized in various systems. Then we also simply exploit the advantage of our method relative to others.

2.1. Traditional code cache management

In 1996, dynamic binary translator---DAISY [15] developed by IBM, dynamically translates from PowerPC binaries to VLIW instruction codes. Based on this project, in 2000, BOA framework [16] presented in IBM allows PowerPC code to execute on a VLIW/EPIC processor. The Transmeta Crusoe processor [14] shipped with Code-Morphing Software (CMS) executes binary translation from IA-32 to an underlying proprietary VLIW architecture, which is the first commercial processor authentically integrated with binary translation technique. However, powerful code cache management policies are not definitely characterized in these dynamic binary translators.

Dynamo [17], as a transparent dynamic optimizer developed from HP Labs in 1999, executes on HP-UX OS providing an efficient software-management policy for code cache. It caches superblocks in code cache and takes advantage of preemptive flush mechanism as code cache policy. Virtually, this alternative management is trigged by program phase change detected. A follow-up infrastructure is DELI [18], which is a VLIW version of Dynamo developed by Hewlett-Packard in conjunction with ST Microelectronic. The code cache management employed in DELI is a special-flush cache policy controlled by user, that is, a passive flush cache policy. As the successor of Dynamo, DynamoRIO [19] is a excellent dynamic optimizer developed by Hewlett-Packard and MIT. The attractive feature of it is that it can execute on IA-32 architecture not only in Linux but also in Windows. DynamoRIO partitions the unified code cache into two independent-distinct code caches employed to cache basic code blocks and superblocks respectively, yet the superblock cache is a thread-private cache. The cache replacement management in DynamoRIO is approximate unavailable, that is, none of evictions would happen, due to unbounded code cache used to store all translated or optimized codes. Mojo [20] exploited by Microsoft, which is targeted Windows NT running on IA-32, is able to execute several large desktop applications. It also has two code caches---a thread-private basic code cache and a thread-shared trace cache, which is managed in a heavyweight manner by suspending all other threads. With regard to code cache management, each cache is subdivided into two units. For each unit, it would be flushed in special order (i.e. FIFO, LRU) when filled with codes. This leads to complicated cache management to ensure synchronism between threads due to shared code. In 2004, a novel dynamic binary translator---DigtalBridge [21] developed by Institute of Computing Technology, Chinese Academy of Science, is able to execute from X86 to MIPS infrastructure on Linux OS. Specially, its cache management differs from others, that is, several equivalent units (space size) are attained via partitioning unified cache. The cache management for DigtalBridge is deemed as a combination policy with Flush, FIFO, and LRU. But the situation where fragmentation still embarrasses overall performance, needs to be concerned as well.

Strata [2] and Walkabout [3] are research infrastructures for dynamic binary translation that are specifically designed to be retargetable. Strata has been retargeted to run on SPARC, MIPS, and IA-32 architectures. Walkabout, which was based on UQDBT [22], has been retargeted to execute on both SPARC and IA-32 architectures. CrossBit [7], is a resourceable and retargetable DBT system with intermediate representation (IR). Until recently, it has fully or partially supported guest platforms including SimpleScalar, IA32, MIPS, SPARC, and has fully supported the IA32 host platform. Another RISC instruction sets platform host is on the plan, for instance, PowerPC and SPARC. HDTrans [23] is a simple fast Linux-based binary translator. Its simplicity speeds up its cold code translation performance and it shows competitive performance among DBT systems that do not optimize hotspots. StarDBT [24] is a multi-platform translation system that is capable of translating application level binaries on either Windows or Linux OSes. However, the code cache management policy in these systems is to flush the entire code cache when it becomes full.

2.2. Our work

As we know, the traditional replacement strategies, such as FIFO, LRU and Flush, have been widely used in operating systems. However, due to the unequal size of each block in DBT system, the traditional strategies used in code cache might encounter some problems which would not happen in OS, such as the fragmentation, and de-linking, etc, especially causing cache space waste. As a result, they may not achieve their expected performance in DBT.

In this paper, we define working set as the set of blocks that run recently. Finally, according to working set detected, DBT system can automatically adjust its code cache space, and this avenue saves more cache space for memory-consuming applications. That is to say, it enhances the utilization rate of code cache.

3. Background

3.1. Features of software code cache

Compared with physical memory, software code cache has its explicit challenges that directly impact overall system, mainly focusing on its cached code blocks.

