2013 IEEE International Conference on High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing

VBIW: Optimizing Indirect Branch in Dynamic Binary Translation

Xiaochun Zhang*†‡, Xiang Gao*‡, Qi Guo§, Jing Huang*†‡, Hongwei Liu*†‡, Xiaofu Meng*†‡

*State Key laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
†University of Chinese Academy of Sciences, Beijing, China
‡Loongson Technology Corporation Limited, Beijing, China
§IBM Research - China, Beijing, China
Email: {[1]zhangxiaochun,[4]huangjing,[5]liuhongwei,[6]mengxiaofu}@ict.ac.cn,
[2]gaoxiang@loongson.cn, [3]guoqi@cn.ibm.com

***Abstract*—A major challenge of Dynamic Binary Translation (DBT) is to efficiently handle the indirect branch. Conventionally, to translate indirect branches (IBs), DBT systems need to conduct *address mapping* between Source Binary Blocks (SBB) and Translated Binary Blocks (TBB). However, even a dedicated address mapping process still results in non-trivial performance overheads to DBT. This paper first provides exhaustive analysis of the overheads of address mapping, and finds that *hash lookup*, *context switching* and *consistency maintenance* are three main sources of overheads. To address these overheads, we further propose a novel approach called Virtual Branch Instruction Write-back (VBIW). The key idea is to dynamically write a Virtual Branch Instruction (VBI) into the SBB once the mapping is determined. Since the VBI contains the target TBB address of a branch, the costly address mapping can be eliminated for further reference of the same branch. In addition to theoretical analysis of VBIW, we also implement VBIW on a X86 to MIPS DBT system of Godson-3. The experimental results show that VBIW can reduce DBT execution time by 29.5% on average (ranging from 1.8% to 58.5%) for singlethreaded benchmarks, and by 19.6% on average (4.5% to 62.5%) for multithreaded benchmarks.**

***Keywords*-dynamic binary translation, indirect branch, virtual branch instruction, consistency maintenance, multithreaded emulation**

I. Introduction

A Dynamic Binary Translation (DBT) system is used to translate source binary code to native binary code at runtime. During the last few years, researchers have proposed several popular DBT systems, such as, Pin [1] and Qemu [2] and Valgrind [3], and such systems have been extensively used to enable ISA translation [4], program instrumentation [5], dynamic optimization [6], secure execution [7] and architectural simulation [8].

The most challenging problem of DBT systems is their performance overhead. For example, when running SPEC2006 INT benchmarks under Pin framework, the performance overhead could even up to 300% to the native execution [9]. Besides, in [10], a pure software-based binary translation system, which emulates the X86 architecture on a MIPS processor, brings more than 400% performance overheads to the native execution on X86 architecture.

One of the major sources of such overhead stems from the process of indirect branches [11], [12], [13], [14]. Traditionally, to reduce the translation overhead, DBT system slices the source binary into Source Binary Blocks (SBBs), and translates them into Translated Binary Blocks (TBBs) for reusing. Each block consists of an instruction sequence bounded by branches. For direct branches, the transferring between SBBs and TBBs could be simplified by code chaining techniques [2] since the execution path is determined before the execution. However, for indirect branches, whose branch target can only be determined until run-time, it is necessary to perform non-trivial *address mapping* to transfer the execution from one TBB to another, which causes a significant overhead to DBT.

To accelerate the address mapping of indirect branches, in this paper, we first conduct exhaustively experimental analysis of the overheads of address mapping of indirect branches on a commercial X86 to MIPS DBT system built upon Godson-3, which is a scalable multicore processor with X86 emulation [15]. Our experiments on singlethreaded (SPEC CPU2000/2006) and multithreaded (NPB [16]) benchmarks show that *hash lookup*, *context switch* and *consistency maintenance* are three main sources of the overhead of address mapping. Then, to reduce such overheads, we propose a novel approach called Virtual Branch Instruction Write-back (VBIW). The basic idea of VBIW is to dynamically write a Virtual Branch Instruction (VBI) into the SBB once the mapping is determined. Since the VBI contains the target TBB address of a branch, the costly address mapping process can be eliminated for further reference of the same branch. In addition to theoretical analysis of VBIW, we also implement VBIW mechanism on Godson-3. The experimental results show that VBIW can reduce the DBT execution time by 29.5% on average, ranging from 1.8% to 58.5% for SPEC benchmarks, and by 19.6% on average, ranging from 4.5% to 62.5% for NPB benchmarks.

