Ma RH, Guan HB, Zhu EZ *et al.* Partitioning the conventional DBT system for multiprocessors. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 26(3): 474–490 May 2011. DOI 10.1007/s11390-011-1148-1

# Partitioning the Conventional DBT System for Multiprocessors

Ru-Hui Ma (马汝辉), Hai-Bing Guan (管海兵), *Member, CCF*, Er-Zhou Zhu (朱二周), Hong-Bo Yang (杨洪波) Yin-Dong Yang (杨吟冬), and A-Lei Liang (梁阿磊), *Member, CCF*

*Shanghai Key Laboratory of Scalable Computing and Systems, Department of Computer Science and Engineering Shanghai Jiao Tong University, Shanghai 200240, China*

E-mail: {ruhuima, hbguan, ezzhu, yanghongbo819, yasaka, liangalei}@sjtu.edu.cn

Received February 19, 2010; revised March 14, 2011.

**Abstract** Noticeable performance improvement via ever-increasing transistors is gradually trapped into a predicament since software cannot logically and efficiently utilize hardware resource, such as multi-core resource. This is an inevitable problem in dynamic binary translation (DBT) system as well. Though special purpose hardware as aide tool, through some interfaces, provided by DBT enables the system to achieve higher performance, the limitation of it is significant, that is, it is impossible to be used widely by another one. To overcome this drawback, we focus on building compatible software architecture to acquire higher performance without platform dependence. In this paper, we propose a novel multithreaded architecture for DBT system through partitioning distinct function module, which is to adequately utilize multiprocessors resource. This new architecture devides couples the common DBT system (DBTs) working routine into dynamic translation, optimization, and translated code execution phases, and then ramifies them into different threads to enable them concurrently executed. In this new architecture, several efficient novel methods are presented to cope with intractable work that puzzles most researchers, such as communication mechanism, cache layout, and mutual exclusion between threads. Experimental results using SPECint 2000 indicate that this new architecture for DBT system can achieve higher performance — speed up the traditional DBT system by about average 10.75%, with better CPU utilization.

**Keywords** DBT, optimization, microprocessor, multi-core, SPECint 2000

## 1 Introduction

For last two decades, more and more consumers depend fully or partially on computers due to the continual performance improvement that enables many complicated applications. The exponential performance improvement is mainly from multi-core microprocessor system[1-2], since the relentless pace of Moore's Law will lead to mainstream multi-core microprocessor designs with extensive on-die integration of a large number of cores[3]. Currently, lower performance of dynamic binary translators is deemed as the main development bottleneck, since out-of-date structure of dynamic binary translation system (DBT) with uniprocessor cannot provide enough power[4-7].

Multi-core microprocessor system offers an opportunity to improve the design and performance of dynamic binary translation system[4-9], which encompasses the idea of providing independent computing service for heterogeneous platforms. Currently, the conventional DBT system, such as QEMU[5], UQDBT[6], and StarDBT[7], remain its obsolete architecture that is successfully applied in large, monolithic uniprocessors system. But this out-of-date architecture is not competent for an eligible consumer to utilize full resource of multi-core microprocessor system. With chip multiprocessor, ADORE[10] is a research dynamic optimizer from the University of Minnesota. It uses a separate OS level thread to perform profiling and optimizations (e.g., prefetching) by taking the advantage of Intel Itanium specific hardware counters. But this architecture needs accurate prefetching. In contrast, Trident framework[11] focuses on adding lightweight hardware to perform all of the profiling needed to guide our dynamic optimizations. The hardware interacts with the optimization framework by generating events that helper threads consume to make optimization decisions as well as to perform the optimizations. In comparison to ADORE, this event-driven hardware profiling support avoids switching context to the additional thread for profiling, and allows continuous monitoring of more complex behavior. Although Trident overcomes

Regular Paper

This work was supported by the National Natural Science Foundation of China under Grant Nos. 60970108, 60970107, the Science and Technology Commission of Shanghai Municipality under Grant Nos. 09510701600, 10DZ1500200, 10511500102, IBM SUR Funding and IBM Research-China JP Funding.

the limitation caused by ADORE, extra hardware employed limits its extendibility. The important point is that the architectures mentioned above only consider their performance assisted by the outer factors, rather than their essential features. For example, the DBT system can achieve better performance through profiling, hot trace, and linking, etc., but adjusting their architecture under chip multiprocessor brings extra performance improvement as well. As we can see, the latter one has better universal property.

As widely noted in the DBT system research community, a novel software architecture of DBT system must extremely extract enough concurrency or parallelism to keep most, even all cores busy. However, parallel programming for DBT system is still a difficult research topic because the worthless byproduct of new DBT architecture puzzles most of researchers as well, such as how to select available-parallel candidate, avoid mutual exclusion, and promise programming logicality.

This paper exploits the design and implementation of new architecture for DBT system under the multi-core executive environment. In DBT system, traditional execution mode could be divided into two parts: translation stage and execution stage. However, either translation stage or execution stage has several important operations to enable application to execute efficiently, such as hot trace optimization, linking, and even profile/instrumentation operation[12]. How to select the parts as the candidates to be concurrently executed is considered as the key to construct the novel architecture. Then a quantitative analysis referring to the overhead of each part is presented, and we partition the traditional DBT system into several parts: the translation part, execution part, and optimization part, which can be concurrently executed in distinct private thread. Then the new architecture — multithreaded architecture relying on the analysis result above is built. The purpose of it is to reasonably and efficiently utilize microprocessors resource. However, like the inevitable problems occurring in most of parallel programming, the intractable software barriers in multithreaded DBT system (e.g., mutual exclusion between threads, communication mechanism, assignment scheduling, etc.) have been encountered. Therefore, several corresponding methods employed to address these relentless obstacles are proposed, which directly impact the system performance. In this paper, the method of private code caches is presented to mutual exclusion caused by code block stored in code cache competing for the limited memory resources. In addition, a lock-free communication mechanism — assembly language communication mechanism (ALCM) is proposed, which reduces much communication overhead between hot trace thread and profile thread, compared to the tradition communication mechanism. The important point is that branch tree module (BTM) is used to schedule different threads in multithread DBT system. There are two contributions of this paper that are different from other DBT system as well.

• In this paper, the main stages that directly impact DBT system performance have been rendered. This paper also analyzes their relationship, and shows the parallel-available candidates.

• To take advantage of full resource of multi-core system and achieve more performance improvement, a new multithreaded architecture for DBT system is presented. It is constructed through decomposing the main stages into separate, low-level threads, which are suitable for concurrent execution, that is, some stages (e.g., translation, execution, hot trace building, context switch, and profile, etc.) executed in sequential will be divided into several groups to be concurrently executed in corresponding threads, according to the result from elaborate quantitative analysis.

The following sections are organized as follows. Section 2 introduces the main stages that affect DBT system performance. In Section 3, we depict the architecture of the multithreaded DBT system, and several efficient methods to cope with arduous problems encountered. Moreover, Section 4 gives the evaluation of new architecture. And Section 5 reviews the related work. Finally, we conclude the paper in Section 6.

## 2 Overview of DBT System

Dynamic binary translation system has been widely used for many years. Its purpose is to transform source binary code into target binary code, either by emulating features of the source machine or by identifying such features and then transforming them into equivalent target machine features. Indeed, this execution flow — providing independent service for heterogeneous platforms is widely applied in various research domain, such as virtualization[13], legacy binary code[14], co-design infrastructure[15-17], and others[7,18-23].

Without loss of generality, DBT system always adopt two-phase execution process: translation stage and execution stage. During running, the system begins with looking up the target block in the code cache via one hash table that stores each cached block's target program counter (TPC) and source program counter (SPC). If successful, context switch will be trigged to execute the target block continuously. Therein, another monitor operation — profile is also acted in this block to achieve more information (e.g., execution counter, branch information, etc.) in order to recognize program hot spots[7,18,20] for optimizations. Through profiled hot spots, the system will construct and optimize efficient hot traces that have only one entrance and several exits. However, if the target block is not cached in the code cache, the translation stage will be awakened to retranslate the target block. The translated target block will be committed to the code cache for next use. Note that, some indispensable optimization methods have already adopted by almost all the DBT system, such as linking (or chaining). Linking is an optimization method performed in all the basic blocks and hot traces by modifying the machine codes after they have been executed for once[24].

## 2.1 Qualitative Analysis for DBT System

As the main execution flow of DBT system is introduced above, we will profoundly analyze some stages that affect entire system performance, to find out which parts can be concurrently executed.

Here the DBT system discussed in this paper embody the intermediate representation named intermediate instruction (II) layer, which is used to unify the representation of various sources and target instruction sets. It mainly functions on quickly adding guest/host platforms for the DBT system designers. Not taking initialization overhead into account, we found several main stages that influence overall performance of DBT system.

• *Look-Up Stage.* During running, the look-up operation is to find that whether the target block is stored in code cache. If successful, the entry address of target block is returned. On the contrary, if it comes to a miss, the translation stage is trigged.