Unfixed-sized cached codes. The significant feature of software code cache that differs from traditional hardware cache is that the size of stored codes (i.e. translated basic block or superblock) is not fixed but variable. This conduces that when replacement algorithm used in code cache takes place (i.e. LRU), fragmentation will appear in code cache. To minimize fragmentation or even avert fragmentation, compression is able to compact fragmentation so that extra space is to be reused, but it is too expensive for system to implement it during execution. In this process, it is necessary to revise all of the branches, for each branch links one code to another congener code (In general, codes are classified into two groups| basic block and superblock). We can see that fragmentation obtained as a byproduct when some replacement algorithm being trigged, drastically affects overall performance, so avoiding fragmentation or lowering the amount of fragmentation (decreasing compression overhead)must be taken into account when selecting powerful replacement algorithm.

Linking repair. Linking is an optimizing method implemented on all the basic blocks through modifying the machine codes after they have been executed for once [8]. All the superblocks need to be linked after it created as well. The essence of linking for basic block or superblock is that inserting jump instruction into the bottom of each code block sacrifices space size to exchange less time. Through linking between code blocks, execution from one code block to another is performed in succession rather than transforming control to system to again determine the next executable code block. This leads to a better performance. However, when replacement other than flush occurs, eviction of code blocks in code cache will bring dangling linking that causes incorrect program execution. Since one code block has several incoming and outgoing linking, how to efficiently and reasonably cope with these linkings is critical for system performance. To ensure program execution correctness, it is easy to evict outgoing linking with code block being replaced relative to incoming linking. While conventional methods to settle this embarrass situation where incoming linking related to the candidate of evicted code block should be disposed immediately, is to build a back-pointer table. It stores incoming and outgoing linking information of each code block. When replacement policy is trigged, the system will firstly lookup this table to acquire the incoming linking of eviction candidate. Then these incoming linkings of the candidate code block will be evicted. In fact, this process can carry extra run-time overhead due to lookups and occupying memory space.

Retranslation overhead. Code cache miss, as a ubiquitous problem to leave a high retranslation overhead, cannot be fully avoided, yet is only attempted to minimize occurrence frequency to some extent. The high retranslation overhead results from a series of successive program execution behaviors. That is to say, this process is that storing context information about running program, regeneration of the previously cached code, copying it into code cache, updating hash table and linkings, and restoring context information about running program and transferring control. We can see that this process is so complicated that more run-time overhead will naturally appear.

In conclusion, conventional replacement scheme---LRU cannot be adequately applied on the code cache. Though Flush clears all the block stored in the code cache to avoid additional repair overhead mentioned above, it doesn't take program behavior into account, leading to more cache misses, but with excessive cache space.

3.2. Working set in DBT system

In traditional OS, a program's working set W(*t*, T) is the set of distinct pages at time *t* among the T most recently referenced pages. Intuitively, it is the smallest subset of its pages that must reside in main memory in order that the program operates at some desired level of efficiency. The working-set principle of memory management states that a program may use a processor only if its working set is in main memory, and that no working-set page of an active program may be considered for removal from main memory [11].

In DBT system, this principle is in practice as well. In this paper, we would rather define working set as the set of blocks that run recently [10]. 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. During this period of time, even if 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.

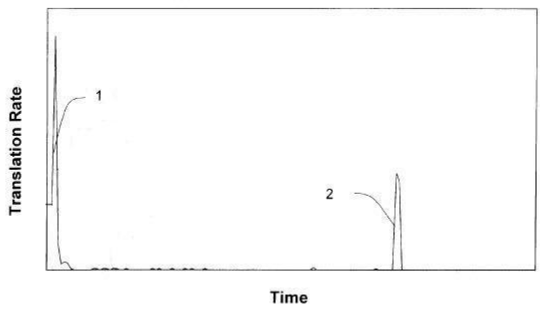

Fig. 1. Working sets detected according to translation rate

From Fig.1, we can easily see that there are 2 working sets during period of execution time, achieved by inaccurate skin-deep partitioning method. In a word, the execution flow of benchmark shows better temporal and spatial locality, and this execution process is also considered as the alternately execution of working sets.

4. Static Code Cache

In this section, we will introduce the code cache management---SCC (Static Code Cache), based on working set, which can achieve better performance than traditional replacement policy---Flush, without too much cache space. It promises the overall performance and saves cache space, which is adequately adapt to manipulate code cache compared to conventional code cache replacement policy.