The rest of this paper processes as follows. Section II presents the detailed experimental analysis of address map-

978-0-7695-5088-6/13 $31.00 © 2013 IEEE
DOI 10.1109/HPCC.and.EUC.2013.206

ping for indirect branches. Section III elaborates the proposed VBIW mechanism. Section IV shows the experimental results of VBIW. Section V compares VBIW with state-of-the-art techniques to optimize translation indirect branches. Finally, section VI concludes this paper.

II. Analysis of Handling Indirect Branches

A. Experimental Setup

Our evaluation is conducted on Godson-3, a 64-bit quad-core 900MHz MIPS processor, with 2G RAM running Fedora OS. Test set consists SPEC CPU2000/2006[1] with test inputs, and NPB compiled in Class S. For the evaluation of singlethreaded benchmarks, the official Qemu 1.4.1 is used. For the evaluation of multithreaded benchmarks, we add a patch to Qemu since the original version fails to support multithreaded programs. The runtime of functions is measured by performance counter register giving the results in CPU Cycles(CC), and the cost of measurement is 6 CC each time which has been eliminated in the final experimental data.

B. Analysis of Indirect Branch Handling

Traditionally, mapping methodologies are based on either data space or instruction space. For the data based methodology [17], pairs of SBB and TBB addresses are saved in data buffers. For the instruction based methodology [18], DBT system executes instruction blocks in which SBB and TBB addresses are loaded by immediate instructions. To emulate the multithreaded programs, consistency maintenance is necessary in both of the aforementioned methodologies, because simultaneous reading, writing and execution may generate mismatched SBB and TBB addresses.

In Qemu, The key process of address mapping is *hash lookup* (HL), which is a typical and traditional data based methodology. In hash lookup, the address of SBB is hashed to a key to select the address of TBB from a hash table. For consistency maintenance, lock mechanism keeps different threads accessing the hash table in sequence. The overall framework of address mapping in Qemu is illustrated in figure 1.

Figure 2 analyzes the execution breakdown of address mapping process in Qemu. The first observation is that the overhead of address mapping varies significantly for different programs. For example, on program *gzip* and *parser*, the overall execution time of address mapping (including HL, context switching and consistency maintenance) is over 60% of the entire performance overhead, while on program *art*, the overhead of address mapping is only less than 5% of the entire overhead.

The second observation is that for most benchmarks, the address mapping is very time-consuming. In detail, the overheads are from three main sources:

[1]www.spec.org

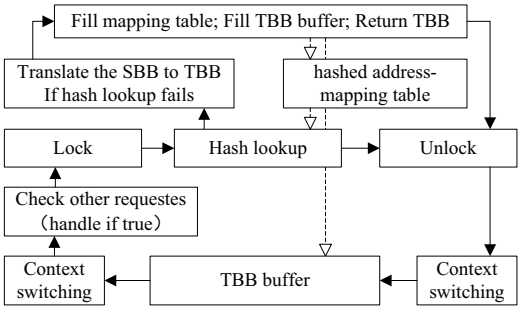

Figure 1. The framework of address mapping in Qemu.

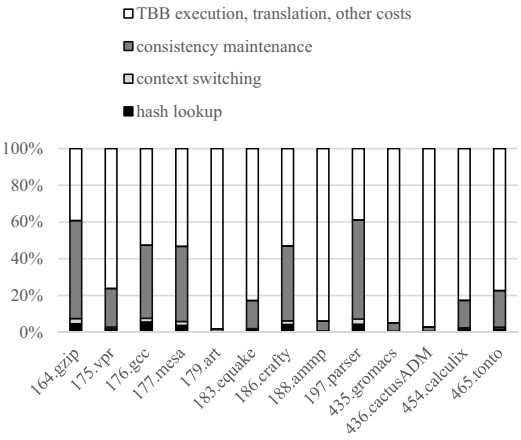

Figure 2. Breakdown analysis of address mapping process.

