CrossBit: A Multi-Sources and Multi-Targets DBT

Yindong Yang, Haibing Guan, Erzhou Zhu, Hongbo Yang, Bo Liu
School of Electronic, Information and Electrical Engineering
Shanghai Jiao Tong University
Shanghai, China
{*yasaka, hbguan, ezzhu, yanghongbo819, boliu*}*@sjtu.edu.cn*

***Abstract*—Dynamic binary translator (DBT) is typically used for software migration or binary code optimization. In this paper, we describe the design and implementation of a multi-sources and multi-targets DBT–CrossBit, which aims at fast migrating existing executable source code from one platform to another alien target platform with lower cost. In order to support code translation among multi-sources and multi-targets better, a new intermediate instruction set–VInst, 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 set 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 unit of translation. Additionally, to address the performance issue, we adopt several generic optimization methods to optimize the translated code. Finally, our experimental result indicates that, for the SPECint2000 benchmarks, CrossBit's performance is pleasant and can meet the design requirement.**

Keywords-DBT; intermediate instruction; CrossBit; basic block;

I. INTRODUCTION

Cloud computing, a relatively romantic term, builds on decades of research in virtualization, distributed computing, grid computing, utility computing, and so on. For cloud computing becomes wildly popular and has the potential to transform a large part of the IT industry, developers with innovative ideas inevitably take it into account. As a new evolution of on-demand information technology services and products, cloud computing has become popular until IBM and Google announced a collaboration in this domain in October 2007 [19]. Thus, there are still lots of obstacles to the growth of cloud computing. One of them is performance unpredictability, which caused by the use of virtual machines (VM). Cloud computing moves data and computing away from desktop and portable PCs into large data centers. It involves applications based on different instruction set architectures (ISA), as well as different hardware platforms. VMs have proved to be ideally suitable for the needs of the heterogeneous computing. Not surprisingly, VMs are likely to be common at multiple levels of the data center or server farm.

Virtualization is a core technology for enabling cloud resource sharing. To a large extent, cloud computing will be based on virtualized resources. In the recent 30 years, the computing machinery technology, both hardware and software, has been developing at a tremendous pace. On one hand, the processing speed of processor gets faster, and accordingly there needs less time to perform each certain unit task. On the other hand, the framework of the processor (e.g., *x*86, MIPS, PowerPC) also becomes multiplex in order to satisfy various requirements and be applicable in different scenarios. Different processors usually base on different ISAs. This leads a problem, one kind of processor can only support one appointed or specific operating system (OS) and applications without virtualization technology. However, due to historical reasons, some of these legacy processor architectures (like *x*86) fail to comply with classical virtualization criteria or hardware support. Though it is quite a challenge to migrate existing OSes or applications running on one platform to another platform, binary translation overcomes the obstacles and successfully improves migrating efficiency.

Binary translation technology proposes a transparent and inexpensive approach to migrate applications or OSes compiled for one processor to another. Binary translation can be implemented in either static or dynamic way, more technical detail can be found in [17]. Binary translation techniques are still in infancy compared with their compiler counterparts, plus many binary translators have been proved that they were handcrafted from scratch. For example, commercial binary translators are always closely bound to the underlying machine, or cannot generate code for more than one source and target machine pair, and hence it is difficult to reuse. To solve this problem, this paper proposes a new DBT–CrossBit, which takes "multi-sources" and "multi-targets" as its design goal. With the help of CrossBit, the applications compiled for a specific processor can run on different processors easily, without bringing too much overhead.

In the academic and commercial fields, many binary translators have achieved some success. In terms of improving the performance of applications, the original HP Dynamo system [2] is a dynamic software optimization system that is capable of transparently improving the performance of a source instruction stream as it executes on the native processor. Microsoft dynamic software optimization system Mojo [3], is capable of handling a wide range of programs, including multi-threaded applications that make use of exception handling. In the aspect of migrating applications, Digital FX!32 provides fast transparent execution of 32-bit *x*86 applications on Alpha systems running Windows NT [1]. IA-32 Execution

 ISBN: 978-1-61208-106-9

layer is designed to support IA-32 applications on Itanium-based systems [16]. The UQBT (University of Queensland Binary Translator) [4], is a reusable, component-based binary-translation framework which lets engineers quickly and inexpensively migrate existing software from one processor to the other. Other translators are designed to save power, like Transmeta's Code Morphing technology which is used for unmodified Intel IA-32 binaries to run on the low-power VLIW Crusoe processor [5]. More discussions on extensive prior work can be found in Altman et al. [6], [7].

