A Dynamic-Static Combined Code Layout Reorganization Approach for Dynamic Binary Translation

Haibing Guan, Erzhou Zhu, Kai Chen, Ruhui Ma, Yunchao He, Haipeng Deng and Hongbo Yang
Department of Computer Science and Engineering & Shanghai Key Laboratory of Scalable Computing and Systems
Shanghai Jiaotong University, Shanghai, China
Email: {hbguan, ezzhu, kchen, ruhuima, ventureheart, denghipson, yanghongbo819}@sjtu.edu.cn

***Abstract*—Dynamic binary translation (DBT) has attracted much attention as a powerful technique for the runtime adaptation of software among different ISAs. It offers unprecedented flexibility in the control and modification of a program during the runtime. However, its inherent high overhead has perplexed researchers for many years. In order to reduce the overhead of DBT, this paper presents a dynamic-static combined approach to reorganize the layout of software cache. Under this approach, we first employ an emulating execution to collect the profile information and the translated target code. Especially, the path of execution flow will be tracked. In the static phase, based on the profile information collected in the previous stage, we first use the method of code replicating to build the traces, and then reorganize the layout of the target code by putting the hottest traces at the top of the software cache. Because of exact prediction and improved locality, the execution stream will concentrate on a small area with less control transfer. This approach can greatly reduce the overhead of DBT on the condition that the program runs repeatedly. Experimental results on executing the SPEC 2000 benchmarks show that our approach can reduce more than 30% run time on average.**

***Index Terms*—Dynamic binary translation, profile information, static optimization, code replication, trace building**

I. INTORDUCTION

Dynamic binary translation (DBT) has attracted much attention as a powerful technique for dynamically adapting software among different ISAs [1]. It offers unprecedented flexibility in the control and modification of a program during the runtime. The technique of DBT can be used in emulating an ISA to a new ISA, monitoring and optimizing performance at runtime, providing resource protection and management, virtualizing resources, and detecting security attacks and so on. In the past decades, a lot of dynamic binary translators have been designed for different goals: IA-32 EL [2] and Aries [3] can migrate applications cross different ISAs; Pin [4], Dynamo [5], Valgrind [6] and HDTrans [7] can detect the behaviors of the programs as well as optimize the programs at runtime; Shade [8] and QEMU [9] use DBT to speed up architecture simulations; DAISY [10] is designed for creating a virtual execution environment from a totally different architecture at ISA level; ADORE [11] is designed as a dynamic binary optimization system.

However, the inherently high overhead feature of DBT has perplexed many researchers for many years. Generally speaking, the overhead incurred in DBT can be divided into two parts: the overhead of translating procedure and the overhead of executing the target code. As a matter of fact, most of the present DBT systems can reduce the translating cost to an acceptable range. Hence, the main attention is laid down on the reducing execution time of target code. In order to reduce the time consuming of executing the target code, many optimizations has been proposed, such as translation block chaining, forming larger translation blocks (superblocks), reordering translated instructions to improve pipeline performance, and borrowing optimization techniques from traditional compilers. In fact many binary-code-specific optimizations are seriously depend on the profile information gathered during the runtime. The richness and correctness of profile information can directly determine which kind of optimization can be implemented and the extent of its efficiency. Profiling is a process for dynamically collecting program information (instructions and data statics) that is used to guide the optimization during the translation process. However, this inevitably leads to performance losing, especially the complex one.

By introducing the static analysis stage into dynamic binary translation, the overhead of profile and dynamic optimizing are reduced remarkably. Under this scenario, the complex profile is collected by the first emulating phase. Then efficient optimization algorithms based on profile information collected by the previous stage are available. The overhead of the subsequent executions will be small, since they can directly execute the optimized target code generated by the static optimization stage. Generally speaking, the method of combines the dynamic translation with the static analysis at least has the following merits: firstly, it can spare the translation time.

Corresponding author: Erzhou Zhu (ezzhu@sjtu.edu.cn)

doi:10.4304/jsw.6.12.2341-2349

The process of translating source code to target consumes a substantial part of time. Once the source code is translated at first run, then the target code will be saved in special files which can be directly loaded at the next runs. Secondly, profiling overhead will be eliminated except for the first run. If all the optimizations are performed statically, the profiling process is not needed anymore. Lastly, complex optimizing algorithms are available now. Since many optimizations are carried out statically instead of at runtime, complex algorithms which are not appropriate dynamically are available now without warring about overhead.

Actually, the approach of employing static phase to aid dynamic binary translation assumes that the execution profile of each block in the profile phase (initial emulating execution phase) is representative of the block throughout its lifetime. In particular, a region is selected for optimization with the assumption that it infrequently takes its side exits and is thus candidate for advanced optimizing. If these assumptions are not hold, however, the performance of the program will be suffered [12].