4.1. Judging criterion for working set

The theoretic. In this paper, we use the translation rate to decide the working set. Here, we define translation rate---T_{rate}:

$$T_{rate}=N_{TranslationBlock} \,/\, N_{ExecutionBlock} * 100\% \qquad (1)$$

In formula (1), T_{rate} represents the running program's translation rate. $N_{TranslationBlock}$ is deemed as the number of translated blocks stored in code cache, and contrarily, $N_{ExecutionBlock}$ represents the number of executed blocks.

An accompanying observation is that an increase in the rate at which translations are created---the translation rate---is often a precursor to an increase in the proportion of time spent executing within the code cache. It then follows that as the proportion of code cache execution time increases, the translation decreases. A high translation rate indicates that the translator is creating a set of translations that will be executed in the near future.

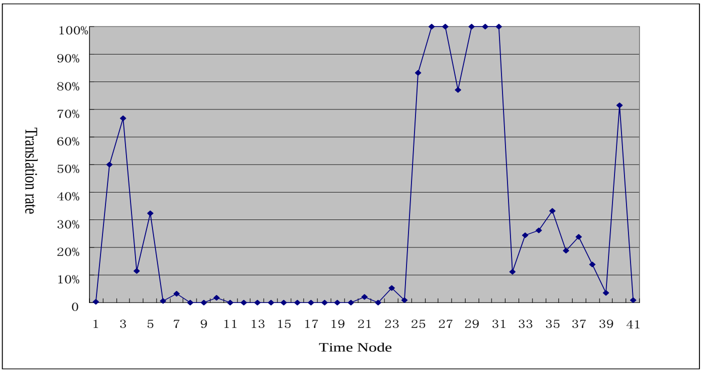

Fig. 2. Working sets detected in MCF

The set of translations can be termed the upcoming working set of the program since it represents the code that will perform the program's work in the upcoming phase of execution. A low translation rate indicates that the current working set has been captured in the cache and thus execution is occurring primarily from the cache. For a program whose behavior is characterized by the execution of different portions of code across distinct phases, the translation rate follows a regular pattern. The translation rate increases when translations for the (upcoming) working set are being constructed, decreases and remains low as the working set executes, increases again when the next working set is constructed, and so on. This phenomenon is described in Fig.1.

In Fig.1, the rising side of the leftmost peak shows the translation rate increasing as a new working set is built in the cache. The trailing side of that peak shows the translation rate decreasing as formation of a working set nears completion. The period between the peaks shows a translation rate remaining relatively low as most of the execution occurs in the working set stored in the cache. The next peak shows that the program is entering a new phase of execution: code that has not been stored in the cache is needed for execution; another working set is being formed. This principle in general purpose program is also work, such as mcf, and this is depicted in Fig.2.

The value of thresholds. Through the description of the theoretic, we can see that the key point for judging working set in DBT system is to find the value of the two thresholds: the threshold1 and threshold2 indicated in Fig.1. These two values could determine whether we could get the right working set or not. It could be easily told that if threshold1 is set too low, the whole program might just be only one working set as a whole; if it's set too high, as threshold2 must be higher than it, the next working set might never come. Threshold2 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 SPECint 2000: MCF, as the test program. Firstly, we record the block ID of the first 1750 blocks of MCF. As the program is running, the same block would be executed again. That is, in a period of time, the translated code blocks stored in code cache can be reused. In Fig.2, many working sets exist, which keep to the principle mentioned above, and there are many transition points used to judge working set. Through this experiment, we can find different working sets easily, but how to divide working set accurately according to many transition points is the key problem. So according to the transition point of two working sets, we can achieve the accurate thresholds through the following experiments in Table 1.

Table 1. Transition points of working set detected with different thresholds

Threshold1	Threshold2	Working sets transition number
20	25	2
20	30	2
30	35	1
30	40	0
40	45	0
40	50	0
50	55	0
50	60	0

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 based on formula (1) are shown in Table 1. In a period of time, we expect that only one working set can be detected according to Threshold1 and Threshold2. But if more than one working sets or 0 working set is detected, that illuminates that the thresholds are not competent for this system. So in this system, we choose 30% and 35% as the two thresholds.

4.2. 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, and meanwhile threshold1 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 threshold1 is never reached within these 10 times, we add 2% to both two thresholds.

This self-adjust method is so efficient that the thresholds are flexible when facing different programs.