Different from binary translators mentioned above, CrossBit doesn't translate source code into specific target code directly. Instead, CrossBit dynamically translates one BB of source code into VInst specifications and then into a BB of target code each time. A BB refers to a block of contiguous instructions, starting at the first instructions of the executable file or the instruction executed immediately after the end of last block, and ending with a branch or jump instruction. In particular, it is to be noted that using intermediate instructions bring many benefits. First, it becomes easier to add other source ends and target ends. Second, the workload and complexity of development can be reduced. For example, to translate n kinds of different source ISAs to n kinds of different target ISAs, theoretically, the complexity can be reduced from $O(n*n)$ to $O(n)$, as depicted in Figure 1. Third, intermediate instruction blocks can offer better opportunities for optimization and thus make optimization easier.

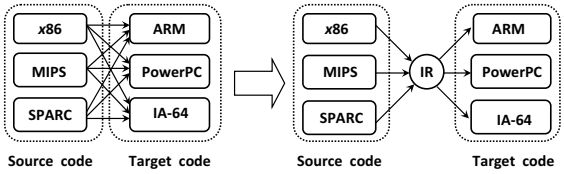


Figure 1. traditional translation model vs. CrossBit's translation model

In this paper, our aim is to to develop a multi-sources and multi-targets DBT with reasonable performance. Specifically, main technical contributions are as follows:

- We design a new DBT–CrossBit, which translates binary code based on different ISAs to other ISAs conveniently;
- We design a new intermediate instruction set–VInst for CrossBit, which involves no specific machine instructions;
- We give a comprehensive experimental evaluation of CrossBit for translating MIPS to *x*86, and MIPS to POWER.

The rest of this paper is organized as follows. Section II describes the framework of CrossBit and how it works in detail. Section III focuses on the design of intermediate instructions. Section IV discusses some optimization tactics adopted by CrossBit. Section V discusses the main issues relevant to our approach to DBT. Section VI presents the preliminary performance of CrossBit, using SPECint2000 as our benchmarks. Section VII gives the summary of this paper and the future research on DBT.

II. The Framework of CrossBit

Dynamic binary translation is an attractive technology for running legacy applications and OSes on the platforms that the software is not originally compiled for. However, the dynamic binary translation technology itself is very complex and difficult to implement. For one thing, developing a complete application-level translator from scratch always takes a lot of manpower and material resources, let alone the development of a system-level translator [18]. Moreover, if one wants to run binary code of one popular platform (like *x*86) on a less popular platform (still in use currently), e.g., SPARC, PowerPC, and MIPS, he has to develop a translator for each of these platforms. In contrast, it is appreciated if there is a translator that can add source ends or target ends easily without repeating the development work. CrossBit is such a translator, and the design goal of CrossBit is retargetable and extensible. We hope that CrossBit will be a promising and reliable basic research platform for studying application-level DBT in the future.

We divide the framework of CrossBit roughly into three parts:

- *Front end*: loads binary executable code into memory and transforms the source binary code into VInst specifications;
- *Intermediate layer*: forms the VInst specifications to blocks and realizes optimization;
- *Back end*: transforms the intermediate blocks to target instructions and executes them immediately.

Figure 2 shows a high-level view of CrossBit. The rectangular boxes related to front end and back end, while the dotted boxes belong to the intermediate layer. We first give a brief overview of all the main components, and then explain how they interact with each other.

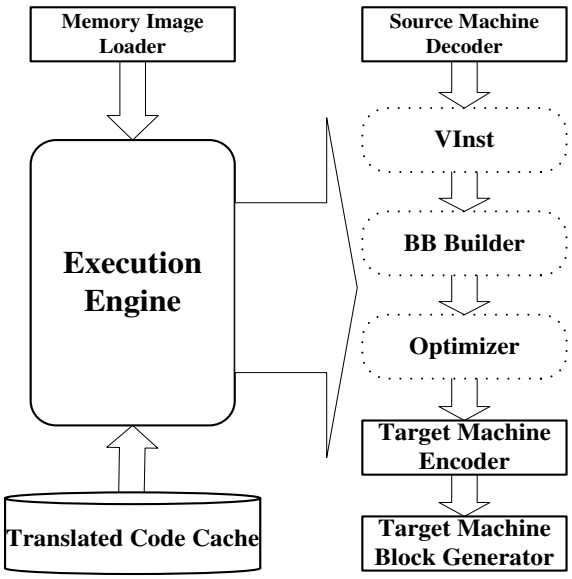


Figure 2. The framework of CrossBit

The names and functions of these components are given as follows:

- *Memory Image Loader*: loads the source binary code into the memory of the target platform, and maps the guest application address space to the host virtual address space;
- *Source Machine Decoder*: analyzes and decodes the source binary code, and converts it to VInst specifications;
- *BB Builder*: constructs VInst specification blocks;
- *Optimizer*: optimizes VInst specification blocks;
- *Target Machine Encoder*: encodes intermediate instruction blocks to target machine instructions;
- *Target Machine Block Generator*: manages SPC (Source Program Counter) and TPC (Target Program Counter) in a hash table and updates their relationship when necessary, so as to speed up the lookup process of target blocks;
- *Translated Code Cache*: caches the translated code so as to save the time of retranslation;
- *Execution Engine*: the commander of CrossBit, in charge of scheduling all components.

The workflow of the entire CrossBit system is as follow:

1) Get the SPC of the first instruction in the next BB, and check whether its corresponding TPC exists in hash table;
2) If the TPC exists, indicating that BB has been already translated and cached, jumps to that TPC and executes that BB directly; Otherwise, turn to 3);
3) *Source Machine Decoder* decodes the following source instructions until the last BB is met, where the decoded instructions will be the input to *BB Builder*;
4) *BB Builder* constructs intermediate instruction blocks;
5) *Optimizer* conducts optimization on the intermediate instruction blocks, e.g., elimination of redundant load instructions, BB linking and so on;
6) *Target Machine Encode* encodes intermediate instructions to target machine instructions which are cached in *Translated Code Cache*, and then *Target Machine Block Generator* updates hash table.

Execute Engine executes 1)∼6) repeatedly until the end of the program. CrossBit takes one BB at a time, keeps doing the job of "translate-execute" cycle, as shown in Figure 3.

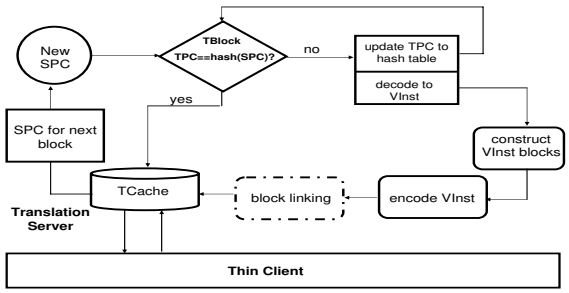


Figure 3. The workflow of CrossBit

III. INTERMEDIATE INSTRUCTION SET

The challenge of designing an intermediate instruction set is how to provide enough high-level run time information about program behavior, as well as be appropriate as an architecture interface for all external applications, libraries, and operating systems. There are many virtual machines using intermediate instructions as a transition layer, such as JVM and Microsoft's Common Language Infrastructure (CLI). However JVM and CLI have some limitations. Both of them use complex, high-level operations with large runtime libraries, and they are tailored for object-oriented languages with a particular inheritance model and their complex runtime systems require significant OS support in practice [15].

An intermediate instruction set can include rich program information for optimization, and can be independent of most implementation–specific design choices, but it is not suitable for a certain hardware implementation. Take LLVA [15] for example, LLVA is a good intermediate instruction set for C++ high level language, but LLVA not suitable for CrossBit to handle low level binary code. Of course, we know that it's impossible to design a universal intermediate instruction set for all conceivable architecture designs. Instead of designing VInst (Virtual Instruction) as the intermediate instruction set that is suitable for DBT to handle low level binary code. VInst must be designed to be regular and simple, for the reason that source instructions and target instructions are all low level machine instructions, and the performance of CrossBit is sensitive to the cost of translation. In a sense, VInst is a kind of low-level ISA. It is similar to RISC ISA and has main characteristics as follows:

- unlimited numbers of 32-bit virtual registers, marked by V_n, wherein the value of V_0 always returns zero;
- "*Load-Store*" style, that is only 'load' or 'store' instruction can access the memory;
- base plus displacement is the only addressing mode;
- instructions of VInst only exist in memory;
- every instruction of VInst is orthogonal, in other words, each instruction can't be replaced by other VInst instructions.