In DBT systems, the native code that CPU needs to execute is stored in software maintained cache. By reorganizing the layout of software cache, the overhead of dynamic binary translators are reduced remarkably. Once the target code in software cache is reframed properly, hot code will be gathered together and organized properly. Therefore, the execution stream will concentrate on a small but hot area with less happening of control transfer. As a whole, the method of reorganizing the layout of software cache can: (1) decreases the execution of jump instructions, (2) makes the pipeline going on with fewer interruptions, (3) reduces physical cache miss rate, (4) cuts down TLB access miss rate, and (5) decreases the page faults.

This paper presents a new dynamic-static combined approach for reorganizing the layout of software cache in DBT system. This approach is based on the static-integrated optimization framework appeared in [30], and the DBT system that our approach is going to optimize is Crossbit [29]. As a whole, the approach contains three stages to fulfill its optimization target. In the initial phase, we employ the emulating execution to collect profile information and the translated target code. Especially, the path of each execution flow will be tracked, which is used to build the trace in the static analysis phase. In the static analyzing phase, based on the profile information and target code that are collected from previous stage, we first use the method of code replicating to build traces, and then reorganize the layout of the target code by putting the hottest trace at the top of the software cache. At the last stage, i.e. the subsequent executions, due to the profile information and the optimized target code are all available the overhead will be remarkably reduced. Different from our original approach [27], this approach introduces a replicate-based trace building algorithm in the static analysis phase. By code replication, more hot traces will be detected and generated in the memory space, through which programs could execute more continuously and therefore performance promotion will be achieved.

The remainder of this paper is organized as follow. Section II introduces a brief overview of the DBT system (Crossbit) that out approach is going to optimize. Section III presents the detail implementation of the approach, which includes the overall workflow of our approach, traces building, code replicating and code layout reorganizing. Section IV gives the performance evaluations. Some related works are presented in section V. At last, we conclude our work.

II. BACKGROUNDS

Crossbit is the target DBT system that our approach is going to optimize. It is designed and implemented as a multi-source architectures and multi-target architectures dynamic binary translator, which aims at fast migrating existing executable source code from one platform to another alien target platform with lower cost. Until recently, it has fully or partially supported source platforms including SimpleScalar, IA32, MIPS, SPARC, and supported target platforms such as IA-32, Power PC and SPARC. The operating system that Crossbit support is Linux. In order to support code translation among multi-sources and multi-targets better, a new Intermediate Instruction set—VInst [31], which is independent of any specific machine instructions, has been introduced. Unlike many other existing DBTs which directly translate the binary code of one instruction architecture (ISA) to another ISA, Crossbit first converts source binary code to VInst specifications and then transforms them into target platform code, using a granularity of a basic block (BB) as the basic unit of translation. The detail introduction of Corssbit is described in [29].

Software-managed code cache, also called software cache or code cache for short in Crossbit, is very critical in Dynamic Binary Translator. It usually occupies an area of main memory and stores translated native code, making the translator reuse the native code to avoid the overhead of retranslation during the whole execution. Therefore, it can remarkably improve the performance of DBT system. In order to remove noise in performance comparison and analysis, we suppose the size of Software Cache is unbounded. It can store all of the translated code without any replacement policy. Then in Crossbit, the running time of execution over translated code takes more than 97% on the whole according to the benchmark of SPEC2000, while the cost of initialization, translation and optimization is trivial.

In Crossbit, many techniques are employed to improve the quality of translated code in an effort to develop the performance:

(1) Linking between blocks to reduce the incidence of returns to Crossbit
(2) Changing indirect jump into several compare and direct jump in translated code to reduce the context switch to Crossbit when executing the translated code.

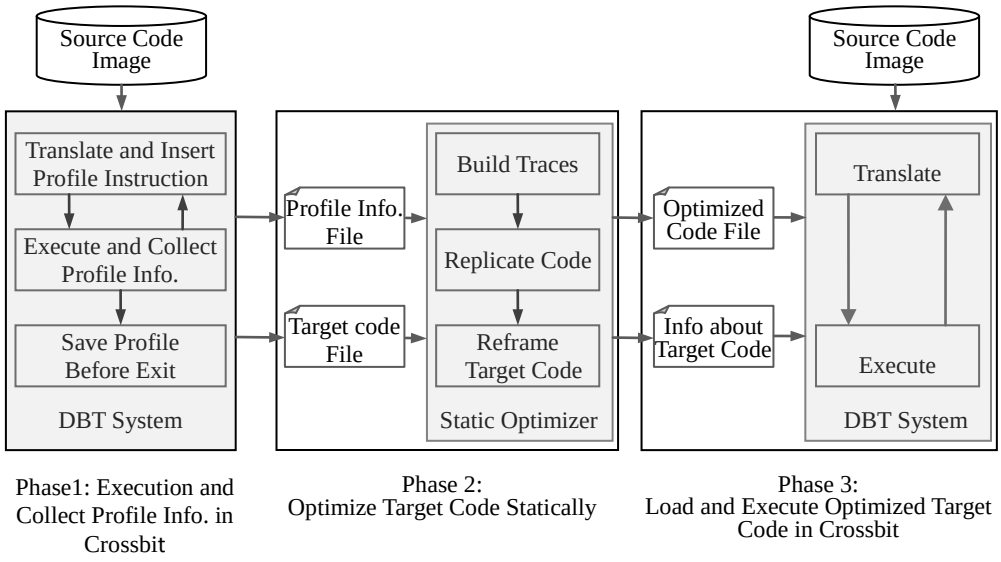

Figure 1. The Workflow of Our Approach

(3) Building superblock according to profile.
(4) Redundant code elimination.