• *Context Switch Stage.* After the entry address is achieved, dispensable context switch will occur to transfer control to the translated target block. When the execution of this block completes, another context switch is reactive to back across control.

• *Translation Stage.* When there exists a miss for target block, that is, the target one cannot be found in the code cache, the translation stage is awakened to translate or retranslate it. The translation overhead includes several parts: decode, intermediate code optimization, and encode.

• *Execution Stage.* It is the native execution of the translated code blocks. The better the quality of translation algorithm applies, the less time it would take. Additionally, optimizing intermediate representation of executed block either statically or dynamically will reach to performance improvement as well. The execution overhead is only from the execution process of translated code blocks.

• *Linking Stage.* In order to keep the original program behavior and relieve context-switch counter, linking is employed to chain the block that is being executed with the next one.

• *Profile Stage.* By way of accelerating the application execution, DBT system use runtime information to detect hot (frequently executed) code and optimize it. Currently, either special hardware support or software support for profile has been widely used, but in this paper, to improve general-purpose DBT system performance, we only consider software profile method. The profile method is to add several codes into each code block to get its running information. Due to several codes inserted into each code block, the execution time for each block is extended, so the profile overhead is decided by the added codes and the blocks' execution time.

• *Hot Trace Building Stage.* It is essential to build hot trace for DBT system to achieve significant performance improvement, according to hot spots monitored.

In fact, the overall running time of DBT system is composed of two parts: all the execution time of each target block, and the sum of each target block's running time. The overall running time of DBT system is depicted as:

$$T_{\text{total}} = T_{\text{execution}} + T_{\text{running}}, \tag{1}$$

where $T_{\text{execution}} = \sum_{i=1}^{n} Te_i$, $T_{\text{running}} = \sum_{i=1}^{n} Tr_i$. In traditional DBT system, the execution time of target block $Te_i$ is only affected by the quality of translated code, that is to say, it is a fixed value since translation mode is invariable. However, the running time of target block $Tr_i$ is variable, involving with several elements.

There are three main factors that drastically impact system running overhead. The first one is unlinking that causes much unnecessary overhead. For example, if code block unlinked by others is to be executed before long, transferring control to system will occur, that is, lookup, context switch will be reactive orderly. We can see that this leads to two parts overhead: $T_{\text{lookup}}$ and $T_{\text{context-switch}}$, which are expressed as follows,

$$T_{\text{unlink}} = T_{\text{lookup}} + T_{\text{context-switch}}. \tag{2}$$

Another one is miss penalty. When cache miss taking place, new translation stage will be reactive to translate target block. The new one translated is committed to code cache with another linking. In this process, interpret operation happens before translation, while cache replacement cannot be fully abandoned. (3) exposes cache miss penalty in detail.

$$T_{\text{miss}} = T_{\text{interpret}} + T_{\text{translate}} + T_{\text{replace}} + T_{\text{link}}. \tag{3}$$

The last one is hot trace building optimization. In fact, hot trace building depends on enough block executing information collected by profile operation.

Although accurate profile information is able to give us chance to implement optimization for some blocks, the expensive overhead caused by profile operation cannot be negligible. This overhead is described in (4).

$$T_{\text{optimization}} = T_{\text{profile}} + T_{\text{hot-trace}}. \tag{4}$$

Finally, the running time of DBT system based on analysis above can be expressed as follows:

$$\begin{aligned} T_{\text{total}} = & T_{\text{execution}} + T_{\text{lookup}} + T_{\text{context-switch}} + \\ & T_{\text{interpret}} + T_{\text{translate}} + T_{\text{replacement}} + T_{\text{linking}} + \\ & T_{\text{profile}} + T_{\text{hot-trace}}. \end{aligned} \tag{5}$$

Similar to evaluating CPU via average accessing time from physical cache, the performance of DBT system is able to be evaluated by average executing time of target block as well. The running time of DBT system — $T_{\text{total}}$ could be considered as:

$$T_{\text{total}} = N_{\text{block}} \times T_{\text{block}} + T_{\text{hot-trace}}. \tag{6}$$

In (6), almost all the main stages that impact system performance can be represented by per-block time consumption except the stage — hot trace building, since not all translated basic block will be used to comprise different hot traces. $N_{\text{block}}$ is the number of blocks treated. Its value equals the time of look-up operations. It is determined by the code inflation rate, the cache size, and the replacement strategy/algorithm of code cache. The smaller the inflation rate, the bigger the cache size would certainly benefit the DBT system at runtime. On the other hand, $T_{\text{block}}$ is the per-block time consumption, and it is shown in (7):

$$\begin{aligned} T_{\text{block}} = & T_{\text{ave-execution}} + T_{\text{ave-profile}}. \\ & R_{\text{unlink}} \times (T_{\text{lookup}} + T_{\text{context-switch}}) + \\ & R_{\text{miss}} \times (T_{\text{interpret}} + T_{\text{translate}} + T_{\text{replace}} + T_{\text{link}}). \end{aligned} \tag{7}$$

The $R_{\text{unlink}}$ parameter stands for the percentage of the translated blocks unlinked. It is complementary to the linking rate — $R_{\text{link}}$ which is composed of two categories sorted by the blocks' exit type. Commonly, they are of the direct control-transfer type, and the indirect control-transfer type. The former may be a constant at runtime. But the linking rate is contrary. It is affected by the possibility of the indirect control-transfer linking failures. A relatively stable linking rate can be reached, but it is brought down when the code cache system evicts some translated blocks.

The $R_{\text{miss}}$ parameter represents the miss rate of the code cache system. It is commonly determined by the cache size, the binary inflation rate, and the replacement strategy. The foremost factor is the cache size. Commonly, if the size of the text section of the guest platform (guest binaries' size) multiplying the binaries' inflation rate of the DBT system does not exceed the size of code cache, or not much larger than it, the miss rate is low and consequently leads to the good system performance. Otherwise, replacement happens frequently.

The per-block profile overhead — $T_{\text{ave-profile}}$ is that profiling basic blocks require code instrumentation and executing instrumentation code. Since instrumentation code together with each basic block is indistinctive, regardless of which profile method employed (e.g., hot spot profile, edge profile, and trace profile, etc.), it brings about the same overhead for each block.

The per-block execution time — $T_{\text{ave-execution}}$ means the per-block execution time of the native binaries wrapped in the translated blocks. The per-block hashmap look-up time — $T_{\text{lookup}}$ together with the per-block context switching time — $T_{\text{context-switch}}$, comes as the penalty for the blocks-unlinked condition. We call it the unlinked penalty — $P_{\text{unlink}}$:

$$P_{\text{unlink}} = T_{\text{lookup}} + T_{\text{context-switch}}. \tag{8}$$

However, without the replacement of code cache, translated blocks may be well linked. But another intractable issue encountered is that as the size of new software releases grows, unbounded code cache sizes will grow proportionately or exponentially. The ideal method of addressing this problem is to provide adequate cache size without any replacement strategy, but the memory resource is significantly limited so that this is unavailable. Currently, the best avenue is to assign certain cache size with efficient replacement algorithm.

When the code cache system misses, there are several kinds of miss penalty — $P_{\text{miss}}$: per-block interpret time ($T_{\text{interpret}}$), per-block translating time ($T_{\text{translate}}$), per-block replacement algorithm time ($T_{\text{replacement}}$) and per-block linking time ($T_{\text{link}}$).

$$P_{\text{miss}} = T_{\text{interpret}} + T_{\text{translate}} + T_{\text{replace}} + T_{\text{link}}. \tag{9}$$

The replacement not only brings about miss penalty, but also brings up the unlinked rate as mentioned above. What is more, the number of blocks treated also rises. Under such a condition, the whole system processing time is determined by the $R_{\text{miss}}$. The per-block consumption is:

$$\begin{aligned} T_{\text{block}} = & T_{\text{ave-execution}} + T_{\text{ave-profile}} + \\ & R_{\text{unlink}} \times P_{\text{unlink}} + R_{\text{miss}} \times P_{\text{miss}}. \end{aligned} \tag{10}$$

And the system running time is:

$$T_{\text{total}} = N_{\text{block}} \times (T_{\text{ave-execution}} + T_{\text{ave-profile}} +$$

$$R_{\mathrm{unlink}} \times P_{\mathrm{unlink}} + R_{\mathrm{miss}} \times P_{\mathrm{miss}}) + T_{\text{hot-trace}} \tag{11}$$

or

$$T_{\mathrm{total}} = N_{\mathrm{block}} \times (R_{\mathrm{unlink}} \times P_{\mathrm{unlink}} + R_{\mathrm{miss}} \times P_{\mathrm{miss}}) + T_{\mathrm{execution}} + T_{\mathrm{profile}} + T_{\text{hot-trace}}. \tag{12}$$

(11) shows all the factors. Generally, if the $R_{\mathrm{miss}}$ is low enough, the whole system processing time is still dominated by the $T_{\text{ave-execution}}$. However, when it grows up, the $R_{\mathrm{unlink}}$ grows too, and the system execution time will not be predominated by a single factor. Any stage may harm or benefit the $T_{\mathrm{total}}$. At last, when the $R_{\mathrm{miss}}$ reaches up to a certain level, miss penalty will be the dominator. In addition, $T_{\mathrm{profile}}$ is dominated by the execution counter of basic block and also deemed as a considerate overhead. But it brings about another attracting advantage for system to indicate optimization — hot trace building, without compromising weighty overhead. Although the optimization on the $T_{\mathrm{execution}}$ is welcome in most of the cases, such as hot trace building, other optimizations on any of the stages are not welcome because they may be the candidates of the system bottleneck as the execution environment changes. So, in this paper, DBT system only employ hot trace building method to enable more performance.