The design of VInst affects the quality of generated target code, and therefor affects the efficiency of CrossBit. When designing VInst, we seek the balance between the performance and the cost. On one hand, VInst is kept as simple as possible so as to reduce the cost of translation. On the other hand, the sematic of VInst should be rich enough to support various characteristics of different ISAs. Thus, by studying the design of some popular ISAs, we have picked up 27 most commonly used instructions to compose VInst. The translation effort is in combing VInst instructions into more complex specific machine instructions.

VInst comprises six kinds of basic instructions which

 ISBN: 978-1-61208-106-9

are compatible with most popular ISAs. They include arithmetic/logical, control transfer, data transfer, memory access, register state mapping and special instructions. Every kind of instructions is shown in Table I.

Table I
THE MOST COMMONLY USED IRS

Type	Instruction Name
register state mapping	GET PUT
memory access	LD ST
data transfer	MOV LI
arithmetic/logic	ADD SUB AND NOT XOR OR MUL MULU DIV DIVU SLL SRL SRA CMP SEXT ZEXT
control transfer	JMP BRANCH
special	HALT SYSCALL CALL

According to the original version of CrossBit we developed, CrossBit translates binary code compiled for PISA (Portable Instruction Set Architecture) into *x*86. That is, CrossBit translates binary PISA code into *x*86 instructions. CrossBit reads the memory at the address indicated by SPC to produce intermediate representation objects, where each intermediate representation object represents one source instruction, not binary representation actually. The translation from a source instruction to VInst takes three steps:

- via GET instruction, mapping source machine registers which the PISA instruction requires to read to virtual registers;
- use one or more VInst instructions to implement the function of each PISA instruction;
- via PUT instruction, mapping the result from virtual registers to source machine registers.

So far, we have used GCC compiler to generate binary executable file. A simple “hello world” C++ source program leads almost 20 thousands of VInst instructions, the details are omitted herein for brevity it’s impractical to present a whole example here. A key issue of translation is semantic matching. Instruction swelling is inevitable because the translation policy we adopt here is to be correct first then better. Thus, there needs later optimization to offset the cost.

IV. OPTIMIZATION

Binary translation always serves as an alternative way to execute legacy software. The performance of the translated code should be competitive with the legacy architecture’s performance. However, overhead is inevitably lost during the process of translation, because the legacy software has the luxury of being produced using an optimizing compilation. In lack of high-level language code, binary translators cannot perform available optimizations compared to compilers. In this case, optimization is particularly important to dynamic binary translation. One common approach to improving binary translation performance is profile-guided optimization. Profiling is a process for dynamically collecting program information (instruction and data statistics) that is used to guide optimization during the translation process. As a DBT, CrossBit can do some optimization at the run-time according to the profile information. The profile information that CrossBit collects is the number of executing times of each BB, which can be used to find out hotspots. A hotspost is a region of contiguous code which is frequently executed. In CrossBit, a hotspot is taken as a super block, which consists of numbers of basic blocks. Because the optimization process itself is time-consuming and potentially performance degrading, it should be applied to the hot code (e.g., superblocks) instead of the cold code.

With the profile information, i.e., the number of executing times of each BB, a BB is determined to be hot if its number of times reaches a threshold, like 2000, as in the case of dynamo [13]. Once a BB becomes hot and it is not part of some superblocks, a new superblock can be constructed beginning from that BB. If the BB ends with a branch instruction, the next BB is chosen to add to the superblock according to their numbers of executing times, reversing the branch condition if necessary, like Figure 4 depicts [9]. Otherwise, it is easy to find out the next BB. This process repeats until the termination condition is met, e.g., the last instruction of the BB is an indirect jump. When the next encountered BB belongs to other superblock, the numbers of BBs reach the maximal value.

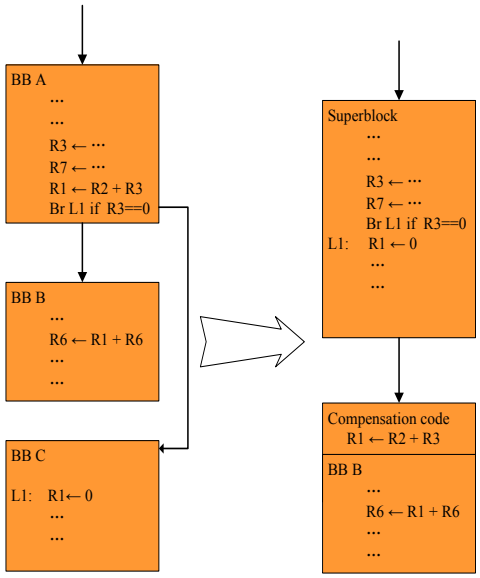


Figure 4. Build superblock based on profiling.