These conventional techniques can greatly develop the performance of Crossbit, making it comparable to QEMU. To do further optimization, we present the method of dynamic-static combined code layout reorganization in software cache.

III. IMPLEMENTATIONS

A. Overview

As Fig.1 demonstrates, the overall workflow of our approach can be divided into three phases: dynamically collects profile information at the first emulating execution phase, performs profile-directed optimizations (trace building and code layout reorganizing) in static analysis phase, and loads the target code that has been optimized offline for the subsequent executions.

A) Emulating Execution

In this phase, the DBT executes the source program dynamically. Instrumentations are deployed to collect the profile information. Profile information contains the execution times of each block, the incoming blocks and outcoming blocks of each block, the time of each jump direction, and so on. Especially, the path of each execution flow will be tracked, which is used to build traces in the static optimization stage. After the execution of the source image, the profile information will be saved in profile file. Specially, in order to spare the execution time of the subsequent execution, the translated target code will also be saved (in target code file) for the purpose of optimizing in the static analysis phase.

B) Static Analysis

Since it is needn't taking much consideration of analyzing overhead in the static analysis phase, sophisticated optimizations can be deployed. We can perform the optimizations than can take full advantage of rich profile information, such as conditional branch directions prediction, inline and build superblocks according to the execution times of current blocks and its hottest outgoing edges. In this paper, we focus on trace building based on the method of code replication and translated code layout reorganizing. In this approach, hot traces are identified by their head blocks, and the granularity of the code to be reorganized is basic block. At the end of the static analysis stage, the hottest traces are placed at the top of the software cache, while the coldest ones are placed at the bottom.

C) Subsequent Execution

In this stage i.e. the subsequent executions, the operations of loading (profile file and target code file) and initializing are carried simultaneously. The main work flow of DBT is not changed, which means that the binary source image should be loaded as normal. The difference is that the target code is directly loaded from target file instead of been translated from source image. When execution starts, DBT tries to find the optimized code other than source code. If the target code that should be executed now has existed, DBT just executes it directly. Otherwise, DBT will translate the source code as normal. This approach can greatly reduce the overhead of DBT on the condition that the program runs repeatedly.

In the overall workflow of our approach, stage 2 (i.e. the static optimization stage) is the most important one. It first takes the profile information and the translated code as the input, then builds traces by employing the method of code replicating, at last, reorganizes the layout of code cache by putting the hottest traces at the top area.

B. Trace Building and Code Replicating

Hot trace building plays an important role in enhancing the performance of dynamic binary translation. As a matter of fact, in most cases only 10% of code takes 90% of execution time of the whole program. Hot traces can promote the code position to make better the locality of the code, and therefore programs can achieve a better performance. In conventional, there are many traces in a

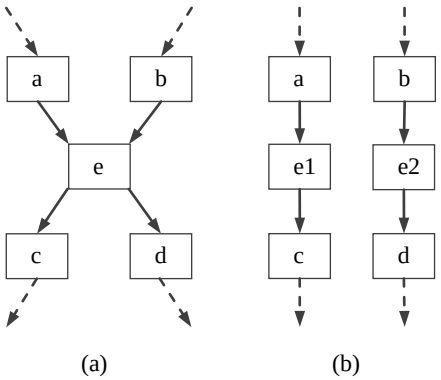

Figure 2. An Example of Code Replication.

program, control transfers are frequently occurred from one to another. In DBT or other dynamic systems, trace building is a different issue, which must take a balance between the runtime overhead and efficiency. So how to enhance the quality of the trace fragment, which is a sequence of frequently-executed basic blocks in the memory, is quite important. At present, many trace building algorithms have been developed which can detect hot traces accurately. However, the high overhead make them less useful when applied in dynamic systems. In additional, there is a common situation in which a block is shared by many traces, only one of them is maintained intact while the others will be truncated into many short trace fragments. These short trace fragments will be produced in memory space and the instruction locality will get worse.

To overcome the problems mentioned above, our approach employs a different strategy. In this strategy, we statically build traces based on the profile information that has been collected from the dynamic emulating execution stage, and employ the method of code replication to resolve the problem that one block is shared by many traces. By employing the dynamic-static combined method of trace building, the unbearable high overhead of DBT is turned away. It is need to be mentioned that, in this situation, the process of trace building and code replicating are on the level of logical, the works that really happen are in reorganization stage.