4.3. Static Code Cache

Compared to Flush policy, SCC replacement policy is based on the transition of working set to do flush operation, rather than code cache filling with translated blocks. That is to say, Flush is a passive flush algorithm, yet SCC is a active flush algorithm. Due to different alternative condition between the two strategies, the performance of them is impacted as well. With tradition replacement policy---Flush, the code cache may store lots of unexecuted code blocks in a long time, and that will bring extra space waste. However, the novel policy---SCC can actively flush the unnecessary code blocks timely, according to program behavior, so that is advantage for the system to save more space. Then we will give the qualitative analysis about performance and space in detail.

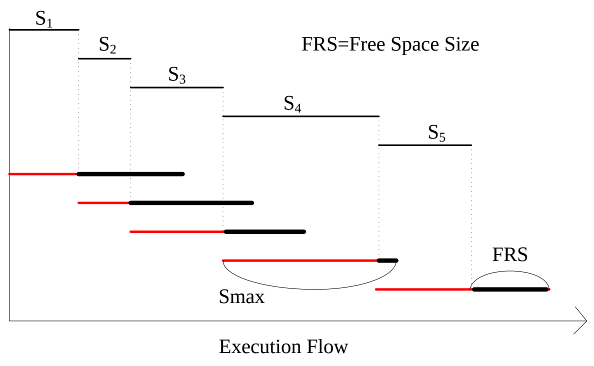

Fig. 3. Difference between Flush Policy and SCC Policy

Here, we define the size of code cache is *Smax*. Suppose that one source program is executed through 5 working sets: WS_1, WS_2, WS_3, WS_4, WS_5. The size of each working set mentioned above is respectively: S_1, S_2, S_3, S_4, S_5, where *Smax* > S_i (i=1,2,3,4,5). And the same source program is also executed with Flush policy. The comparison of them is depicted in Fig.3. If Flush replacement policy is selected, when code cache is full of translated code blocks, the code cache will be adequately cleared without considering program behavior. That leads that more than one working set will be flushed, where there exists one working set being constructed. Indeed, some code blocks not executed in a period of time, still stored in the code cache, yet those blocks occupy so much cache space. If we choose SCC replacement policy, the active flush method is applied on code cache, rather than passive one. The flush condition is altered to working set transition, and this conforms to program behavior, which has been proved in section 4.1. So in Fig.3, extra free space (overstriking black beeline showed in Fig.3) can be saved when code cache fully flushed. Indeed, the overall space utilization with SCC policy is total of working sets' size, that is, $\sum S_i$ (i=1,2,3,4,5), and that is further smaller than 5 * *Smax* (When passive Flush policy is employed, all the space utilization is 5 * *Smax*).

5. Dynamic Code Cache

Although SCC policy can save extra free space compared to passive Flush policy, the space saved cannot reused by other applications. Consequently, we propose another novel replacement policy---DCC (Dynamic Code Cache), which gives other applications another chance to reasonably utilize extra free cache space. In addition, if most of working sets in the executed program are too small, this leads that only a few code cache space is utilized. So, to efficiently utilize code cache space compared to SCC policy, we set another parameter, S_{part}, code cache initial size in DCC algorithm, corresponding with system initial size.

5.1. Code cache initial size

On the one hand, the saved code cache space cannot utilized by other applications, associated with SCC replacement policy. On the other, 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 (i.e. 10%, 20%) has already been used to form working set. So we propose another code cache parameter--- S_{part}, which is to take part of the code cache initial size as the initial size, and it is a good idea to make it a rule in our strategy. If the S_{part}-sized code cache 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: the flush time could be reduced compared to SCC replacement policy, so the overhead of this part could be avoid; 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 of space size could be neither too small nor too big. That is because the extreme situation may lead that frequently adjusting thresholds used to judge working set (too small) or more working sets simultaneously cached in code cache (too big). Here, we take the middle value as the result of this point, which is 50%.

5.2. Dynamic-size code cache

When we check whether the transition of working set happens after the S_{part}-sized code cache has been fully taken, if so, a flush job should be done; if not, what should we do? or 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, and the increasing size of code cache S_{add} will be discussed in section 6.

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

- We initialize our code cache with S_{part} of the given size in DBT system.
- As program running on, we keep on recording the translation rate---T_{rate}. If it drops below *Threshold1*, we begin to watch it whether would rise over *Threshold2*, 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 is true, we flush code cache, start over again to record translation rate and do the second step; if not, we apply S_{add} more of the code cache size, when it is 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 *Threshold1* has never been reached, or *Threshold1* is reached but *Threshold2* is never touched, we would accordingly drop or rise our two threshold by 2%.