Another optimization tactic we adopt is modifying the direct jump instructions and indirect jump instructions. As direct jump instructions are executed frequently (nearly one in seven instructions is the direct jump), the Execution Engine has to look up into the hash table to find out the next BB after the execution of each direct jump, which leads to a bottleneck in the performance of CrossBit. A simple solution to this problem is to modify the instructions to jump directly to the next basic block, for there is only one target address for each jump. Instead of transferring the control to the Execution Engine, the modi-

 ISBN: 978-1-61208-106-9

fied instruction simply jumps to the translated instructions corresponding to the source machine instructions. This mechanism is called BB chaining and avoids significant overhead associated with returning to the Execution Engine after every instruction executed.

One of the major sources of overhead in CrossBit stems from handling indirect jumps. Experimental evidence shows that the effect of methods that handle indirect jumps depends on the features of the target architecture such as addressing modes, branch predictors, cache sizes and the ability to preserve architecture state efficiently. To tackle indirect jumps in CrossBit efficiently, we take two methods. One uses hash table to keep the target address of an indirect jump. The other mechanism is taken when some indirect jumps have only a few exits for most of the time. It uses indirect jump inlining, and the TPC of the next basic block can be obtained by comparing the jump target address to the inlined target one.

V. Other Issues

In this section, we discuss the other main issues relevant to our approach to DBT.

A. Self-Modifying Code

Self-modifying code which refers to those programs modify parts of their source code during the translation processes, though it is uncommon. When this happens, translated target code stored in the code cache would no longer correspond to the modified source code. The mechanism we adopt to handle this problem in the CrossBit is setting the original source code region write-protected. That is, any attempt to write a page in the protection area will be trapped and the delivery of a signal to the CrossBit. This can be done via a system call made by the *Execution Engine*. Consequently, the translated code will be discarded and the CrossBit invokes retranslation.

B. Address Space

Address space management is an very important aspect of DBT. In the development of CrossBit, we don't allow specific code to be placed on different regions of the memory space, like Figure 5 depicts. Because CrossBit is a process VM, it views memory as a logical address space supported by the target machine's OS. Where regions of the source memory address space could map onto regions of the target memory address space.

VI. Performance

In this section, we present preliminary results of CrossBit. Of course, no single benchmark characterizes the performance of a system, we adopt the most common method of testing, and that is running the SPECint2000 test benchmarks. Rather than giving the performance results of all the front ends and back ends we have developed, we choose two typical pairs, MIPS–*x*86 and *x*86–Power, and make an evaluation of them. The experiment of MIPS–*x*86 was taken on Intel® Pentium® 4 CPU 2.0GHz with

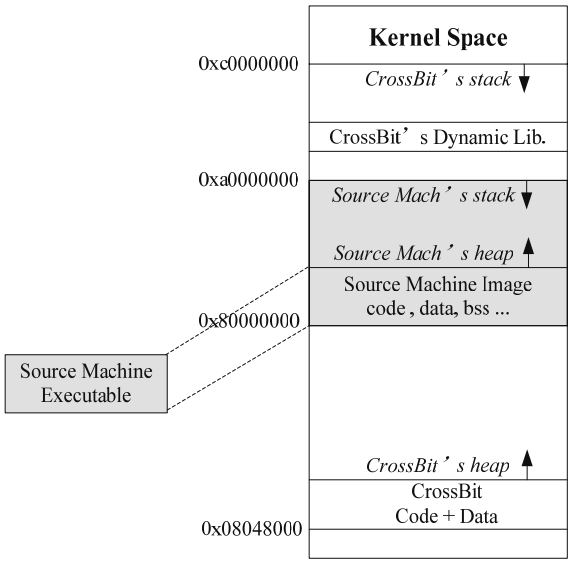
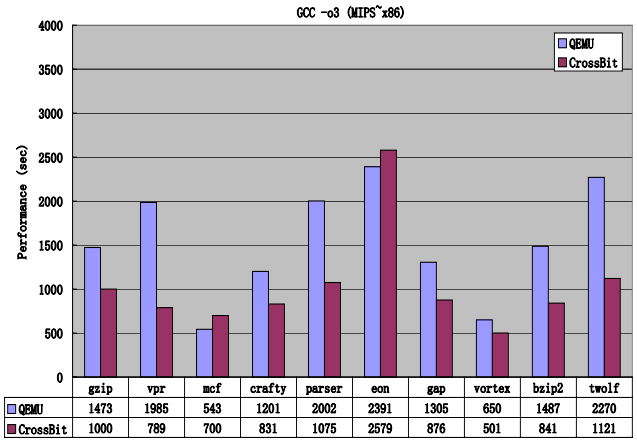


Figure 5. CrossBit runtime address space layout.