(12) indicates several main factors affecting system performance. To better analyze the experimental result, we present $\Delta\mathit{Translate}$ to represent $N_{\mathrm{block}} \times (R_{\mathrm{unlink}} \times P_{\mathrm{unlink}} + R_{\mathrm{miss}} \times P_{\mathrm{miss}})$, so (12) can also be concluded as:

$$T_{\mathrm{total}} = \Delta\mathit{Translate} + T_{\mathrm{execution}} + T_{\mathrm{profile}} + T_{\text{hot-trace}}. \tag{13}$$

## 2.2 Quantitative Analysis for DBT System

According to (12) and (13), we only can be conscious of several parts affecting DBT system performance, but we still do not know which one is the most important factor. So we want to know their relationship that can judge whether they can be concurrently executed via reasonable yet convincing experiments. In this paper, CrossBit[25], as a resourceable and retargetable DBT system with intermediate instruction, is selected as the experiment platform, which is developed mainly to provide the platform independent computing service for a new virtualized execution environment. Until recently, it has fully or partially supported guest platforms including simplescalar, IA32, MIPS, SPARC, and has fully supported the IA32, SPARC host platform. On the other hand, the experiments are taken on the host platform (CPU: Intel 4-Core, 2.66 GHz; memory: 8 GB) with Linux Kernel Version 2.6.22. The benchmark programs are some selections from the SPECint 2000[26].

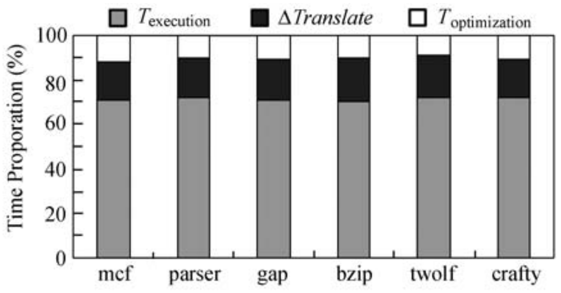


Fig.1. Several stages execution ratio in the conventional DBT system — CrossBit.

From Fig.1, we can see that DBT system — CrossBit have a relative-stable overhead ratio. For example, the main running time cost is execution stage, and its consuming proportion about 70% ∼ 72%, while translation stage changes about 17% from 20%. Additionally, optimization stage — profile and hot trace building takes the proportion about 8% ∼ 13%. Since each benchmark has its own architecture, the experimental result is so distinct but with rough rule. Although execution stage is the first main factor, other overheads cannot be ignored. If we can reduce the overheads except execution stage, the overall performance may be enhanced extremely.

First, in the overhead of $\Delta\mathit{Translate}$ mentioned in (13), context switch overhead is caused by control transferring, which can be fully avoided if anyone or anything is not allowed to influence target block to be normally executed, that is, another thread utilized to look up target block and return its entry address is unavailable to interwine execution thread.

On the other hand, hot trace building from optimization phase is executed in the translation mode as well. Indeed, when it is running, translation mode translating basic block is instantly ceased to retranslate blocks used to construct hot trace. This leads to another overhead that is also obviated via extra thread. Therein, it is the fact that several translation threads executed concurrently also bring another chance: two or more basic blocks can be translated synchronously. It may reduce some overhead with correct program logic. There is another reason that drives us decoupling translation mode and execution mode. Unfortunately, there is one thing inevitable for DBT system running upon one single-core uniprocessor: the translation mode and execution mode have to share the native registers which cause the register context-switch overhead. Under certain condition, the context-switch overhead could still be the foremost factor influencing DBT system. One suggestion is that the translation mode and execution mode share the target machine registers under the control of some static or dynamic monitoring mechanisms. However, for the DBT system developers, they handle only the registers' usage of translated code, which is determined in the translation phase (mostly done by the register allocation algorithms). But the register context of execution mode is determined by the compiler ahead of time. It is almost impracticable or inefficient to make them co-workers because the code under execution mode is always built prior to that of translation mode, and is changed by versions if the developer does not make any modification to the local compilers. As well, the later Subsection 3.5 will depict context-switch overhead elimination.

Finally, DBT system still sustain the inevitable profile overhead. Virtually, profile overhead embodies two kinds of overhead: inserting profile instruction overhead as well as executing profile instruction overhead. The former one results in further less overhead than the latter one, for it only inserts some codes into each basic block. However, the size of basic block fluctuates about 100 $\sim$ 200 bytes, while that of profile code is about 50 bytes or more. When each translated basic block is executed, profile codes embedded will also be executed. Therefore, the latter will drive much more overhead due to its large size. Suppose that inserting profile code is only empowered when basic block translating, while the other operation — executing this profile code is permitted in different thread associated with another core. This cannot impact applications to be executed properly with much overhead, but another performance improvement will come due to well taking advantage of multi-core resource. In a word, multi-core architecture offers an opportunity to improve the design and performance of DBT system.

## 3 Multithreaded DBT System

As the processor technology is evolving into the multi-core age, there is another possibility to decouple several parts. The simple concept is to make the parts be separated by threads. And the threads may be then mapped into different processor cores on the fly. Through qualitative analysis and quantitative analysis for DBT system above, we conclude that several parts can be concurrently executed, such as translation thread, execution thread, and profile thread. Meanwhile, translation thread includes two kinds of threads: one hot trace building thread, one or more basic block translation threads.

In fact, the design of new execution engine mainly aims at reducing the $T_{\rm running}$ overhead. One obvious advantage is that either the translation mode or the execution mode has its own register context. Consequently, the elimination of the context switching overhead achieves. Additionally, tangled profile overhead is extremely depressed in different threads, via efficient communication mechanism. Except for avoiding context switch and decreasing profile overhead, another goal of the new architecture is to reduce the cost of translation time. This profit not only comes from independently constructing hot trace, but also simultaneously translates basic block. As mentioned in the previous section, the long-time translation phase may greatly prolong the total running time of DBT system

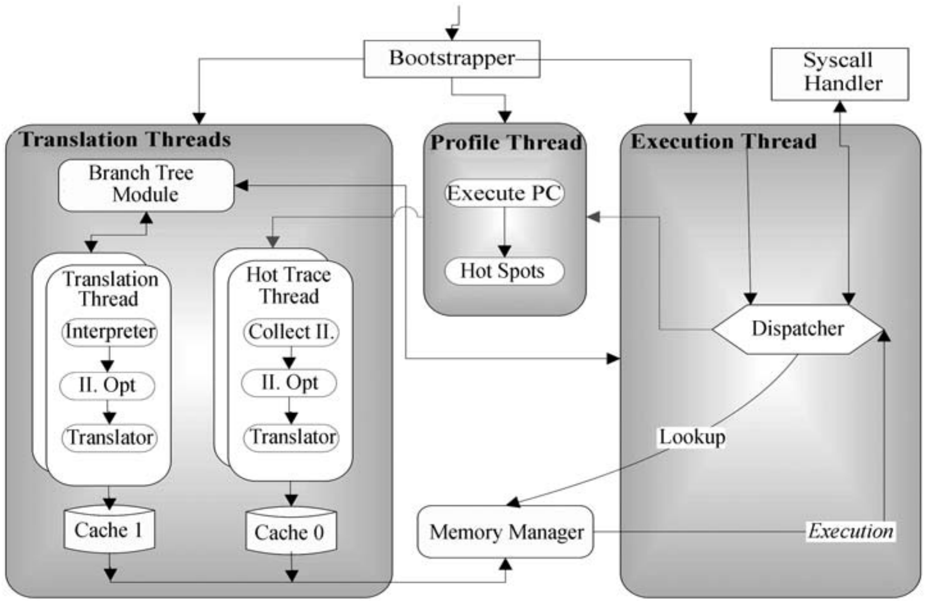


Fig.2. Architecture of multithreaded DBT system.

if the frequent replacements from code cache occur. This is not occasional since there are many memory-limited embedded processors planning to have multi-cores. The simple implementation is that the execution mode issues several translations at one time. If the prediction of translation is always correct, the translation mode can get more code to execute and have less time waiting for the long time translation.