6. Evaluation and analysis

To evaluate this novel code cache management in DBT system, we have applied it on original DBT system---CrossBit, which is a resourceable and retargetable DBT system with IR [7]. That is also a large basic research platform, based on the CrossBit, and research works are extended, such as multi-core technique, code behavior analysis, mobile computing for Thin-client in heterogeneous resource, distributed virtual execution strategy, defense for anomaly attack, and swam intelligent, etc. In addition, SPECint 2000 [12] is selected as the test benchmark. And the configure of physical machine is that: CPU---Intel Core I5 (2.66GHz * 4), 8GB memory with Linux kernel version 2.6.33.4. The size of code cache---S_{max} is assigned about 32 KB (Since our code cache management will be extended on embedded system in future, the size of code cache assigned is so small). The traditional replacement strategies---Flush and LRU are employed to do comparison with SCC and DCC. We use the conclusion from section 5.1: using the 50% size as the initial one for code cache (S_{part} is 16 KB).

First, we do the experiment to test difference between SCC policy and Flush policy. In this experiment, the *Threshold1* and *Threshold2* are initialized as 30% and 35%, respectively. In addition, according to different programs, the thresholds will be adjusted by themselves. The results about their performance are shown in Fig.4.

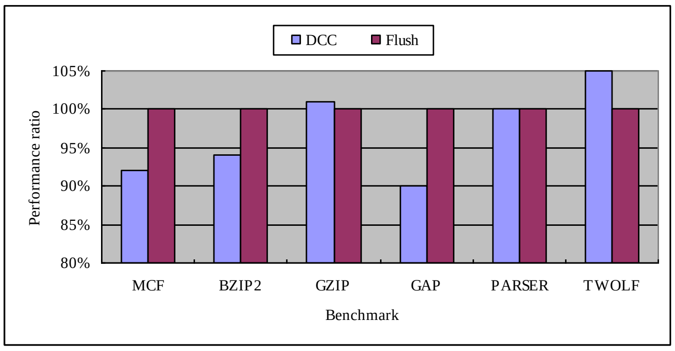

Fig. 4. Relative Runtime of Benchmarks using DCC and Flush (Normalized by Flush policy)

From Fig.4, we can see that the performance with SCC policy significantly outperforms that of Flush policy, and the average performance improvement is about 3%. But GZIP and TWOLF benchmarks are the victims. The reason why GZIP and TWOLF undergo lower performance is that: since the thresholds used to create working set is decided by T_{rate}, the translation rate---T_{rate} is also determined by $N_{TranslationBlock}$ and $N_{ExecutionBlock}$ according to formula (1). To accurately get them, profile instructions should be added into each code blocks to record corresponding information. Here, we give the execution time of code blocks in Table 2.

Table 2. Execution time of each benckmark

Benchmark	Execution time
MCF	53011039
BZIP2	530628945
GZIP	1948235411
GAP	48558303
PARSER	238666896
TWOLF	1131632548

In Table 2, the execution time of GZIP and TWOLF are so many that the overhead caused by profile instructions is relatively increased, so the performance is lower naturally. In addition, thrashing occurring in code cache, brings inaccurate working set, leading lower performance.

Although the performance improvement is achieved associated with SCC policy, the extra free space is still vacant. So, we propose DCC policy for other applications to further efficiently utilize saved code cache space. In DCC policy, we should define the increasing grain when S_{part}-sized code cache fills with translated blocks, and this point is discussed in section 5.2. We run the six benchmarks on CrossBit with two different strategies as we increase the cache size by 5% (fine-grain), 10% and 20% (coarse-grain) each time and test their performance. Compared with the Flush strategy's running time, we get the result in Fig.5.