1.5GB PC3700 DDR SDRAM Memory, while the experiment of *x*86–POWER was performed on POWER® 6 1-core 4.2GHz CPU with 2 x 512 DRAM 667MHz Memory. The bottom of the charts is the executing time (sec) that every benchmark consumes to finish the testing.

Figure 6 and Figure 7 give the test results of SPECint2000 benchmarks for CrossBit translating MIPS-*x*86, and *x*86-POWER respectively. The rest of the benchmarks failed to run successfully which might be due to the lack of complete support for all Linux system calls. We still deal with these issues now. Meanwhile, in order to evaluate the performance of CrossBit, we have chosen QEMU as a reference for comparison. QEMU [12] is a multi-sources and multi-targets DBT and also using intermediate instructions. QEMU got its reputation for multi-function, not performance. QEMU itself didn't take much optimization measures.



	gzip	vpr	mcf	crafty	parser	eon	gap	vortex	bzip2	twolf
QEMU	1473	1985	543	1201	2002	2391	1305	650	1487	2270
CrossBit	1000	789	700	831	1075	2579	876	501	841	1121

Figure 6. Performance comparison of the CrossBit with QEMU. The bars represent the time (sec) consumed by them from translating MIPS-*x*86 (shorter is better).

As the figures shown in Figure 6 and Figure 7, we can

 ISBN: 978-1-61208-106-9

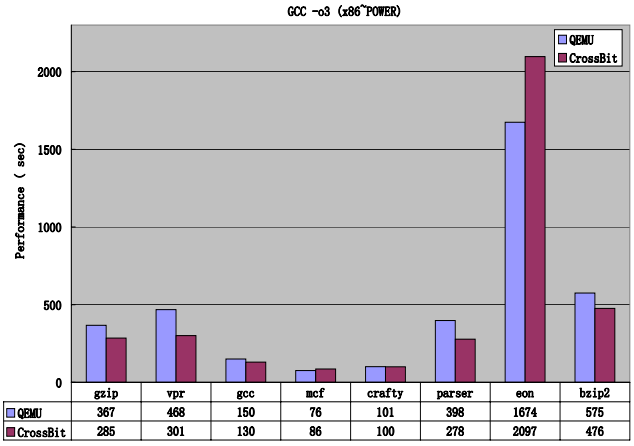


Figure 7. Performance comparison of the CrossBit with QEMU. The bars represent the time (sec) consumed by them from translating *x*86-POWER (shorter is better).

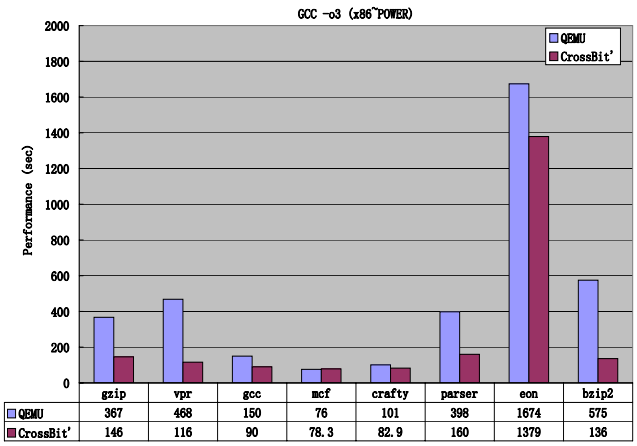


Figure 9. Performance comparison of the CrossBit after optimization (called CrossBit') with QEMU. The bars represent the time (sec) consumed by them from translating *x*86-POWER (shorter is better).

see that performance of CrossBit consistently outperforms than QEMU for nearly all these benchmarks. When compared with the QEMU, CrossBit performs better, since QEMU is not famous for its performance. We continue to optimization issue. Poor performance is due to code swelling. Take an example where CrossBit translates *x*86 ISA to POWER ISA, *x*86 is CISC ISA, with variable-length instructions. During the translation, from source *x*86 instructions to VInst, and from VInst to target POWER instructions, the code swelling is very big, even dozens of times. After the optimization to VInst blocks is performanced, a striking result shows that the performance of optimized code efficiency is more than 1 time faster than the natively translated code (as seen in Figure 8 and Figure 9). As CrossBit has introduced an intermediate layer to support multi-sources and multi-targets, this result is acceptable.