A) Trace Building

Since the translated code (organized as basic blocks) and its corresponding profile information are dynamically collected from the emulation stage, the process of trace building can be easily performed during the static phase. We start our approach by traverse the files that contain profile information and target code, and then employ the following two steps to build traces: the first step is to identify the trace head. The head block of a trace is determined by the Begin-Threshold, a threshold value that we predefined. When the execution time of a basic block exceeds the Begin-Threshold, it will be set as a trace head. Secondly, when the trace head is identified, the trace building procedure will be called to build trace. As we know, when one block becomes hot, the other blocks surrounding this hot block also seems to be hot. Consequently, the exit block of the current block that has the maximum execution time will be selected as a part of the current trace and also be labeled as “trace-men”. When the maximum execution time of the exit block is blow the Finish-Threshold (usually a multiple of the Begin-Threshold) or the block to be appended is already labeled as “trace-men” the trace building procedure will be closed, and thus the trace fragments emerged.

B) Code Replicating

The method of trace building in section A) will produce many trace fragments. In that situation, when a block is shared by different traces, only one of these traces is maintained intact, while the others will be truncated into many short trace fragments. These short fragments will be placed in the memory and the instruction locality will get worse. To handle this problem, we present a method of replicating of the very block that been shared by many different traces, and thus improve the instruction locality as well as memory continuity.

Fig.2 gives a simple example of code replication. Fig.2 (a) supposes the situation that both block a and b are jump to e, and then e passes the execution to c and d respectively. In this case, four traces might be constructed: ⋯a→e→c⋯,⋯a→e→d⋯,⋯b→e→c⋯,⋯b→e→d⋯. This way, the execution stream will be interrupted more frequently, and exhibits bad locality characteristics. According to our approach, we make a copy of block e when the second trace is built, like ⋯b→e2→d⋯. By doing this, the number of trace fragments is reduced, and the length of trace fragments is increased. Then two longer traces will be constructed (as Fig.2 (a)) which display a good instruction locality and allow the execution stream to run more continuously. However, this method also introduces a side effect of memory expansion, this side effect will be detail discussed in section IV.

C) Trace-Table

Since the works performed in this section are all on the level of logical. We just record the trace information and the replicated blocks. The works that really happen are in reorganization stage. In order to record this information, we maintain a Trace-Table that records the trace information, such as the SPC of the head block (identifies the trace) and its corresponding execution time (identifies the frequency of the trace), the SPC of the remainder blocks of the trace and so on.

C. Code Layout Reorganization

As mentioned previously, the scale of the software cache in Crossbit (the very DBT system our approach is applied to) is been set unbounded. By doing this, it is only needs a little time to initialize and translate. Additionally, reorganizing code layout of software cache can improve the performance on the ground that the execution stream will be more approximate to its control flow graph after reframing. On the target code in software cache is reframed properly, hot code will be gathered together and be well organized. Because of exact prediction and improved locality, the execution stream

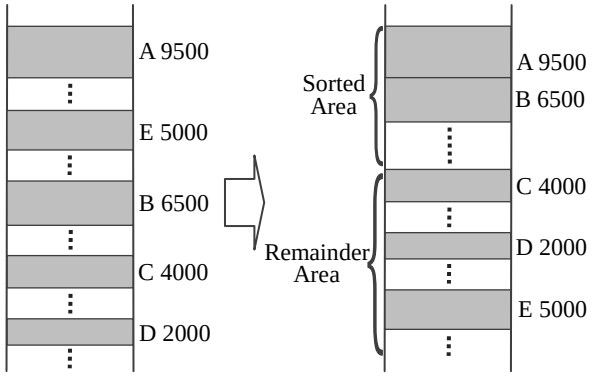

Figure 3. Sorting the head blocks according to their execution frequency.

will concentrate on a small area with less control transfer. The work of code layout reorganization is followed by the work of trace building and code replicating.

A) Sort Target Traces and Blocks

Prior to perform the work of code reorganizing, is has to sort the target traces and the remainder target blocks (scattered block we called) that have been generated previously. Since traces are identified by their head blocks, we just need to sort the head blocks. The right figure in Fig.3 displays the code that has been sorted from the left one.

B) Reorganize Code Layout

Since a trace is identified by its head block. If the head block of a trace is placed on the top of code cache, then the remainder blocks of the trace will be placed adjacent to their head blocks. Fig.4 demonstrates the change of the code layout. Supposes A, E, C are built into a trace, A is the head block of this trace, and A is the hottest block, then all of them will be placed on the top in spite of block B is more hotter than most of blocks in this trace.