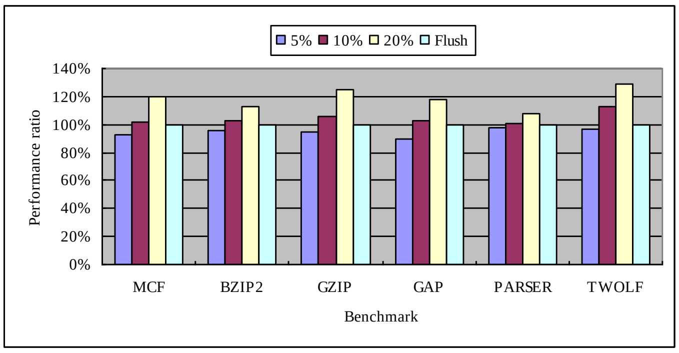

Fig. 5. The running time with different increasing grains (Normalized by Flush policy)

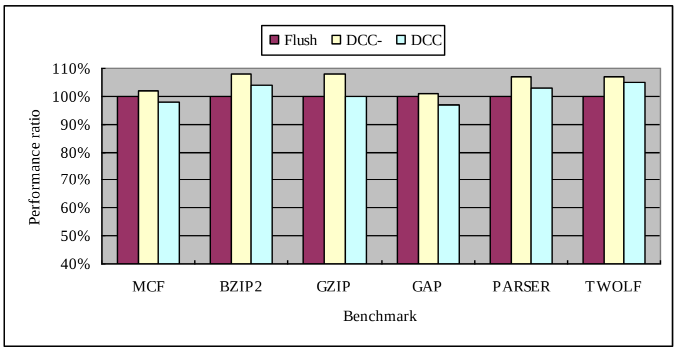

Fig. 6. The performance with different replacement policies (Normalized by Flush policy)

In Fig.5, fine-grain increasing (5%) mode is adapt to dynamic binary translation system, which is better than other increasing modes. Indeed, the remain space of code cache can be reused by other applications. Since fine-grain increasing mode can save more space for other applications, the corresponding performance outperforms that of others as well. Here, we select 5% increasing grain as the increasing mode when initial space filling with translated blocks.

Furthermore, we apply several replacement policies on the CrossBit to test the system performance, such as Flush, DCC, and DCC- (DCC- is the DCC policy without thresholds self-adjusted).

In Fig. 6, we can see that the original replacement policy (i.e. Flush) outperforms the novel one presented by us, especially DCC-. And DCC policy outperforms DCC- a little. Indeed, either DCC or DCC- spends time dynamically adjusting code cache, but the conventional replacement policy only considers space size without any adjustment on code cache. So the performance caused by DCC and DCC- is lower than the original one. The distinction performance between DCC and DCC- is mainly from the accurate thresholds used to decide working set. The more accurate working set is, the better performance improvement system can be achieved. In DCC-, invariable thresholds will cause inaccurate working set, and the reason is analyzed in section 4.2. By the way, the decreasing performance with DCC policy is about 2%-5%, compared to Flush policy, since extra code cache space occupied by other applications cannot retrieved for code cache timely. Though the new replacement policy---DCC causes some performance degradation (average of that is about 1.17%), extra code cache space is saved that can be used by other applications. In embedded system, reasonably utilizing finite memory space (code cache space) will achieve another performance improvement or promise adequate function for different applications.

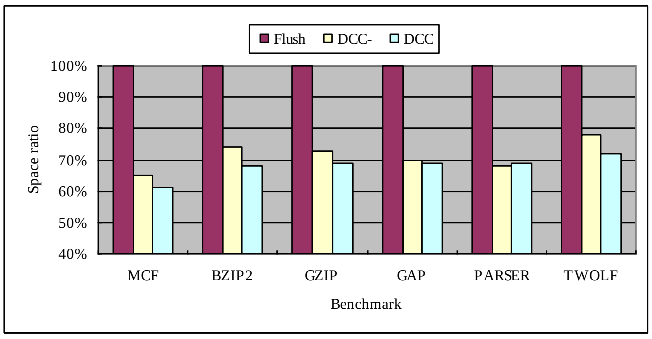

Fig. 7. The space used by different replacement policies (Normalized by Flush policy)

Next, we do the experiment to test memory space utilization. When system initializing, 32 KB memory space is assigned to code cache. Especially, the code cache with Flush replacement policy constantly occupies this section of memory, so other applications cannot preempt it. However, the novel replacement algorithms---DCC and DCC- don't occupy overall assigned memory space all the time. That is to say, the unused memory space assigned to code cache can be utilized by other applications, and it improves space utilization rate. The results is depicted in Fig.7.

Fig.7 depicts space utilization rate of each replacement policy. The simple replacement policy---Flush only considers that whether the cache space fills with translated code blocks, rather than program behavior and space utilization rate. About space utilization, the novel policies---DCC- and DCC are significantly outperform the original one, since they follow program behavior to do replacement (the replacement unit is working set). That is to say, extra free code cache space can be reused by other applications. From Fig. 7, the space utilization of DCC strategy is better than that of DCC-, since self-adjust thresholds impact the final space utilization rate. In DCC-, fixed thresholds that used to judge working set is not flexible adequately, so flushing operations with fully bounded size is more than that of DCC.