1) *Hash Lookup*: Figure 3 presents the execution overhead of HL. For all the beanchmarks, the average HL execution time is 29.4 CC. The worst arises in program *cactus*, which is 44.4 CC. The total overhead of HL is on average 69.1% compared with the native execution time, and is up to 147.2% in *crafty* test. The results indicate that the process of HL is a main drawback of DBT efficiency.

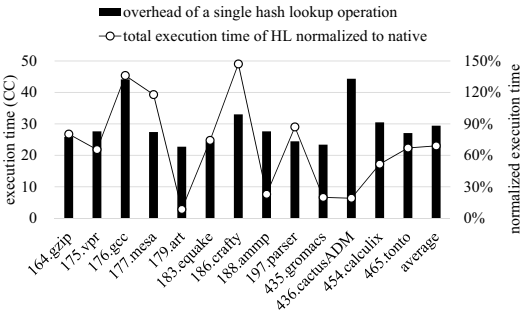

Figure 3. Performance overhead of hash lookup in Qemu.

2) *Context switching*: Before and after HL, execution leaves and re-enters the translated binary, and the registers used by TBB should be stored and recovered.

Qemu for MIPS takes a cautious way that eight registers in the total 32 are saved and reloaded, which follows the arrangement of MIPS compilation.

3) *Consistency maintenance*: For multithreaded emulation, several threads may handle IBs together, but only one could access to hash table because elements in it may be modified according to lookup result. Qemu uses NPTL to do mutex operations, which will incurs significant overheads even when no conflict occurs. Because it is hard to recognize a multithreaded program before execution, the consistency maintenance is necessitated in every address mapping even in singlethreaded emulation.

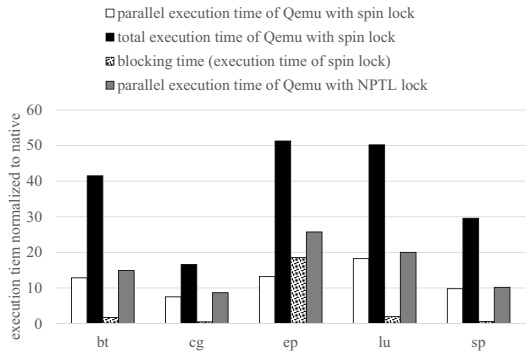

Figure 4. Breakdown analysis of address mapping process in multithreaded tests

Figure 4 presents the breakdown of IB handling in NPB multithreaded tests. The total execution time is the summation of 4 threads running on 4 CPU cores, which is about 4x of the parallel execution time in a highly parallelized program. In multithreaded emulation, when different threads access the hash table simultaneously, other threads must wait for the executing one to finish its HL. The blocking in this case incurs more overhead than in singlethreaded emulation. We analyze the overhead of blocking with spin lock. Results in figure 4 show that, the blocking time is on average 9.8% of the total execution time, and is up to 36.1% in *ep* test which is highly parallelized.

Adversely, spin lock will occupy and waste CPU continuously during blocking. In comparison, consistency maintenance in Qemu is based on NPTL lock mechanism. On one hand, it enables OS to take control and to dispatch CPU to other processes when the blocking occurs. On the other hand, the overhead of program schedule makes multithreaded emulation more inefficient. Figure 4 shows that, with NPTL lock, the parallel execution time of Qemu is on average 1.3 times of the modified Qemu with spin lock, and is up to 1.9 times in *ep* test.

Based on the above observations, an efficient address mapping mechanism should comply with the following guidelines:

- Simplify the consistency maintenance in order to avoid mutex operations.
- Do the address mapping in parallel in multithreaded emulation.
- Simplify the address mapping to make it faster than HL.
- Handle the IBs mainly in TBB to eliminate context switching.

III. VBIW: Virtual Branch Instruction Write-Back

The key idea of VBIW is to provide a light-weight address mapping path instead of hash lookup operations. To achieve this goal, VBIW overwrites SBB's instructions at the IB targets with VBIs during runtime. A VBI contains the target TBB address of the branch, and once it is detected in next executions, DBT jumps directly to the correspondent TBB block without hash lookup. In this section, we detail the proposed VBIW mechanism.