## 3.1 Architecture of Multithreaded DBT System

Fig.2 is the architecture of multithreaded DBT system. According to the decoupling principle as mentioned above, the traditional DBT system can be divided into three parts: translation threads, profile thread, and execution thread. This new architecture is not only simply divided, but also adopts several novel parts to address arduous problems encountered. BTM, as the controller of dispatching tasks to translation threads, is added into the new one. On the other hand, a lock-free communication mechanism is responsible for transferring information between profile thread and hot trace thread. In most traditional DBT system, unitary code cache is always utilized to cache translated basic block or hot trace[18], while some DBT systems select two-level or heterogeneous code cache to improve system performance. However, in this new architecture, unitary code cache is out of state, that is, it cannot accommodate multithreaded executive environment[27]. So private code caches employed are advantageous to obviate unpredict data mutual exclusion without lock/unlock mechanism.

The engine starts from bootstrapper. The translation Threads perform the overall translation for basic block, including translating source binaries, wrapping translated code into blocks and committing them to the code cache. The other task for translation threads is to collect hot spots to build hot trace, translate them, and then insert the hot trace into the corresponding private cache. Note that hot trace thread is not controlled by BTM, and it is only affected by profile thread. The BTM is the controller of threads' work dispatching. The translation thread for basic block asks the BTM for a source PC address from which the entry address translation starts. At the end of each translation, the translation thread commits a translated code block to the code cache. As mentioned above, all the commitments are out of order. And then the translation thread would query the BTM for next translation. Definitely there is a mechanism inside the branch to make the work of translation thread valuable. This would be introduced later.

Meanwhile, profile thread gets each block's execution information from execution thread, and this process only needs a lower-cost queue. Then the information about hot spot is passed to hot trace thread through a new communication mechanism — ALCM. In addition, the execution thread is initialized and gets to work. However, it is a much easier work to look up the translated code blocks from code cache and perform the target block circularly.

## 3.2 Branch Tree Module

BTM is the key data structure to enable the parallel translation and the translation/execution/profile decomposition. In this subsection, we will give the architecture of BTM that possesses three modules: BranchTree, BranchCodeGrabber, and ThreadDispatcher, which are depicted in Fig.3 as follows.

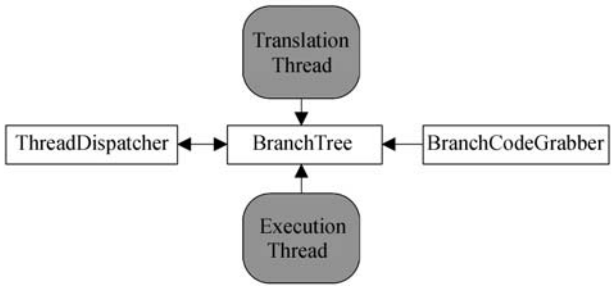


Fig.3. Architecture of branch tree module.

### 3.2.1 *BranchCodeGrabber*

The module of BranchCodeGrabber assists BranchTree to achieve the branch instruction's target address of relocated binary code, and it provides BranchTree with the following interface *BItem GetNextBranch*(*MemAddr blockentry*), where *blockentry* represents the basic block's entry address. The return value of this function is an architecture body that embodies all the indirect-branch target addresses of the basic block and their information about instruction type.

### 3.2.2 *ThreadDispatcher*

ThreadDispatcher is also the important member of multithreaded DBT system, responding to constantly assigning work to the translation thread, and it is outer interface defined as *MemAddr ScheduleThread*(*MemAddr curAddr*). Through looking up BranchTree, the engine will receive the information of block to be translated, and its entry address as the return value can be achieved as well. Note that ThreadDispatcher controls the assignment algorithm that is the key point to affect system performance, which is depicted as follows.

Suppose that there are several translation threads. If translation thread *A* has completed translating *Block X*, it will pass QueryNextEntry to BranchTree to require the next block.

• Then *Block X* is tagged with "done".

• If the left (right) child node is tagged with "undone", this sign will be altered and shown as "doing", and the entry address of this block being translated is returned.

• If there does not exist the node tagged with "undone", nulla address will be returned. When translation thread receives nulla address, it will go to sleep until the architecture of BranchTree is modified.

We can see that when there is a request from translation thread, it selects the most valuable and available node from the BranchTree for next translation. The selection algorithm determines the accuracy of prediction (branch prediction of instructions accordingly). In the current version, the new architecture presented implements a simple selection algorithm based on tree structure as mentioned above.

#### 3.2.3 BranchTree

BranchTree, as the kernel of BTM, has the responsibility to provide sufficient interfaces for other modules, such as translation thread interface and execution thread interface. The translation thread employs the former interface to gain the next translated block's entry address, which is defined as *MemAddr QueryNextEntry*(*MemAddr curentry*). Indeed, translation thread is only to do query and translation operations. The latter one is used to not only receive the SPC (source program counter) to fulfill execution thread, but also adjust BranchTree's architecture. It is described as *Status BranchTakenAt*(*MemAddr spc*). The following steps introduce modifying algorithm for BranchTree.

• Looking up the node corresponding with SPC in BranchTree.

• If the node is null, this is caused by register addressing branch instruction. Meanwhile, the engine will delete the original BranchTree and rebuild BranchTree, the root of which is SPC.

• If the node is BranchTree's root, there is no alteration for BranchTree.

• If the node is not the root node in the BranchTree, the engine will protect it and its subtree, delete other nodes, and complement it. Specially, the complement principle is that the depth of rebuilt BranchTree is the max depth formulated by system, and the root of it is the node protected.

Notice that either the initialization of BranchTree or dynamically adjusting its architecture absolutely depends on the binary code's address information analyzed by BranchCodeGrabber. And dispatching the work for the threads is done in accordance with the algorithm provided by ThreadDispatcher and the current architecture of BranchTree.

In multithreaded DBT system, the BranchTree is the description of control flow ignoring any loop conditions, and is organized as a binary tree. The BranchTree has the following features:

• every tree node represents a basic block, and tagged with the block entry address;

• the root of BranchTree represents the block last being executed;

• the subnodes are the exits (destine address of branching) of this block.

Whenever the execution thread finishes a block execution, it will update the global-shared Source PC variable. Then whenever the translation thread queries for new task, it may find out the dismatch between the BranchTree root and the global shared variable. Then the translation thread may update the BranchTree data structure according to the change. This mechanism ensures that the translation thread never delays in response to the real requirement from the execution thread. What is more, it limits the synchronization overhead.

### 3.3 Assembly Language Communication Mechanism

From Fig.2, we can see that the communication happens when the profiling module is going to pass the information of hot spots to the hot trace building thread. It is well known that an adaptable and effective communication mechanism is a key point to pass the important information of hot spots between threads. However, traditional threads' communication mechanisms, such as monitor or producer/consumer, do not fit DBT system effectively due to their drawbacks.

*Monitor.* A monitor supports synchronization through condition variables as a part of monitor[28]. Indeed, this model forms the backbone of almost all the event-driven multithreaded software system in the world. The advantage is that all the threads waiting on the condition variables only require signals to run, instead of competing for the processor time slice all the time. Unfortunately, the fatal disadvantage of this mechanism does not make it adapt to our framework. As when there is no thread waiting on condition $x$, then the execution of *csignal*($x$) has no effect at all. In other words, the hot trace building thread will lose the signals sent by the profile thread if it is still building the hot trace, rather than waiting on the condition variables.

*Producer/Consumer.* This model[28] makes all threads operate their critical section, which employs extra semaphore operations such as *SemSend*() or *SemWait*(). The critical section refers to the portion of the program that uses a single non-sharable global resource which is accessed by two or more threads. All these operations are required to be done at the same language level. The following is the procedure of the detailed conventional communication mechanism in producer/consumer.

1) The system firstly initializes the writing semaphore $S = 1$, which controls the producer (main thread) to write into critical section, and the reading semaphore $C = 0$, which controls the consumer (helper thread) to read from critical section.

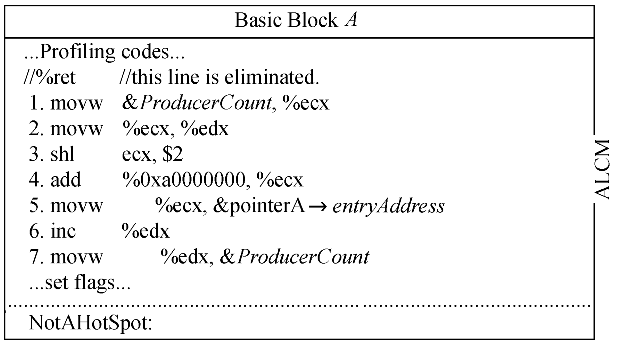
2) When producer brings new data to be inserted into the critical section, the writing semaphore $S$ will be decreased by 1, and then the main thread obtains the control of critical section. Meanwhile, the critical section is represented as the program code that accesses the critical resource (critical resource is the sharable resource that is utilized by only one thread).