In conclusion, this new replacement policy can save extra code cache space used by other applications with a little performance improvement. As well, compared to traditional replacement policy---Flush, the novel one has more advantage, such as better performance and saving extra memory space.

7. Conclusion and future work

Code cache management in dynamic binary translation system is deemed as a crucial yet intractable issue. The high cost of preparing translated basic blocks and superblocks inserted into a code cache has incurred many researchers to slide over this serious issue through implementing either an larger code, such as merging several code blocks, or an unsophisticated replacement scheme, such as flush policy. However, the traditional replacement policies cannot fully adapt to dynamic binary translation system. So in this paper, based on working set, a novel replacement policy---SCC is proposed. The performance of it outperforms the original one---Flush, due to it considering program behavior, but extra free code cache saved is still vacant. Then we present another new replacement policy---DCC, which is not in light of program behavior, but also make saved code cache space to be reused by other applications. Unfortunately, the system performance with DCC policy is decreased. Thus, our future work is to improve its performance, and implement our method on other dynamic binary translation systems.

Acknowledgment. This work was supported by the National Natural Science Foundation of China (Grant No. 60970108, 60970107), the Science and Technology Commission of Shanghai Municipality (Grant No. 09510701600, 10DZ1500200, 10511500102), IBM SUR Funding and IBM Research-China JP Funding. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the presentation.

References

1. Hazelwood, K., Cohn, R.: A Cross-Architectural Interface for Code Cache Manipulation. In Proceedings of the 4th International Symposium on Code Generation and Optimization. Manhattan, New York, USA, 17-27. (2006)
2. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J. W., Soffa M. L.: Retargetable and reconfigurable software dynamic translation. In Proceedings of the 1st International Symposium on Code Generation and Optimization, San Francisco, California, 36-47. (2003)
3. Cifuentes, C., Lewis, B., Ung, D.: Walkabout-A retargetable dynamic binary translation framework. In Proceedings of the 4th Workshop on Binary Translation, Charlottesville, Virginia, 22-25. (2002)
4. Ung, D., Cifuentes, C.: Machine-adaptable dynamic binary translation. In Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, Boston, Massachusetts, 41-51. (2000)
5. Tannenbaum, A.: Modern Operating Systems, Second Edition. Prentice Hall, New Jersey. (2001)
6. Source codes and Introduction of CrossBit (2005). [online]. Available: http://sourceforge.net/projects/CrossBit/
7. Shi, H. H., Wang, Y., Guan, H. B., Liang, A. L.: An intermediate language level optimization framework for dynamic binary translation. ACM SIG/PLAN Notice. Vol. 42, No. 5, 3-9. (2007)
8. Hiser, J. D., Williams, D., Hu, W., Davidson, J. W., Mars, J., Childers, B. R.: Evaluating indirect branch handling mechanisms in software dynamic translation systems. In Proceedings of the 5th International Symposium on Code Generation and Optimization, San Jose, CA, USA, 61-73. (2007)
9. Bruening, D., Garnett, T., Amarasinghe, S.: An infrastructure for adaptive dynamic optimization. In Proceedings of the 1st Annual International