A. Virtual Branch Instruction

VBI is the key of our approach, we first elaborate the design of VBI as follows:

1) Opcode: As VBI will overwrite source binary code, it must be never identical with the source ISA in any form, which means we should pick the reserved opcode in source ISA.

2) Sub-address: If the source ISA is in variable length, it is possible that the VBIs overlap each other and destroy the whole mechanism. To avoid such overlapping, the address to put VBI must be aligned, and a sub-address is necessary to indicate that according to which source instruction is the VBI generated in the aligned memory section.

3) Address field: An address field in VBI records the target TBB's position which is translated from the target SBB of the indirect branch. It must cover all memory space of translated binary buffer.

VBI read/write operation should be atomic to avoid consistency maintenance overhead. On our platform, which executes x86 source code on Godson-3, VBI is 64-bit wide, *opcode* field is 29-bit, and *sub-address* is 3-bit to address each byte in the 64-bit VBI. And *address field* is 32-bit to provide full memory coverage on TBB code space. 64-bit read/store atomic operation is fulfilled by nature on our platform. The detailed design of VBI is shown in Figure 5.

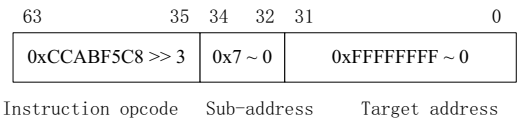

Figure 5. Detailed design of VBI.

B. VBI Operation Procedure

Once an IB instruction is encountered, DBT will fetch data on 64-bit-aligned destination address. If it is a VBI, and the destination address offset matches the VBI's sub-address field, DBT will jump directly to the target TBB based on the address saved in the VBI's address field. Otherwise, which means it is not a VBI or the sub-addresses mismatches, DBT resorts to hash lookup function. If the corresponding mapping from the SBB to the TBB exists in the hash table, a proper VBI will be generated and written to 64-bit-aligned SBB address, or a new translation procedure is invoked and a new address mapping item will be inserted into hash lookup table without any VBI operation. Here, in our implementation, we do not write the VBI immediately after translation, because the translation procedure cannot recognize whether the translated SBB is a IB target.

Under some conditions, more than one VBI may be put into the same position because the 64-bit VBI is longer than source ISA width. In this case, later IB execution may overwrite a previous VBI. To reduce this VBI-write contention, a counter is provided for each hash lookup item. The VBI will be written to the position according to a zero counter, and non-zero counter will prevent the overwrite because a VBI must have been already written. Until the counter reaches a certain threshold, which implies a more accessed branch, the VBI will be overwritten again. Figure 6 shows The workflow of address mapping via VBI.

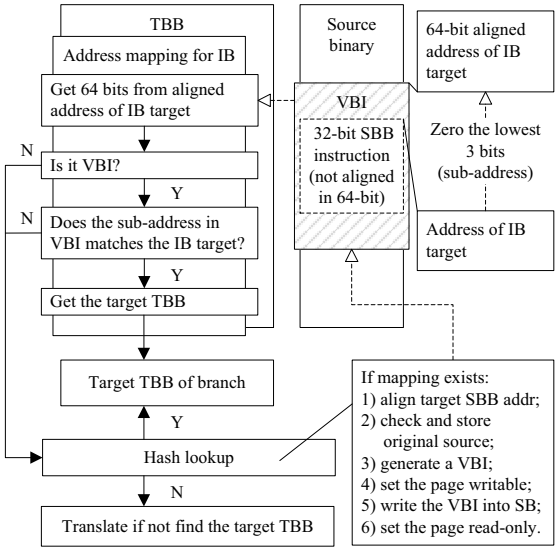

Figure 6. The workflow of VBIW.