3) When the write thread accomplishes the writing operation with global variable, the value of the two communication semaphores will be added by 1. Then the read thread gains the control to decrease the value of reading semaphore by 1, and successively, accesses to the critical section.

We can see that producer/consumer needs a lock/unlock algorithm to prevent mutual exclusion in the critical section, which is associated with extra overhead. Otherwise, in multithreaded architecture, the profiling module in Fig.2 is realized by the assembly instructions in each basic block while the hot trace building thread is implemented by C++, so we must introduce some extra overheads, if we still employ this model. This disadvantage comes from the redundant context switching operation, which is used to exit the back-end execution process to perform semaphore operations like *SemWait*(). The reason why context switch occurs frequently in multithreaded architecture is that once the profile module finds a hot spot, the context switch is trigged to back to high level language to complete passing hot spot information. Additionally, the critical section of this model, a section of memory, allows only a sort of operation to access. Since in this critical section, only two sorts of operations — write/read — exist, if the write operation (producer) is being executed, the read operation (consumer) must wait to be executed until another execution is completed (or vice versa). Note that the waiting time cannot be fully ignored.

In multithreaded DBT system, profile thread reads the edge information in a simple queue, which links two translated blocks. Through filtering, hot spot should be passed into hot trace thread. To make our architecture more effective and sound, a new communication mechanism that acts between profile thread and hot trace thread is proposed.

As shown in Fig.4, the new multithreaded architecture uses a new threads' arguments lock-free communication mechanism called ALCM which is implemented at assembly language level. Compared with producer/consumer communication method, ALCM does not have to suspend the target-machine codes' executing process of the main thread to make any semaphore operations. And several operations can be simultaneously executed in critical section. This mechanism based on producer/consumer is also composed of three parts: producer, consumer, critical section, but it has its bright feature.



```
Basic Block A
...Profiling codes...
//%ret      //this line is eliminated.
 1. movw    &ProducerCount, %ecx
 2. movw    %ecx, %edx
 3. shl     ecx, $2
 4. add     %0xa0000000, %ecx
 5. movw        %ecx, &pointerA → entryAddress
 6. inc     %edx
 7. movw        %edx, &ProducerCount
 ...set flags...
NotAHotSpot:
```

Fig.4. Detailed implementation of the producer of ALCM.

Fig.4 describes the producer of ALCM that is inserted into the translated basic block in the wake of profiling codes. It is trigged only by a hot spot discovered, or else it is skipped. The processing procedure of it is as follows in detail: if the hot spot just discovered has not been stored in the memory space — $M$, the value of *ProducerCount* will be moved into the arithmetic register — %ecx and the backup register — %edx. In addition, the variable *ProducerCount* is used as the writing semaphore. Then the arithmetic register is moved left by 2 bits, that is, it is equal to multiplying 4, for the hot spot to be cached in the memory space needs 4 bytes space (the entry address of hot spot is an unsigned integer variable). We select shift operation, instead of multiplying operation, due to the former outperforming the latter. We will get the real memory address of the hot spot that is equal to the value of modified arithmetic register adding with the first address of $M$. After the entry address of hot spot is stored via movw operation, the original value of *ProducerCount* from %edx is added by 1.

In hot trace thread, it firstly reads the value of *ProducerCount*, which is to be compared with the value of *ConsumerCount* (reading semaphore). If the comparison result is unequal, this indicates that the *ProducerCount* is modified by the profile thread because the initial values of them are 0. Due to hot trace thread built by high-level language — C++, the first address of sucessive memory space is easily achieved by addressing mode used in high-level language. When the work of building hot trace is finished, the hash map is also updated. On the other hand, when the comparison result is equal, the hot trace thread will continuously wait until the *ProducerCount* is altered.

The exclusive architecture of critical section is depicted as follows.

1) Resembling producer/consumer, the critical section of ALCM is similarly a segment successive memory space, but it is a software queue. The advantage of it is that more than one hot spots can be cached in the critical section while only one hot spot is cached in producer/consumer's critical section. In addition, the frequency of replacement in producer/consumer's critical section should be considered.

2) The producer and consumer of ALCM that dispense with the lock/unlock algorithm (it is lock-free), can be simultaneously executed in the critical section. That is, producer is writing hot spots' information, while consumer can read it at the same time. In ALCM, the so-called critical section is not the real critical section in the strict sense, for more than one threads can be executed in it at the same time. We can see that this mode is able to avoid the extra overhead caused by asynchronous waiting.

Compared with producer/consumer communication method, ALCM does not have to suspend the target-machine codes' executing process of the profile thread to make any semaphore operations. Fig.5 shows the performance comparison of ALCM and producer/consumer communication mechanism applied in the new architecture (the experiment setup information is: CPU, Intel 4-Core, 2.66 GHz; memory, 8 GB, with Linux kernel version 2.6.22). The result is that the decreased execution time of multithreaded DBT system with ALCM unexpectedly reaches 10.7 seconds on average, and this indicates that ALCM outperforms the Producer/Consumer method when it comes to the multithreaded architecture.

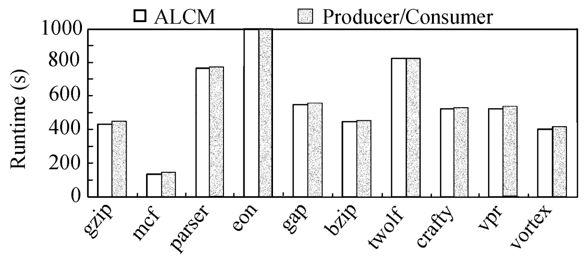


Fig.5. Performance comparison of ALCM and producer/consumer strategy.

## 3.4 Private Code Caches

Code cache[29-30] utilized to store translated blocks has been widely employed in traditional DBT system. Compared to physical cache, it is much easier to be controlled, since it is only a software cache — a section of successive memory space. Moreover, since a translated block inserted into code cache is to be executed or replaced, the writing policy in code cache need not be considered, which must be seriously considered in physical cache. In software code cache, stored blocks are independent of other sections of memory space. Multithreaded DBT system also select code cache to amortize miss penalty, but it does not take advantage of monolithic code cache.

If multithreaded DBT system still choose monolithic code cache, the relentless mutual exclusion will drastically strike the new architecture. For example, there are two or more writing operations (inserting translated blocks into unitary code cache) simultaneously acting on the critical section — unitary code cache to competing for the same address, mutual exclusion must be reactive. To avoid it, lock/unlock algorithm is competent for this work but with huge overhead caused by asynchronous waiting time. Therefore, to solve this serious problem with lower overhead, we present a lock-free and no waiting-time method — private code caches, that is, rationally dividing monolithic code cache into several parts. In detail, one part is used to store hot trace, while the other parts are assigned to store translated basic block. Note that it is not merely the result that comes from dividing the unique code cache into several special caches, and each cache is thread-private cache, but the objects stored in the distinct caches can communicate with each other through linking. Along with writing basic blocks into the respective translation caches, the operations about reading/writing with translation caches are controlled by hash function. The detailed mapping relationship of hash function is: the last four bits of hexadecimal entry address are the key to acquire the corresponding mapping result. For instance, assuming that the entry address is denoted as 0x4005678, simultaneously, and the offset of it is 5678; the data, 5678, as the exclusive memory addressing data, indirectly indicates that the simplest addressing mode further consolidates the memory management without extra cost. In addition, the explicit partitioning proportion for code cache is also in accordance with system architecture. This will be introduced later.

## 3.5 Context-Switch Elimination

In multithreaded DBT system, the context-switch overhead is not necessary for the system any more. This overhead mentioned in Subsection 2.2 still puzzles many DBT systems under uniprocessor physical machine. As Fig.6 shows, the execution thread involves some shared contexts, including the shared variables such as BranchTree represented as *btree*, code caches represented as *cache*, and SysCall Handler represented as *syscall*, etc. When the optimization options are selected by the native compilers, shared variables may also be compiled and optimized into registers. This may lead to unpredictable runtime behaviors of execution thread.

```
while (1) {
    //Lookup the translation cache 0/1 for the target code
      block,
    //if success, execute it.
      curtblock = cache → lookup(*lookupAddr);
      if (curtblock == NULL){
        btree → BranchTakenAt(*lookupAddr);
        continue;
               }

    //TBlock retrieved, execute it
      ((void(*)()) curtblock → enterAddr()) ();

    if (*exitReason == TBlock:ExitReason::SYSCALLEXIT)
    {
      //Exit syscall
        if (syscal → syscode() == 0x1) {
            exit(0);
              }
    (*syscall) ();
            }
      }
```

Fig.6. Main flow of execution thread.