Symposium on Code Generation and Optimization, San Francisco, California, 265-275. (2003)
10. Banerjia, S., Bala, V., Duesterwald, E.: Preemptive replacement strategy for a cacheing dynamic traslator. USA Patent, No.US6,237,065 B1. (2001)
11. Denning, P., Schwartz, S. C.: Properties of the working-set model. Communications of the ACM. Vol. 15, 191-198. (1972)
12. SPEC CPU2000 Documentation (2010). [online]. Available: http://www.spec.org/osg/cpu2000/docs/
13. Hazelwood, K., Smith M. D.: Code Cache Management Schemes for Dynamic Optimizers. In Proceedings of the 6th Annual Workshop on Interaction between Compilers and Computer Architectures. Cambridge, MA, 102-110. (2002)
14. James, C. D., Brian, K. G., John, P. B., Richard, J., Thomas, K., Alexander, K., Jim M.: The transmeta code morphing^{TM} software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In Proceedings of the 1st International Symposium on Code Generation and Optimization. San Francisco, California, USA, 15-24. (2003)
15. Ebcioglu, K., Altman, E. R.: DAISY: Dynamic complication for 100% architectural compatibility. In Proceedings of the 24th International Symposium on Computer Architecture. Denver, Colorado, 26-37. (1997)
16. Altman, E. R., Gschwind, M., Sathaye, S., Kosonocky, S., Bright, A., Fritts, J., Ledak, P., Appenzeller, D., Agricola, C., Filan, Z.: BOA: The architecture of a binary translation processor. IBM Research Report RC 21665. (1999)
17. Bala, V., Duesterwald, E., Banerjia, S.: Dynamo: A Transparent Runtime Optimization System.In Proceedings of the 4th International conference on Programming language design and implementation. Vancouver, British Columbia, Canada, 1-12. (2000)
18. Desoli, G., Mateev, N, Duesterwald, E., Faraboschi, P., Fisher, J. A.: Deli: A new runtime control point. In Proceedings of the 35th International Symposium on Microarchitecture. Istanbul, Turkey, 257-268. (2002)
19. Bruening, D., Duesterwald, E., Amarasinghe, S.: Design and implementation of a dynamic optimization framework for Windows. In Proceedings of the 4th ACM Workshop on Feedback-Directed and Dynamic Optimization, Austin, Texas. (2001)
20. Chen, W. K., Lerner, S., Chaiken, R., Gilles D. M.: Mojo: a dynamic optimization system. In Proceedings of the 3rd ACM Workshop on Feedback-Directed and Dynamic Optimization. Monterey, CA, 81-90. (2000)
21. Bai, T. X., Feng, X. B., Wu, C. G., Zhang, Z. Q.: Optimizing Dynamic Binary Translator in DigitalBridge. Journal of Computer Engineering, Vol.31, No.10, 103-105. (2005)
22. Ung, D., Cifuentes, C.: Machine-adaptable dynamic binary translation. In Proceedings of ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization. Boston, MA, 30-40. (2000)
23. Swaroop, S., Jonathan S. S., Prashanth P. B.: HDTrans: An Open Source, Low-Level Dynamic Instrumentation System. In Proceedings of the 2th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Ottawa, Ontario, Canada, 175-185. (2006)
24. Wang, C., Hu, S., Kim, H., Nair, R. S., Breternitz, M., Ying, Z. W., Wu, Y. F.: StarDBT: An Efficient Multi-platform Dynamic Binary Translation System. In Proceedings of the 12th Asia-Pacific Computer Systems Architecture Conference. Korea, 4-15. (2007)

25. Baiocchi, J. A., Childers, B. R., Davidson, J. W., Hiser, J. D.: Reducing pressure in bounded DBT code caches. In Proceedings of International conference on Compilers, architectures and synthesis for embedded systems. Atlanta, GA, USA, 109-118. (2008)
26. Bruening, D., Kiriansky, V.: Process-shared and persistent code caches. In Proceedings of the 4th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments. Seattle, WA, 61-70. (2008)

Ruhui Ma is currently a Ph.D. candidate at Shanghai Jiao Tong University, China. He received the B.S. and M.S. degrees at School of Information and Engineering from Jiangnan University in 2006 and 2008, China, respectively. His main research interests are in virtual machines, computer architecture and compiling.

Haibing Guan received his Ph.D. degree in computer science from the Tongji University (China), in 1999. He is currently a professor with the Faculty of Computer Science, Shanghai Jiao Tong University, Shanghai, China. He is a member of CCF. His current research interests include, but are not limited to, computer architecture, compiling, virtualization and hardware/software co-design.

ErZhou Zhu is currently a Ph.D. candidate at Shanghai Jiao Tong University, China. He received the M.S. and B.S. degrees in computer science and technology in Anhui University, China, in 2004 and 2008 respectively. His research interests include virtual machine, binary translation and computer architecture.

Yongqiang Gao is currently a Ph.D. student of at the Shanghai Jiao Tong University (SJTU), China. He received his M.Sc. in Computer Applied Technology in 2009 from Northeastern University, China. His main research interests are computer architecture, virtualization and green computing

Alei Liang received his Ph.D. degree in computer science from Shanghai Jiao Tong University, China, in 2002. He is currently an assistant professor with the Faculty of Computer Science, Shanghai Jiao Tong University, Shanghai, China. His current research interests include, but are not limited to, parallel computing via swarm intelligence and virtualization computing with dynamic binary translation.

Received: March 27, 2010; Accepted: January 14, 2011.