C. Translation in VBIW

Because the source code is modified in VBIW, to keep the translation correct, a mechanism is required to store the original source binary and restore the source during translation. Therefore, Original Source Binary Backup Table (OSBBT) is needed. It is organized like this: 1) It makes an entry for every 16K-byte range to benefit the self-modifying handling. 2) In every entry, 16 sub-pages expands to lower the sum of data when recovering source code. 3) Each sub-page contains 8 slots to save the 64-bit source codes and their 32-bit addresses. 4) All the main sub-pages are arranged in an array. 5) Extra sub-pages can be linked to the relative main sub-page, in case that the slots in the main one is not enough. A piece of OSBBT is presented in Figure 7. When writing the VBI, the original source binary will be stored with its position in OSBBT.

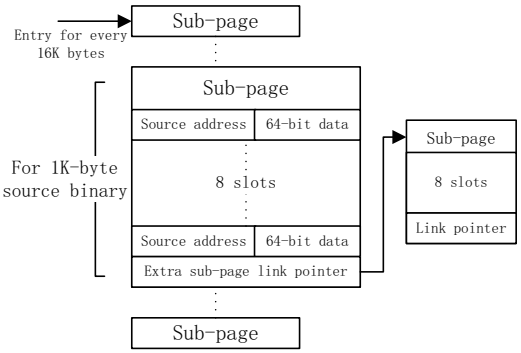

Figure 7. Original Source Binary Backup Table

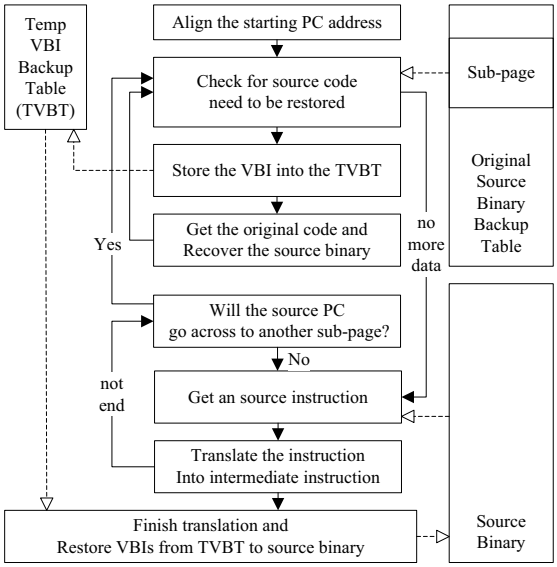

Figure 8. The workflow of translation process.

Figure 8 shows The workflow of translation with source code recover. During translation, all the pages containing the SBB will be set writable and back to read-only after translation. According to the SBB address, sub-pages for this area can be found, and the source code in the translation area will be restored. It is necessary to check if the translation goes across to another sub-page, and more source codes will

be recovered in this case. Whenever restoring the source, the written VBIs will be saved to a buffer temporally, and be put back after translation. Generally, VBIW increases the complexity of translation to reduce the overhead of execution. Considering the execution takes more importance in DBT, we think such trade-off is worthy.

D. Consistency maintenance

The consistency maintenance during address mapping in VBIW is quite simple. Obviously, VBI-read operations can be executed in parallel. In case that reading encounters during writing, there are only two results: 1) the data is the VBI matching the IB, and the TBB will go on executing correctly. 2) The data is another VBI written by another hash lookup, or it is the source binary restored by the translation, then the TBB will get unmatched data, and go to hash lookup. The result is nothing but only an extra hash lookup happens, and the execution continues correctly.

In VBIW, writing VBI is in a lock period of hash lookup and translation. Therefore, there is no need to worry about mis-operations on writing. Mostly, each VBI is written to SBB only once and translation is mainly finished during startup, the conflict in such case incurs minor costs.

E. Limitations

Trading off between generality and performance, a thorough compatibility with arbitrary self-modifying and self-reference is abandoned, because such applications reveal nearly no usage but incur extreme overheads on our platform. However, rectifying is considered as following:

1) Self-modifying: When writing a VBI, if a self-modifying encounters on the same page in a multithreaded application, the self-modifying cannot be recognized in VBI-W because the verification is based on page write-protection scheme. To support this condition, we can keep the page in read-only state, and an exception will be triggered when writing a VBI to SBB or recovering source binary. A extra flag could resolve that whether it is caused by VBI operations or self-modifying, and former process could be finished in the exception handling.