IV. Performance Evaluations

In this section, we first carry out some experiments on comparing the overall performance of our approach with the original version of Crossbit (the DBT system that our framework based from) and QEMU (a fast and portable open source DBT system). Then the experiments on verifying the priority (compared with original Crossbit)

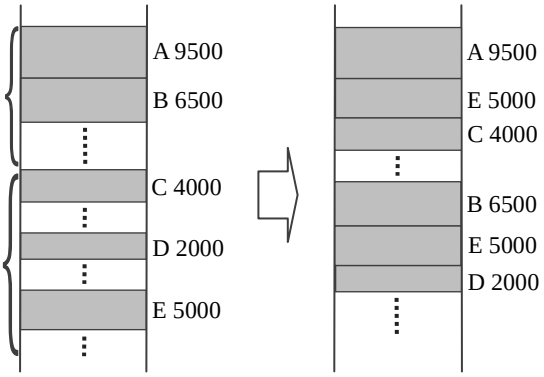

Figure 4. Reorganizing the Layout of Software Cache

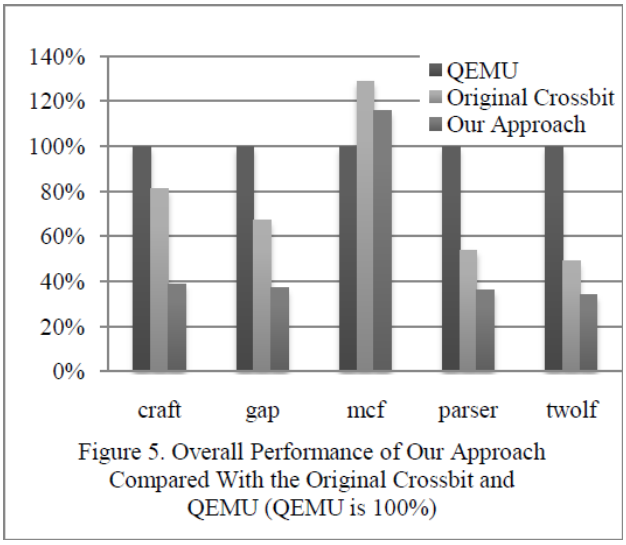

Figure 5. Overall Performance of Our Approach Compared With the Original Crossbit and QEMU (QEMU is 100%)

of our approach are performed. At last, we emphasize the side effect of our approach that incurred by code replication. We carry out these experiments on an Intel Pentium 4 dual-core machine, where each core has a 2.8GHz Pentium with an 800 MHz processor bus, 32KB of L1 cache and 1.5MB L2 cache. The machine's memory system uses a 533MHz bus with 1.5GB of dual interleaved DDR SDRAM memory. We use the SPEC CPU2000 benchmarks as the test suite. Some of the benchmarks cannot run successfully on our approach, which might due to the base Crossbit lacking of complete support for all Linux System calls, or there are some errors in address relocation and linking of exits stub after code layout reframing, our team still deals with these issues now. So the comparisons and analysis are based on these programs that can be right executed on our approach.

A. Overall Performance Comparison

Fig.5 shows the overall comparison of performance among our approach, original version of Crossbit and QEMU [9] (a fast and portable open source DBT system). When compared with original version of Crossbit, our approach displays consistently better performance than that of Crossbit for all of the tested benchmarks. And also outperforms of QEMU for nearly all the tested benchmarks. The overall executing time of our approach is reduced by 34% on average relative to the original version of Crossbit, and more than 50% to QEMU. The better performance we gained largely due to static phase optimization and partly because of directly loading optimized target code to the following execution.

Under our approach, fewer blocks are needed to be translated than the original Crossbit. Since the size of software cache of our approach is set unbounded, there would be no retranslation. Experimental statistic also shows that the overall consumption of initialization, optimization and translation are no more than 3%. So the reduction of the execution time, compared with original Crossbit, is about 31% in SPEC 2000. When compared with QEMU, our approach is much faster except for the mcf benchmark, the reason of this defect is described in [29].

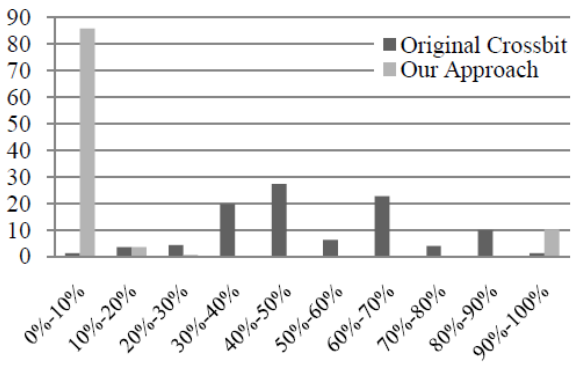

Figure 6. Code Execution Frequency Distribution.

The following two experiments are performed to verify the reasons of high performance we gained from static optimizations.

A) Code Execution Mainly Concentrates on A Few Hot Code

It is well known that when a program is executed, the code execution frequency is often not evenly distributed. A small portion of the hot code usually occupies most of the execution time, while a large portion of other code is to cover all eventualities but executed little. So when the execution focuses on a small portion of the hot instructions, the overhead will be decreased. Fig.6 gives the experiment's results to show the code execution frequency distribution our approach and original version of Crossbit. From the results, we can conclude that the code execution frequency is evenly distributed under original Crossbit. However, under the schema of our approach, more than 85% execution time mainly concentrates on 10% hot code.