However, with clear definition of threads' categories, the shared variables inside the execution thread are under control. In fact, all the shared variables may be protected by the compiler-supported keyword (like "volatile" in C++) which makes sure that they would not be optimized into registers, that is, they cannot be affected when target code block is running. And the protected variables does not occupy the registers required by the execution thread when target block is running. Compared to the register context in conventional DBT engine, the shared variables are less in quantity. The programmer can protect them all easily. Moreover, the DBT system version changes do not affect the correctness either.

## 4 Evaluation

Recently, many traditional DBT systems have been popularly applied in various research domains. Although some differences still exist, most parts are basically uniform. So this paper selects conventional DBT system — CrossBit as the experimental platform for the new architecture. In addition, CrossBit is running on the physical machine (Intel(R) Xeon(R) CPU (4-core) 2.66 GHz, 8 GB memory) with Linux Kernel Version 2.6.33.4. And the test benchmark is selected from SPECint 2000[26].

Since the code cache is one main factor that impacts on system overall performance in DBT system, we should firstly ascertain its cache layout. In this paper, the size of total code cache is defined as 1 MB. Note that, the lower miss rate — $R_{\text{miss}}$ in (12) system produces the performance we achieve. Kim[31] has been proved that with lower overhead and cache miss rate, replacement strategy FIFO (first-in-first-out) outperforms others. So in this paper, FIFO algorithm is employed for each cache. In DBT system, hot trace is defined as the compositive block constructed by frequently executed blocks. Though the number of it is further less than that of basic block, almost all the performance is from the execution of hot trace. For instance, Table 1 depicts the contribution for system performance when hot traces and basic blocks from benchmark mcf and crafty are respectively executed at runtime.

In Table 1, we can see that the system performance is mainly from frequently executing hot trace (*hot trace's execution time*: *that of basic block* $\approx$ 1000 : 1), yet without considering the proportion of hot trace either in mcf or in crafty. As we know, a program spends 90% (or more) of its execution time in 10% (or less) of its code. Likewise, the execution of hot trace insists on keeping to this principle. It is concluded that if hot trace can be executed at once when called, that is, it is stored in hot trace cache, higher performance will be normally achieved.

In fact, the method of addressing the problem mentioned above is to prolong residence time of hot trace in its private cache, which can reduce cache miss rate. There are two main factors that can fully influence hot

**Table 1.** Comparison of Hot Trace and Basic Block in SPECint 2000

| Benchmark | Size (KB) | Total Blocks (block) | Hot Traces (block) | Execution Time (hot trace vs. basic block) |
|---|---|---|---|---|
| mcf | 286.0 | 1 703 | 901 | 911 212 689 vs. 895 790 |
| crafty | 1 228.8 | 7 083 | 6 332 | 641 538 402 vs. 767 935 |

trace's residence time: adequate cache size; reasonable replacement strategy. Note that cache size is the only one thing that should be considered, while FIFO is selected as the replacement algorithm candidate. In multithreaded DBT system, monolithic code cache is divided into several code caches to satisfy the requirement of each thread in translation mode. So, how to divide the monolithic code cache becomes the focus, that is to say, the dividing proportion for each code cache should be precisely decided.

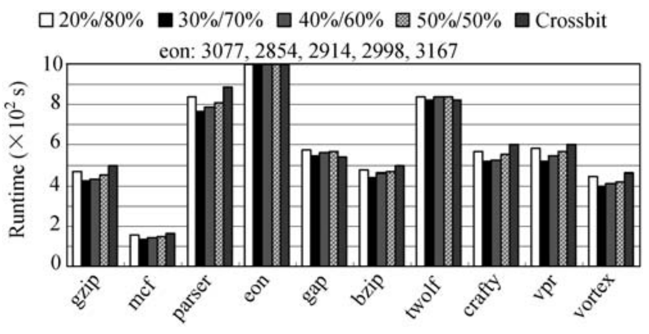


Fig.7. Performance of multithreaded DBT system with generational code caches.

Fig.7 compares the performance of three instances of generational code cache configurations in multithreaded DBT system with 4 threads to that of traditional DBT system — CrossBit and native execution. While the best cache configuration varied by benchmark — a generational code cache with a 30~70% size ratio between basic block cache and hot trace cache, respectively, performs best overall. The performance improvement compared with traditional DBT system — CrossBit, is about 10.75%. From Table 1, the execution time of hot trace is far longer than that of basic block, while the former gets more space size than the latter. As compared to others in multithreaded DBT system, the small cache size of hot trace will lead to inadequate hot traces in its private cache to cause more retranslation overhead, while the large one will result in less basic blocks in their caches when total cache size is invariable. Indeed, the hot trace miss penalty is even greater than that of basic block, since each hot trace is composed of 2 or more basic blocks (average number is 4 block blocks, yet the max number is about 23). The overhead of context switch is fully obviated, while intractable profile overhead is concurrently executed, and these bring performance enhancement as well. Meanwhile, the performance of benchmark crafty is enhanced to 14.59% in degree, due to its special architecture (the proportion of hot trace is 89.40%). On the other hand, less hot trace in the benchmark mcf and its small size can be considered as the main handicap to achieve more performance, and more basic blocks stored in smaller code caches will cause more miss rate.

Obviously, the benchmark gap brings down entire system's performance due to its special program architecture as well. The benchmark gap has multiple parts: one part of it is the standard gap speed-benchmark, exercising mostly the combinatorial functions and big number library, then some test functions for the finite field, permutation group and subgroup lattice computations, a program comparing two different methods of finding normalizers in solvable groups and finally a test exercising the collector for so-called ag-groups, and this is a piece where the bulk of the computation is not done by the interpreter. The benchmark gap is designed for mostly computing in groups, that is, it used to test most large or complex codes. Indeed, in multithreaded DBT system, if the executing process has many large basic blocks or special basic blocks (less loops) to run without encountering the ret instructions which are used to suspend the executing of target-machine codes, the hot trace linking module cannot access its critical section immediately after the hot trace is built. This may cause the hot traces linked into other blocks later than function-call. This fact has weakened the improvement brought by hot trace building as well as multi-threaded, such as gap in Fig.7. We can see that compared to conventional DBT system — CrossBit, multithreaded DBT system have a good performance through adding several threads. The essence of this new architecture is to utilize multi-core resource reasonably.

If cache size is not considered, that is, unlimited cache size is provided to accommodate all the blocks, the system performance will be enhanced significantly. Clearly, the novel one presented gains more performance from multi-core resource than traditional DBT system, and even this benefit is larger than the multithreaded DBT system with limited code cache, since there does not exist cache miss. However, this unbounded code cache cannot be widely applied[32].

Another research focus is CPU utilization when multithreaded DBT system is executed on multi-core physical machine. The goal of the novel architecture — multithreaded DBT system is to efficiently use multi-core resource to get more performance for DBT system. Additionally, to examine the bearing capacity of this new architecture, another experiment is done through scaling threads in DBT system, that is, whether this architecture can make more performance with gradually adding threads into DBT system is important. The tradeoff between threads (cores) number and overall performance is valuable, which can fully show the sensitive degree of this architecture for multi-core system.

Fig.8 shows that thread number can affect CPU utilization and system overall performance. With the increment of thread number, the performance of multithreaded DBT system is synchronously enhanced. Though the advantage from multi-core resource is large, the slope of curve that represents performance improvement goes to decrease gradually, that is, this advantage is also from bad to worse. This is mainly caused by thread layout, communication mechanism, and thread scheduling, etc. And the detailed threads layout for multithreaded DBT system is listed in Table 2. While the new architecture that has dual threads is rebuilt from traditional DBT system with only one thread, the new one has a good performance improvement. Meanwhile, this new architecture extremely reduces redundant overhead caused by interrupted processing for execution operation to cope with cache miss. Then based on this dual threads architecture, the harvester — multithreaded DBT system with 4 threads even benefits from optimization operation explained in (12). The burdensome overhead from optimization operation — building hot trace and profile is decreased via this operation executed in parallel with other threads. In this process, extra context switch is completely avoided, and the efficient communication mechanism brings another performance.

However, the fact that 8 threads or more are inserted into multithreaded DBT system cannot produce more performance but a little. Compared to 4-thread multithreaded DBT system, this one has the ability to efficiently translate basic block utilizing 5 translation threads. This is very beneficial for huge application procedure to amortize translation overhead. But this leads to one side-effect — thread scheduling. If this scheduling algorithm, such as BTM in this paper, is not efficient enough to accomplish scheduling between threads, the performance cannot be improved significantly. Moreover, the benchmark SPECint 2000 is only to test correctness, rationality and validity of the new architecture, but it cannot be deemed as general huge application.

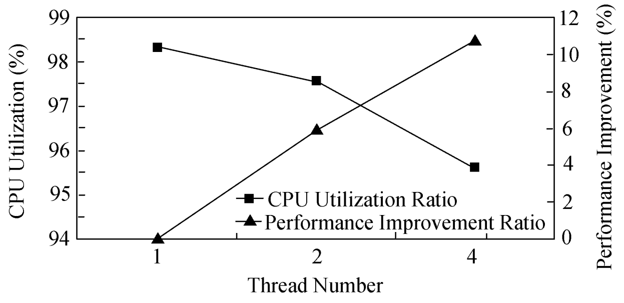


Fig.8. Relationship of thread number and CPU utilization\performance improvement.