2) Self-reference: In VBIW, self-reference is a great challenge because the source binary is modified with VBIs. If the self-reference is to the IB targets, it cannot be supported by VBIW. We can address this issue inelegantly, by allocating an extra mapping buffer in the same size of source binary code. In this case, the VBI will be written into the buffer and the source binary will be kept original. By adding an offset between the mapping buffer and the source binary buffer, the IB address can be translated to get the VBI. However, the memory cost is not ignorable. Future work must extensively test the applications within the range of our usage to check for self-reference. If self-reference on the IB targets exists, the method mentioned above should be carried out, and further optimization will be needed to lower the memory cost.

IV. Experimental Evaluation

In this section, we evaluate the performance of VBIW for both singlethreaded and multithreaded benchmarks.

For singlethreaded benchmarks, we conduct experiments on several programs selected from the widely-used SPEC CPU2000 and CPU2006. The experimental results are shown in Figure 9. The most notable example is program *parser*, VBIW can improve the performance by about 58.5% compared with original Qemu since this program is IB intensive. For other benchmarks such as *art*, which is not IB intensive, VBIW can still improve the performance by 1.8% compared with original Qemu. On average, VBIW can improve the performance by 29.5% compared with Qemu, which well demonstrates the effectiveness of VBIW for efficient processing of IB. Figure 10 shows that the sum of hash lookup is reduced significantly in VBIW, and Figure 11 presents the execution time of address mapping in both Qemu and VBIW, identifying that our methodology outperforms the Hash Lookup and the consistency maintenance based on lock mechanism.

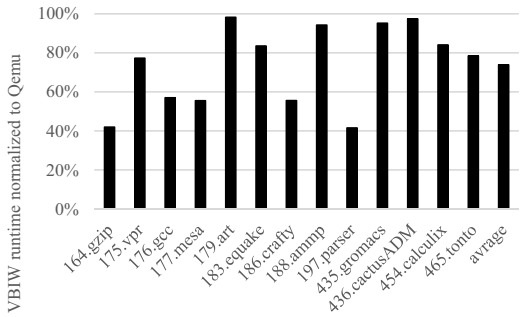

Figure 9. SPEC CPU2000 and CPU2006 benchmarks for VBIW evaluation

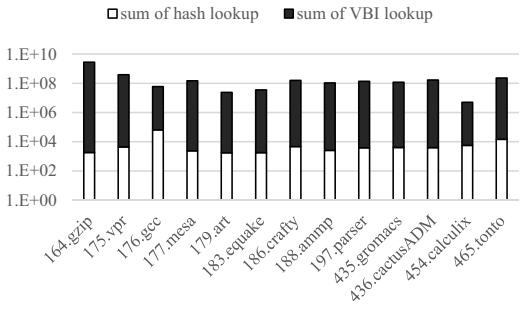

Figure 10. Sum of hash lookup and VBI lookup in VBIW

The NPB multithreaded benchmarks are also employed in VBIW evaluation. Firstly, the benchmarks can as well be executed in singlethreaded mode, and in this mode, the experimental result shown in Figure 12 demonstrates an outcome similar to the evaluation based on SPEC benchmarks.

By comparison, VBIW presents more potency to optimize the multithreaded emulation, which is supported by the result

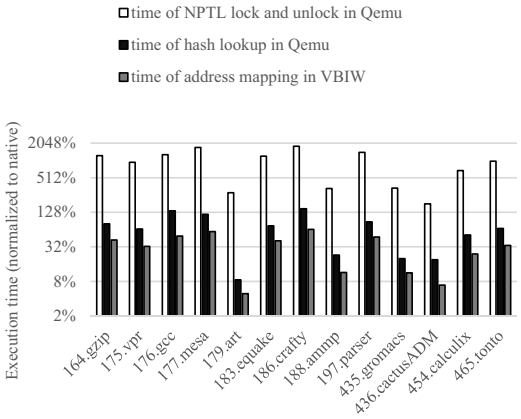

Figure 11. Execution time of the operations in address mapping process

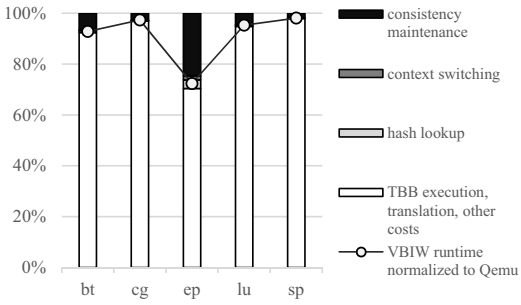

Figure 12. Breakdown analysis and VBIW evaluation on NPB benchmarks in singlethreaded mode