B) Less Control Transfer Occurrence

Longer traces can greatly reduce the occurrence of control transfer. Less transfer control occurrence in the same program can enhance the accuracy of instruction pre-fetch, reduce interruption of pipeline, and finally introduce better performance. Fig.7 and Fig.8 show the direct jump occurrence and indirect jump occurrence respectively. In the figures the jump occurrence in original version of Crossbit is supposed as 100%. From the figures we can conclude that our approach can greatly reduce the occurrence of control transfer no matter it is a

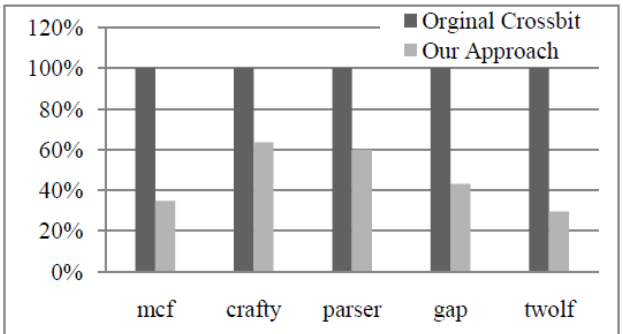

Figure7. The occurance of direct jump (Original Crossbit is 100%)

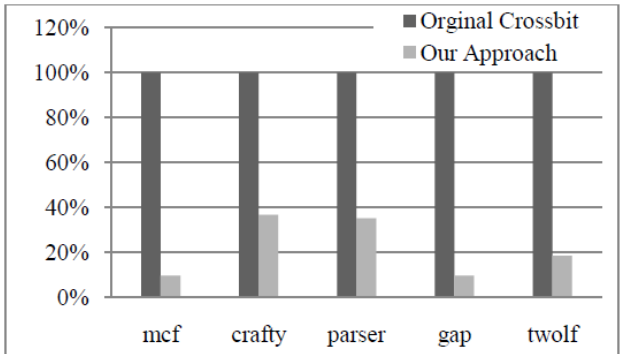

Figure 8. The occurance of indirect jump (Orginal Crosssbit is 100%)

direct jump or an indirect jump. This because our approach can find hot spots and hot traces of the program, and then build longer traces to fulfill loop without transfer control change.

B Side Effect of Our Approach

Lastly we talk about the side effect of code replication: memory expansion. Fig.9 plots the memory expansion ratio (MER) of code replication. It shows the maximum MER of nearly 6% when running the crafty test case, and an average expansion ratio of 2.25%. Particularly, in the case of mcf, we get a performance promotion of 34%, with only a memory expansion of less than 2%. We can find that this code replication is quite suitable for mcf to enhance performance. On the situation of crafty, we can't ignore the memory expansion sacrificed to performance promotion.

V. Related Works

For the purpose of gaining better performance, the static process has been adapted in many binary translation systems. FX!32 [13] is a profile-directed binary translator, which combines emulation (first run) and binary translation (subsequent runs) techniques. It possesses a database to accommodate the target code, which is generated by the background emulator according to the online profile information. Then the target code is available at the subsequent runs of the source image.

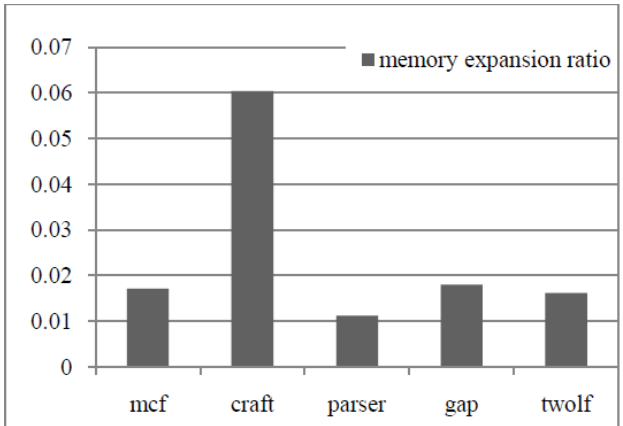

Figure 9. Memory Size Expansion Ratio (compare with original Crossbit)

IA-32 Execution Layer (IA-32 EL) [2] is a two-phase dynamic binary translating software designed for supporting IA-32 applications on Intel Itanium family systems. In IA-32 EL, the first phase (also called cold code translation) is designed to be fast, with minimal optimizations and overhead and uses instrumentation to identify hot spots. The second phase (also called hot code translation), retranslates and further optimizes the hotspots identified by the cold translation. Metadata driven DBT [14] proposes a novel method of passing performance critical information to dynamic binary translator through metadata. By metadata, the dynamic binary translator is able to perform aggressive optimizations to generate high quality code. This approach behaves well if the source programs (written in high level language) are available.