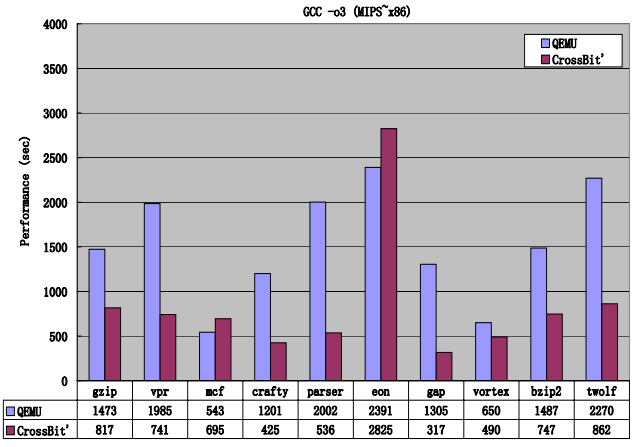


Figure 8. Performance comparison of the CrossBit after optimization (called CrossBit') with QEMU. The bars represent the time (sec) consumed by them from translating MIPS-*x*86 (shorter is better).

Meanwhile, we study the relationship between performance improvement and profiling overhead in Figure 10 and Figure 11. Profiling could enhance the performance of CrossBit, and also brings the extra overhead to the system.

The final optimization result is obtained by the updating performance minus the overhead. Sometimes we should balance the depth of optimization and the performance of the system. From these figures, we can see that the profiling process only increases part of overheads for the CrossBit, but it does improve the quality of translated code and the execution efficiency of the whole system.

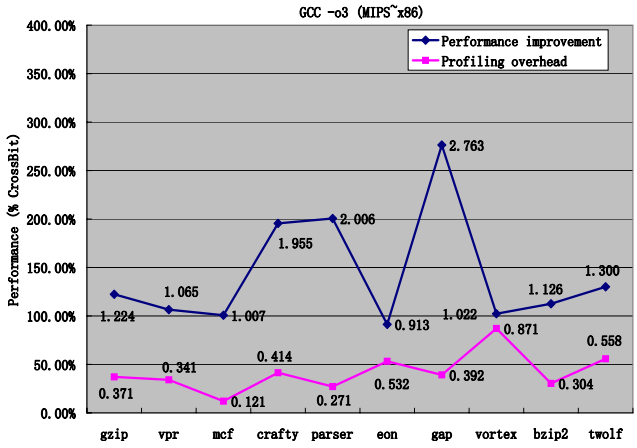


Figure 10. The relationship between performance improvement and profiling overhead of CrossBit compared with the natively translated code from MIPS-*x*86.

VII. Conclusion and Future Work

Our initial primary goal is to build a DBT platform that runs the same applications on different architectures with reasonable performance. The foundation of application migration is that instruction set should be translated correctly first. So far, we have implemented several front ends and back ends. The front ends included MIPS, *x*86 and SPARC while the back ends included *x*86, POWER and SPARC. Right now, we are focusing on improving the performance of CrossBit, so as to rival the other popular binary translators.

Our system is yet still in the absence of exceptions (traps and interrupts), and some of benchmarks fail to run

 ISBN: 978-1-61208-106-9

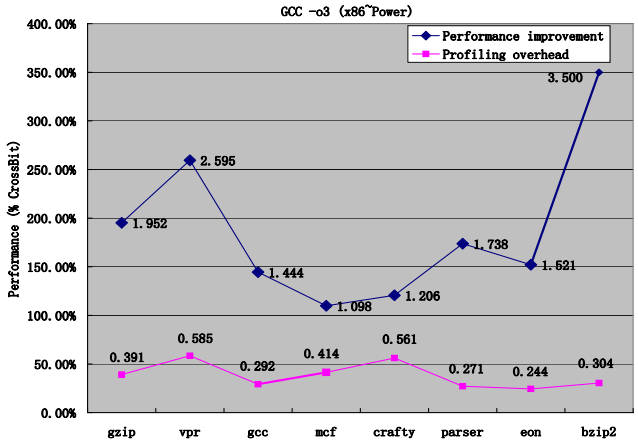


Figure 11. The relationship between performance improvement and profiling overhead of CrossBit compared with the natively translated code from *x*86-POWER.

correctly. In the future, we wish these problems would be solved.