shown in Figure 13. In multithreaded experiment, the NPB benchmarks are configured to run in 4 threads, and all threads are running in parallel on a 4-core Godson-3 processor. With VBIW, every benchmark gains more reduction on performance overheads than in singlethreaded test. For the IB intensive benchmark, *ep*, the improvement gets up to 62.5%, which is only 27.6% in singlethreaded mode. This fact verifies that, VBIW can prevent the conflicts of address mapping between threads, and benefits the IB handling more significantly in multithreaded emulation. Conclusively, VBIW can reduce the execution overhead of multithreaded emulation by 19.6% on average in NPB test.

V. Related work

Researches on DBT system have been kept popular for decades. Among many topics on DBT performance enhancement, IB handling is one of the hottest and has been studied extensively.

Several techniques, such as Sieve [18], IBTC [17], Inline [19], RATS [20], have been proposed to handle and speedup the IB translation. Sieve implements branch chains to redirect IBs. IBTC is based on a hashed address table. Inline method can optimize the branches with stabilized targets, and RATS is focused on return type branches.

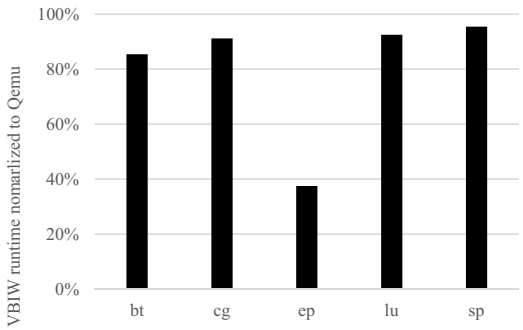

Figure 13. VBIW evaluation on NPB benchmarks in multithreaded mode

These methods emphasize on improving address-translation efficiency, but do not conduct in-depth investigations on the related overheads. In these methods, address-mapping entries are used for multiplex IBs. The contents in the entries will be filled dynamically and even be rearranged to increase the hit rate. Such processes will incur access conflicts in multithreaded emulation, and makes consistency maintenance a major overhead according to our experiments. A state-of-the-art research on IB handling, SPIRE [14], uses a shadow space of the source code filled with branch instructions redirecting IBs to benefit mapping efficiency. However, potential conflicts also exist in multithreaded emulation in case that an IB jumps to the shadow space where is being filled by another thread simultaneously. The cost in such case depends on the conflict handling scheme. Moreover, SPIRE puts strict limitation on the guest and native ISAs because of a deficiency of handing unaligned instructions on aligned architectures. By comparison, VBIW solves the instruction alignment and consistency maintenance problems by a well designed VBI and atomic memory access.

There are also hardware methods to optimize IB handling. For example, CAM [10] can extremely speedup the address translation, and the consistency is promised by hardware. However, the restriction of hardware cost tampers its malleability in pervasive DBT systems. Moreover, the modest size of hardware mapping table restricts its hit rate in some programs, and the overhead of conventional software methodology will still be dominant in such cases. On these observations, pure software optimization such as VBIW is still indispensable.

VI. Conclusion

For IB handling in DBT, the overheads are incurred not only by the address mapping, but also by the operations on the path to the mapping function. The major source is the context switching and consistency maintenance. In VBIW, address translation is faster than hash lookup and is finished directly in TBB, and the consistency maintenance mechanism is simple and costless. Therefore, the VBIW handles the IBs efficiently. Future work will be focused

on the adaptability, and we have to ensure whether it is necessary to adapt VBIW to all kinds of self-modifying and self-reference applications.