Static process has also been used in conventional compilation systems. GCC [15] requires the programs to be built and executed twice. In the first time, the applications are compiled to generate profile information. And the second time, the applications are compiled and use the profile information that was generated by the first execution.

Efficient software cache Management is an important mean to improve the performance of DBT. In [16] and [17], the authors discussed the replacement policy of code cache. They compared several strategies of code cache replacement policy and presented the appropriate granularity of code cache eviction size, trying to protect more hot code from swapping out when the software cache is limited. Moreover, in an effort to cut down the memory consumption of code cache, they elaborated several methods [18] to reduce exit stub that is used to handle the egress of control transfer. [19], [20] and [21] presented the schemes to keep the software cache consistent between the code cache and the original code, to keep it transparent to user and operating system, to dynamically bound code cache size to match the current working set of the application when it is running, and to make software cache becoming process-shared when this merit is necessary.

For most DBTs, the trace is the unit of choice of function and method. Traces are found to perform well and be easy to translation [22]. Many trace detecting methods have been employed to improve the performance. Next-Executing-Tail (NET) is a popular trace selection algorithm [23]. And it attempts to select traces that begin at loop headers by considering only instructions that are the targets of backward branches or exits from existing traces [24]. But it cannot span inter-procedural cycles and the nested loops. Mueller and Whalley [25] have applied code replication to avoid jumps in the program and improve instruction locality. Measurements taken from a variety of programs showed that not only the number of executed instructions decreased, but also that the total cache work was reduced (except for small caches) despite increases in code size. Adreas Krall [26] presents a kind of code replication techniques to improve the accuracy of semi-static branch prediction scheme. It uses profiling to collect information about the correlation between different branches and about the correlation between the subsequent outcomes of a single branch.

VI. Conclusions And Future Work

As a power technique for the runtime adaption of software among different ISAs, dynamic binary translation has gained much attention. It offers unprecedented flexibility in the control and modification of a program during the runtime. But these attractive features are confined by substantial overhead, especially in the situations that translate among totally different architectures. In this paper, we tried to offset this drawback by introducing a dynamic-static combined approach to optimize the dynamic binary translators. Under this approach, we first employ the emulating execution stage to dynamically collect the profile information and the translated target code. Especially, the path of each execution flow will also be tracked. In the static analysis phase, due to the platitudinous profile information and target code, we employed the method of code replication to build traces, and then reorganized the layout of the target code by putting the hottest trace at the top of software cache. By implementing our approach, the execution flow was concentrated on a small area. Experimental results had shown that our approach reduced more than 34% runtime on average at the expense of 2% memory expansion ratio on average.

In the future, we will employ more algorithms to improve the runtime overhead as well as reduce the memory size expansion ratio.

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, 10ZR1416400, 10DZ1500200, 10511500102), IBM SUR Funding and IBM Research-China JP Funding.

References

[1] Jinpyo Kim, Wei-Chung Hsu, Pen-Chung Yew, “COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications”, International Conference on Parallel Processing (ICPP), 2007, pp: 25-33.

[2] L. Baraz, T. Devor, O. Etzion, et.al, “IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems.” In 36th International Symposium on Microarchitecture, 2003, pp: 191-201.

[3] HP ARIES Dynamic Binary Translator, http://h21007.www2.hp.com.

[4] LUK, C.-K., COHN, R., MUTH, R., et.al, “Pin: building customized program analysis tools with dynamic instrumentation”. In PLDI ’05 (New York, NY, USA, 2005), pp. 190–200.

[5] V. Bala, E. Duesterwald, and S. Banerjia, “Dynamo:

A transparent runtime optimization system". In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00), June 2000.

[6] Nethercote, N., Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In PLDI '07 (New York, NY, USA, 2007), pp. 89–100.

[7] Swaroop Sridhar, Jonathan S. Shapiro, Prashanth P. Bungale, et.al, "HDTrans: An Open Source, Low-Level Dynamic Instrumentation System", In VEE '06: Proceedings of the 2nd international conference on Virtual execution environments (2006), pp. 175-185.

[8] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling", In Proceedings of the 1994 ACM SIGMETRICS Conference on the Measurement and Modeling of Computer Systems (1994), pp. 128–137.

[9] Fabrice Bellard. "QEMU: a Fast and Portable Dynamic Translator", Proceedings of the USENIX Annual Technical Conference, 2005, pp.41-46.

[10] K Ebcioglu, E R Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", 24th Annual International Symposium on Computer Architecture (ISCA'97), pp: 26-37.

[11] J. Lu, H. Chen, P.-C. Yew, et.al, "Design and implementation of a lightweight dynamic optimization system," The Journal of Instruction-Level Parallelism, vol. 6, 2004.

[12] Youfeng Wu, Mauricio Breternitz, Justin Quek, Orna Etzion, Jesse Fang. "The Accuracy of Initial Prediction in Two-Phase Dynamic Binary Translators". Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization. pp:227-238.