Fig.8 also compares CPU utilizations of distinct multithreaded DBT system with different numbers of cores. As we know, each thread occupies one core. And the curve that stands for CPU utilization decreases from 98.32% to 95.58% in Fig.8, that is to say, the trend of it is declined absolutely. When the novel architecture has only one thread to do translation, execution and optimization operations, the performance cannot be good but with high-efficient CPU utilization. If translation operation is independently performed, extra context switch can be entirely obviated. But either the process of translation or execution operation cannot take the advantage of its core fully, since translation thread follows on-demand service for execution thread. If cache miss occurs, execution thread is hanged to wait for the target block translated by translation thread. In this process, the latency time leads to low-efficient CPU utilization. However, another two threads — profile thread and hot trace thread jointly give the system

**Table 2.** Detailed Threads Layout for Different Cores

| | One Core | Dual Core | Quad Core |
|---|---|---|---|
| Thread Number | 1 | 2 | 4 |
| Detailed Threads | Only one thread | Translation thread | Translation thread |
| | | Execution thread | Hot trace thread |
| | | | Profile thread |
| | | | Execution thread |

**Table 3.** More Translation Threads Layout on Multi-Core Physical Machine

| | One CPU (4-core) | Dual CPU (8-core) |
|---|---|---|
| Thread Number | 4 | 5/6/7/8 |
| Detailed Threads | Translation thread | 2/3/4/5 translation threads |
| | hot trace thread | hot trace thread |
| | profile thread | profile thread |
| | execution thread | execution thread |

more latency time. It is because building hot trace is not continuously executed, and its cache size is large enough to accommodate more hot traces. So the fact is that a lower CPU utilization is reasonable.

We should notice that this new architecture — multithreaded DBT system can not only bring another performance improvement for DBT system, but also achieve a good CPU utilization. Through evaluation above, we also know that it is able to adequately bear the pressure from incremental threads. We conclude that multithreaded DBT system can efficiently utilize multi-core resource and gain more performance, and it also takes an excellent extendibility with the development of CPU technique.

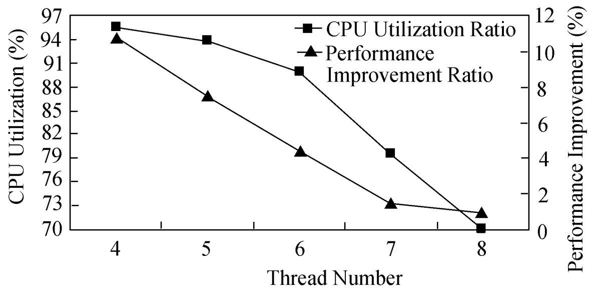


Fig.9. Impact of adding translation threads for CPU utilization and performance improvement.

We want to achieve another performance improvement through adding several translation threads into multithreaded DBT system, but this is handicapped. And the following reason is depicted in detail. The fact that 5 translation threads or more are inserted into multithreaded DBT system cannot produce more performance but a little. Compared to 4-thread multithreaded DBT system, ours has the ability to efficiently translate basic block utilizing 5 translation threads. This is very beneficial for huge application procedure to amortize translation overhead, but the experimental results cannot satisfy us. That is because lots of hardware resource employed by added translation threads cannot bring another performance improvement. Indeed, these extra threads are not able to work all the time. Even though they can be executed simultaneously assisted by the branch tree module, the scheduling overhead trigged is too high. From Table 1, we can see that hot trace is the most important element, since the multithreaded DBT system spends 90% (or more) of its execution time in these codes. However, these extra threads aim at translating basic blocks. So another performance improvement cannot be reached, and even it brings performance to descend compared to that of 4-thread multithreaded DBT system. Additionally, adding extra thread must bring extra overhead, which is trigged by the scheduling of branch tree, and extra memory accesses, etc. So we do another experiment on dual-CPU (4 core, 4 core, 2.66 GHz, 8 GB memory) physical machine, with the new architecture achieved by adding several translation threads, and then the threads are assigned to different cores through hardware affinity. In Fig.9, we can see that with the increment of translation threads, the performance enhancement is gradually decreased, and the CPU utilization also drops down significantly. First, the CPU utilization is decreased from 95.58% to 64.39%, since more translation threads are idle to wait for next translation work, and even if the branch tree module as the scheduler has the responsibility to assign different translation work to each translation thread (the scheduling overhead cannot be ignored). Then the asynchronous waiting time and scheduler must cost lots of resource (they are too expensive), and more writing operations applied on each private cache, that is, many memory accesses existing, bring extra overhead as well. In conclusion, all kinds of overheads mentioned above drastically impact on overall performance, and even bring it down. We can see that adding extra translation thread is not a good idea to enhance entire performance.

So we move to multithreaded execution part, also called compiler part. Although it is difficult, this is our research work currently.

## 5 Related Work

Dynamic binary translation is an invaluable technique for translating one ISA (instruction set architecture) to another ISA transparently. Until recently, there exist many of popular dynamic binary translators such as StarDBT[7], QEMU[5], and FX!32[33]. All these binary translators can be simply divided into two types. One type can only translate single ISA into another specific ISA like StarDBT (from IA32 to IA32), FX!32 (from x86 to Alpha), DAISY (from PowerPC to VLIW)[34], BOA (from PowerPC to EPIC)[35], and even the Transmeta Crusoe processor (from IA32 to VLIW)[36]. The other type is designed to be resourceable and retargetable[27] to support the translation between various instruction sets, such as Strata[37], Walkabout[38], and UQDBT[6]. Typically, the UQDBT dynamic translator, which is based on the famous static UQBT[39] framework, uses specifications to specify the guest/host architectures at various levels of abstraction, and simultaneously completes binary translation dynamically by going through several intermediate representations. One significant advantage of resourceability and retargetability is that it can be applied to a variety of hardware platforms transparently. However, the performance of binary translators attributed to the latter type is not very well, like QEMU[5] (the execution time of it is even 3∼4 times longer than that of the native machine).

A large set of dynamic optimization systems are designed to be partly or completely transparent to the software, the hardware, as well as the user of the system. These systems are characterized by a native optimization engine that is designed to tailor the application to its runtime environment without any intervention or special preparation by the user or application writer[27]. There have been several software dynamic optimization systems, such as Dynamo, DynamoRIO, and Mojo. Dynamo[18] is implemented entirely in software and its operation is transparent which means no programmer's assistance is required at runtime. It observes the program's behavior through interpretation to dynamically select hot instruction traces from the running program. Several successors to Dynamo have since surfaced, such as DynamoRIO[40], which executes on IA-32 machines running Windows or Linux developed by HP and MIT. Mojo[41], which is similar to Dynamo, is one of the first dynamic optimizers to specifically target large, interactive Windows applications. A common attribute of these systems mentioned is that the optimization is typically performed in the same thread as the main execution within a single hardware context, or optimization is sometimes executed in another thread utilizing special hardware. However, sharing the same hardware context requires pausing the current program's execution to perform optimization. This also introduces additional runtime overhead due to heavy weight user-level context switching between execution, profiling, and optimization[11]. Meanwhile, some researchers focus on the special hardware support for better performance, so gradually, there appear co-designed dynamic optimization systems, such as ADORE[10], Trident[11]. The binary optimization framework ADORE based on hardware profiling mechanism proposed by Jiwei Lu *et al.* focus on using performance monitoring hardware to detect the hot regions and bottlenecks instead of instrumentation. The Trident is the closest runtime multithreaded framework to our new architecture. But it employs hardware support to identify the hot path during the execution of the program, and simultaneously uses spare threads which are called helper threads on a multi-core processor to perform dynamic optimizations.

## 6 Conclusion and Future Work

With the development of CPU technique, more and more researchers focus on how to efficiently utilize multi-core resource. Dynamic binary translation system also employs multi-core power to make higher performance. Therefore, we have presented a novel multithreaded dynamic binary translation system which utilizes several threads to concurrently perform translation, execution and optimization operations. As well, multithreaded DBT system on multi-core system has many barriers to overcome. In the process of building multithreaded DBT system, several daunting challenges have been encountered. The first one is cache layout. To address this arduous problem, private code caches are employed to better solve mutual exclusion between threads. Furthermore, ALCM is competent for the task of passing hot spot information between profile thread and hot trace thread. Finally, a new efficient thread scheduling method — BTM is introduced, which can better solve mutual exclusion between translation threads. Through many experiments, multithreaded DBT system are proved that it gives us a chance to control multi-core resource to achieve higher performance with better CPU utilization.