VIII. ACKNOWLEDGMENTS

We would like to thank Yue Xu and Wei Fan for numerous discussions and for their helpful comments on how to improve the paper. Our research is supported by the National Natural Science Foundation of China (Grant No.60773093, 60970107, 60970108), the Science and Technology Commission of Shanghai Municipality (09510701600).

REFERENCES

[1] Anton Chernoff and Ray Hookway, "Running 32-Bit *x*86 Applications on Alpha NT," Proc. USENIX Windows NT Workshop, Usenix Association, Seattle, Washington, August 1997, pp. 17-23.

[2] Bala Vasanth, Duesterwald Evelyn, and Banerjia Sanjeev, "Transparent dynamic optimization: The design and implementation of Dynamo," Hewlett-Packard Laboratories Technical Report HPL-1999-78, June 1999.

[3] Wen Ke Chen, Sorin Lerner, Ronnie Chaiken, and David Gillies, "Mojo: A dynamic optimization system," ACM Workshop on Feedback-Directed and Dynamic Optimization, December 2000, pp. 81-90.

[4] Cifuentes Cifuentes and Van Emmerik Mike, "UQBT: Adaptable binary translation at low cost," Computer, vol.33, March 2000, pp. 60-66.

[5] James Dehnert, Brian Grant, John Banning, Richard Johnson, Thomas Kistler, Alexander Klaiber, and Jim Mattson, "The Transmeta Code MorphingTM Software: using speculation, recovery, and adaptive retranslation to address real-life challenges," International Symposium on Code Generation and Optimization, March 2003, pp. 15-24.

[6] Erik Altman, David Kaeli, and Yaron Sheffer, "Welcome to the Opportunities of Binary Translation," IEEE Computer, vol. 33, March 2000, pp. 40-45.

[7] Kemal Ebcioglu, Erik Altman, Michael Gschwind and Sumedh Sathaye, "Dynamic Binary Translation and Optimization," IEEE Trans, on Computers, vol. 50, June 2001, pp. 529-548.

[8] Yuncheng Bao, Haibing Guan, Jun Li and Alei Liang, "Mobilizing Native machine Code via Dynamic Binary Translation," Proceedings of the 3rd International Workshop on Software Development Methodologies for Distributed Systems, Shanghai, China, 2006, pp. 73-78.

[9] James Smith and Ravi Nair, Virtual Machines: Versatile Platforms for Systems and Processes, Morgan Kaufmann, 2005.

[10] Nicholas Nethercote and Julian Seward, "Valgrind: A program supervision framework," Electronic Notes in Theoretical Computer Science, vol.89(2), 2003, pp. 89-100.

[11] Raymond Hookway and Mark Herdeg, "Digital fx!32: combining emulation and binary translation," Digital Tech.J, vol.9(1), 1997, pp. 3-12.

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

[13] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia, "Dynamo: A Transparent Dynamic Optimization System," Conf. on Programming Language Design and Implementation (PLDI), June 2000, pp. 1-12.

[14] Kevin Scott and Jack Davidson, "Strata: A software dynamic translation infrastructure," Technical Report, UMI Order Number: CS-2001-17, University of Virginia.

[15] Vikram Adve, Chris Lattner, Michael Brukman, Anand Shukla, and Brian Gaeke, "LLVA: A Low-level Virtual Instruction Set Architecture," Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36), San Diego, California, December 2003, pp. 201-216.

[16] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang, and Yigal Zemach, "IA-32 execution layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems," Proceedings of the 36th annual ACM/IEEE international symposium on Microarchitecture (MICRO-36), San Diego, California, December 2003, pp. 191-201.

[17] Mark Probest, "Dynamic binary translation," http://www.complang.tuwien.ac.at/schani/, May, 2009.

[18] Keith Adams and Ole Agesen, "A Comparison of Software and Hardware Techniques for *x*86 Virtualization," Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, October 21-25, 2006, pp. 2-13.

[19] IBM, "Google and IBM Announced University Initiative to Address Internet-Scale Computing Challenges," October 8, 2007, http://www-03.ibm.com/press/us/en/pressrelease/22414.wss.

 ISBN: 978-1-61208-106-9