[13] A. Chernoff and R. Hookway, "DIGITAL FX!32 - Running 32-Bit x86 Applications on Alpha NT," USENIX Association. Berkeley CA: Proceedings of the USENIX Windows NT Workshop, August 1997.

[14] Chaohao Xu, Jianhui Li, Tao Bao, Yun Wang, "Metadata driven memory optimizations in dynamic binary translator", Proceedings of the 3rd international conference on Virtual execution environments. San Diego, California, USA. Pages: 148 – 157, 2007.

[15] GNU Compiler Collection Internals.

[16] K. Hazelwood and M. D. Smith. "Managing bounded code cache in dynamic binary optimization systems", Transaction on Code Generation and Optimization, 3:263-294, 2006.

[17] K. Hazelwood and J. E. Smith, "Exploring Code Cache Eviction Granularities in Dynamic Optimization Systems", Proceeding of the International Symposium on Code Generation and Optimization, pages 89-99, 2004.

[18] A. Guha, K. Hazelwood and Mary Lou Soffa, "Reducing Exit Stub Memory Consumption in Code Caches", High Performance Embedded Architecture and Compilers, Second International Conference, pages 87-101, 2007.

[19] D. L. Bruening and S. Amarasinghe, "Maintaining Consistency and Bounding Capacity of Software Code Caches", Proceedings of the International Symposium on Code Generation and Optimization, pages74-85, 2005.

[20] D. L. Bruening and V. Kiriansky, "Process-Shared and Persistent Code Caches", Proceedings of the 4th International Conference on Virtual Execution Environment, pages 61-70, 2008.

[21] D. L. Bruening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation", Ph.D thesis, Massachusetts Institute of Technology (2004).

[22] Apala Guba, Kim Hazewood and Mary Lou Soffa, "DBT Path Selection for Holistic Memory Efficiency and Perfomance", VEE'10 March 17-19, 2010, Pittsburgh, Pennsylvania, USA.

[23] David Hiniker, Kim Hazelwood, Michael D. Smith, "Harvard UniversityImproving Region Selection in Dynamic Optimization Systems", Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05).

[24] Duane Merrill, Kim Hazelwood, "Trace Fragment Selection within Method-based JVMs", VEE'08, March 5-7, 2008, Seattle, Washington, USA.

[25] D. Bruening, T. Garnett, S. Amarasinghe. "An infrastructure for adaptive dynamic optimization", In International Symposium on Code Generation and Optimization, pages 265-275, San Francisco, California, 2003.

[26] Andreas Krall, "Improving Semi-static Branch Prediction by Code Replication", SIGPLAN 94-6/94 Orlando, Florida USA.

[27] Yunchao He, Chen Kai, Jinghui Gu, et.al, "A New Approach to Reorganize Code Layout of Software Cache in Dynamic Binary Translator", (PAAP2010). Accepted.

[28] Haipeng Deng, Kai Chen, Bo Liu, et.al, "Efficient Online Trace Building Using Code Replication", GCC 2010, Accepted.

[29] Yindong Yang, Haibing Guan, Erzhou Zhu, Hongbo Yang, Bo Liu, CrossBit: A Multi-Sources and Multi-Targets DBT, CLOUD COMPUTING 2010, November 21-26, 2010 - Lisbon, Portugal, accepted.

[30] Jinghui Gu,Chao Xu,Ling Lin,Juyu Zheng,Kai Chen,Haibing Guan, The Implementation of Static-integrated Optimization Framework for Dynamic Binary Translation, ITCS2009, Kiev, Ukraine, 25-26 July, 2009.

[31] Huihui Shi, Yi Wang, Haibing Guan, Alei Liang,"An Intermediate Level Optimization Framework for DBT", ACM SIGPLAN notice, vol 42(5),2007.5:3~9.

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). 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 student at Shanghai Jiao Tong University, China. He received the M.S. degree and B.S. degree in Computer Science and Technology in Anhui University, Anhui, China, in 2004 and 2008 respectively. His research interests include virtual machine, binary translation and computer architecture.

Kai Chen received his Ph.D. degree in computer science from Shanghai Jiaotnog University (China). He is currently a lecturer with the Faculty of information security engineering, Shanghai Jiaotong University (Shanghai, China). His current research interests include, but are not limited to, computer architecture, network virtualization, virtual machines.

Ruhui Ma is currently a Ph.D student at Shanghai Jiao Tong University, China. His research interests include virtual machine, binary translation and computer architecture.

Yunchao He, master candidate. Got bachelor degree from School of Computer Science, Wuhan University of Science and Technology in 2008. Now I am study in Embedded System Lab, School of Software, Shanghai Jiao Tong University in China. My research area is dynamic binary translation and I/O virtualization on KVM.

Haipeng Deng is currently a Master Degree Candidate student at Shanghai Jiao Tong University, China. His main research interest is binary translation.

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