ACKNOWLEDGMENT

This work is partially supported by the National Sci&Tech Major Project (No.2009ZX01028-002-003, 2009ZX01029-001-003, 2010ZX01036-001-002, 2012ZX01029-001-002-002), National Natural Science Foundation (No.60921002, 61003064, 61050002, 61070025, 61100163, 61133004, 61173001, 61222204, 61232009) of China, and the National High Technology Development 863 Program of China (2012AA010901, 2012AA011002, 2012AA012202, 2013AA014301).

REFERENCES

[1] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '05, 2005, pp. 190–200.

[2] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATC '05, 2005, pp. 41–41.

[3] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI '07, 2007, pp. 89–100.

[4] S. Bansal and A. Aiken, "Binary translation using peephole superoptimizers," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08, 2008, pp. 177–192.

[5] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, ser. VEE '12, 2012, pp. 133–144.

[6] D. Pavlou, E. Gibert, F. Latorre, and A. Gonzalez, "Ddgacc: boosting dynamic ddg-based binary optimizations through specialized hardware support," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, ser. VEE '12, 2012, pp. 159–168.

[7] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, J. C. Knight, A. Nguyen-Tuong, and J. Rowanhill, "Secure and practical defense against code-injection attacks using software dynamic translation," in *Proceedings of the 2nd international conference on Virtual execution environments*, ser. VEE '06, 2006, pp. 2–12.

[8] A. Patel, F. Afram, S. Chen, and K. Ghose, "MARSSx86: A Full System Simulator for x86 CPUs," in *Design Automation Conference 2011 (DAC'11)*, 2011.

[9] B. Dhanasekaran and K. Hazelwood, "Improving indirect branch translation in dynamic binary translators," *Proceedings of the ASPLOS Workshop on Runtime Environments, Systems, Layering, and Virtualized Environments*, pp. 11–18, 2011.

[10] W. Hu, Q. Liu, J. Wang, S. Cai, M. Su, and X. Li, "Efficient binary translation system with low hardware cost," in *Proceedings of the 2009 IEEE international conference on Computer design*, ser. ICCD'09, 2009, pp. 305–312.

[11] D. Bruening, T. Garnett, and S. Amarasinghe, "An infrastructure for adaptive dynamic optimization," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, ser. CGO '03, 2003, pp. 265–275.

[12] J. D. Hiser, D. Williams, W. Hu, J. W. Davidson, J. Mars, and B. R. Childers, "Evaluating indirect branch handling mechanisms in software dynamic translation systems," in *Proceedings of the International Symposium on Code Generation and Optimization*, ser. CGO '07, 2007, pp. 61–73.

[13] E. Borin and Y. Wu, "Characterization of dbt overhead," in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*, ser. IISWC '09, 2009, pp. 178–187.

[14] N. Jia, C. Yang, J. Wang, D. Tong, and K. Wang, "Spire: improving dynamic binary translation through spc-indexed indirect branch redirecting," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '13, 2013, pp. 1–12.

[15] W. Hu, J. Wang, X. Gao, Y. Chen, Q. Liu, and G. Li, "Godson-3: A scalable multicore risc processor with x86 emulation," *IEEE Micro*, vol. 29, no. 2, pp. 17–29, 2009.

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks - summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, ser. SC '91, 1991, pp. 158–165.

[17] K. Scott and J. Davidson, "Strata: A software dynamic translation infrastructure," in *IEEE Workshop on Binary Translation*, 2001.

[18] S. Sridhar, J. S. Shapiro, E. Northup, and P. P. Bungale, "Hdtrans: an open source, low-level dynamic instrumentation system," in *Proceedings of the 2nd international conference on Virtual execution environments*, ser. VEE '06, 2006, pp. 175–185.

[19] K. Ebcioglu and E. R. Altman, "Daisy: Dynamic compilation for 100% architectural compatibility," in *24th Annual International Symposium on Computer Architecture, Conference Proceedings*, 1997, pp. 26–37.

[20] K. Hazelwood and A. Klauser, "A dynamic binary instrumentation engine for the arm architecture," in *Proceedings of the International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06, 2006, pp. 261–270.