In the future work, the execution part of multithreaded DBT system will be executed concurrently to achieve higher performance. It also involves the better branch prediction mechanism. Furthermore, optimization methods for execution thread will also be the research focus. And as the number of processor cores grows, we expect to have more experiments on these new platforms.

**Acknowledgments** We appreciate Drs. Liang Liu and Ying Chen's (now at IBM CRL in Beijing, China) contributions to the early version of this paper. We also thank the anonymous reviewers for their valuable and constructive comments that help us to improve greatly the quality of the paper.

## References

[1] Hu W W, Hou R, Xiao J H, Zhang L B. High performance general-purpose microprocessors: Past and future. *Journal of Computer Science and Technology*, 2006, 21(5): 631-640.

[2] Wells P, Chakraborty K, Sohi G. Dynamic heterogeneity and the need for multicore virtualization. *ACM SIGOPS Operating Systems Review*, 2009, 43(2): 5-14.

[3] Tera-scale research prototype: Connecting 80 simple sores on a single test chip. ftp://download.intel.com/research/platform/terascale/tera-scaleresearchprototypebackgrounder.pdf, Jan. 10, 2010.

[4] Moore R W, Baiocchi J A, Childers B R, Davidson J W, Hiser J D. Addressing the challenges of DBT for the ARM architecture. In *Proc. LCTES*, Dublin, Ireland, Jun. 19-20, 2009, pp.147-156.

[5] Bellard F. QEMU, a fast and portable dynamic translator. In *Proc. USENIX ATC*, Anaheim, USA, Apr. 10-15, 2005, p.41.

[6] Ung D, Cifuentes C. Machine-adaptable dynamic binary translation. In *Proc. DYNAMO*, Boston, USA, Jan. 18, 2000, pp.41-51.

[7] Wang C, Ying V, Wu Y. Supporting legacy binary code in a software transaction compiler with dynamic binary translation and optimization. In *Proc. CC*, Budapest, Hungary,

Mar. 29-Apr. 6, 2008, pp.291-306.

[8] Kondoh G, Komatsu H. Dynamic binary translation specialized for embedded systems. In *Proc. VEE*, Pittsburgh, USA, Mar. 17-19, 2010, pp.157-166.

[9] Payer M, Gross T. Fast binary translation: Translation efficiency and runtime efficiency. In *AMAS-BT*, Austin, USA, June 20, 2009.

[10] Lu J, Chen H, Yew P, Hsu W. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 2006, 6: 1-24.

[11] Zhang W F, Brad C, Tullsen D M. An event-driven multithreaded dynamic optimization framework. In *Proc. PACT*, Saint Louis, USA, Sept. 17-21, 2005, pp.87-98.

[12] Hazelwood K, Lueck G, Cohn R. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proc. ISMM*, Dublin, Ireland, Jun. 9-20, 2009, pp.20-29.

[13] Adams K, Agesen O. A comparison of software and hardware techniques for x86 virtualization. In *Proc. ASPLOS*, San Jose, USA, Oct. 21-25, 2006, pp.2-13.

[14] Baraz L, Devor T, Etzion O, Goldenberg S, Skalesky A, Wang Y, Zemach Y. IA-32 execution layer: A two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. In *Proc. MICRO*, San Diego, USA, Dec. 3-5, 2003, pp.191-201.

[15] Cmelik R F, Ditzel D R, Kelly E J, Hunter C B, Laird D A, Wing M J , Zyner G B. Combining hardware and software to provide an improved microprocessor. US Patent # 6031992, 2000.

[16] Klaiber A. The technology behind Crusoe processors. Transmeta Technical Brief, 2000.

[17] Li T, Liang A, Liu B, Lin L, Guan H. A hardware/software codesigned virtual machine to support multiple ISAS. In *Proc. AMSBT*, Beijing, China, Jun. 21, 2008, pp.38-44.

[18] Bala V, Duesterwald E, Banerjia S. Dynamo: A transparent runtime optimization system. In *Proc. PLDI*, Vancouver, Canada, Jun. 18-21, 2000, pp.1-12.

[19] Wang C, Wu Y, Araujo G. Software-based transparent and comprehensive control-flow error detection. In *Proc. CGO*, New York, USA, Mar. 26-29, 2006, pp.333-345.

[20] Luk C, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi V, Hazelwood K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI*, Chicago, USA, Jun. 12-15, 2005, pp.190-200.

[21] Qin F, Wang C, Li Z, Kim H S, Zhou Y, Wu Y. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *Proc. MICRO*, Orlando, USA, Dec. 9-13, 2006, pp.135-148.

[22] Wu Q, Reddi V, Wu Y, Lee J, Conners D, Brooks D, Martonosi M, Clark D. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proc. MICRO*, Barcelona, Spain, Nov. 12-16, 2005, pp.271-282.

[23] Sridhar S, Shapiro J S, Bungale P P. HDTrans: A low-overhead dynamic translator. *ACM SIGARCH Computer Architecture News*, 2005, 35(1): 135-140.

[24] 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 *Proc. CGO*, San Jose, USA, Mar. 11-14, 2007, pp.61-73.

[25] 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*, 2007, 42(5): 3-9.

[26] SPEC CPU2000 documentation, http://www.spec.org/osg/cpu2000/docs/, Jan. 10, 2010.

[27] Hazelwood K, Smith M D. Managing bounded code caches in dynamic binary optimization systems. *ACM Transactions on Architecture and Code Optimization*, 2006, 3(3): 263-294.

[28] Stallings W. Operating Systems: Internals and Design Principles. Sixth Edition, Prentice Hall, 2008.

[29] Sun Y, Zhang W. Improving Java performance and energy dissipation through efficient code caching. *Design Automation for Embedded Systems*, 2009, 13(3): 179-192.

[30] Baiocchi J, Childers B R. Heterogeneous code cache: Using scratchpad and main memory in dynamic binary translators. In *Proc. DAC*, San Francisco, USA, Jul. 26-31, 2009, pp.744-749.

[31] Hazelwood K, Smith M D. Code cache management schemes for dynamic optimizers. In *Proc. INTERACT*, Sydney, Australia, Jul. 21-25, 2002, p.102.

[32] Hazelwood K. Code cache management in dynamic optimization systems [Ph.D. Dissertation]. Harvard University, May, 2004.

[33] Chernoff A, Herdeg M, Hookway R, Reeve C, Rubin N, Tye T, Yadavalli S B, Yates J. FX!32: A profile-directed binary translator. *IEEE Micro*, 1998, 18(2): 56-64.

[34] Ebcioglu K, Altman E R. DAISY: Dynamic complication for 100% architectural compatibility. In *Proc. ISCA*, Denver, USA, Jun. 2-4, 1997, pp.26-37.

[35] Altman E R, Gschwind M, Sathaye S, Kosonocky S, Bright A, Fritts J, Ledak P, Appenzeller D, Filan Z. BOA: The architecture of a binary translation processor. IBM Research Report RC 21665, 1999.

[36] Dehnert J C, Grant B K, Banning J P, Johnson R, Kistler T, Klaiber A, Mattson J. The Transmeta Code Morphing Software: Using speculation, recovery, and adaptive retranslation to address real-life challenges. In *Proc. CGO*, San Francisco, USA, Mar. 23-26, 2003, pp.15-24.

[37] Scott K, Kumar N, Velusamy S, Childers B, Davidson J W, Soffa M L. Retargetable and reconfigurable software dynamic translation. In *Proc. CGO*, San Francisco, USA, Mar. 23-26, 2003, pp.36-47.

[38] Cifuentes C, Lewis B, Ung D. Walkabout — A retargetable dynamic binary translation framework. In *Workshop on Binary Translation*, Charlottesville, USA, Sept. 22-25, 2002.

[39] Cifuentes C, Emmerik M. UQBT: Adaptable binary translation at low cost. *Computer*, 2000, 33(3): 60-66.

[40] Bruening D, Duesterwald E, Amarasinghe S. Design and implementation of a dynamic optimization framework for Windows. In *Workshop on FDDO*, Austin, USA, Dec. 1, 2001.

[41] Chen W K, Lerner S, Chaiken R, Gilles D M. Mojo: A dynamic optimization system. In *Workshop on FDDO*, Monterey, USA, Dec. 10, 2000.

**Ru-Hui 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.

**Hai-Bing 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 andhardware/software co-design.

**Er-Zhou 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.

**Hong-Bo Yang** is currently a Ph.D. candidate at Shanghai Jiao Tong University, China. He received the M.S. degree in 1995 and B.S. degree in 1998 from Institute of Airforce Meteorologyity, China. His main research interests are in virtual machines, computer architecture and compiling.

**Yin-Dong Yang** is currently a Ph.D. candidate at Shanghai Jiao Tong University, China. He received the M.S. degree at School of Computer, Electronics and Information from Guangxi University in 2007, China. In 2004 he received his B.S. degree at School of Information and Technology from Jiangnan University, China. His main research interests are in virtual machines, computer architecture and compiling.

**A-Lei 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.