Efficient Binary Translation System with Low Hardware Cost

Weiwu Hu[1], Qi Liu[1,2], Jian Wang[1], Songsong Cai[1,2], Menghao Su[1,2] and Xiaoyu Li[1,2]
[1]*Key Laboratory of Computer System and Architecture*
Institute of Computing Technology, Chinese Academy of Sciences, Beijing, China
[2]*Graduate University of Chinese Academy of Sciences*
{*hww, liuqi, jw, caisongsong, sumenghao, lixy*}*@ict.ac.cn*

***Abstract*— Binary translation is one of the most important approaches for system migration. However, software binary translation systems often suffer from the inefficiency and traditional hardware-software co-designed virtual machines require the unavoidable re-design of the processor architecture. This paper presents a novel hardware-software co-designed method to accelerate the binary translation on an existing architecture. The hardware supports for source-architecture-only functions, partial decodes and binary translation system acceleration are proposed. These hardware supports help the binary translation system to achieve high performance and simplify the design of the binary translation software. In the meantime, the hardware cost is well controlled in a certain low level. These supports are implemented in Godson-3 processors to speedup the x86 binary translation to the native MIPS instruction set. Performance evaluations on RTL simulation and FPGA emulation platforms show that the proposed method can speedup most benchmark programs by nearly 10 times compared to pure software-based binary translation and achieves about 70% performance of the native program execution. The chip is fabricated in ST 65nm CMOS technology, and the physical design results show that the chip area cost is less than 5%.**

I. INTRODUCTION

Binary translation [1][2] is one of the most important methods for the binary level compatibility. To accelerate the binary translation, lots of software-based optimization methods [3][4][5][6] are proposed. However, the software-based binary translator suffers from its complexity and inefficiency. Traditional hardware-software co-designed virtual machines [7][8] generally aim to performance, power efficiency and so on, meanwhile, keep the compatibility for requirement. But to accomplish those goals, the whole system including the processor micro-architecture needs to be designed from top to bottom. That will cause great design risks and verification costs.

This paper presents a novel hardware-software co-designed methodology on an existing superscalar RISC microprocessor to achieve high performance binary translation. Our method is based on the knowledge that the difference between source and target architecture in process-level (ABI) is not so great as system-level (ISA) and the ABI gap can be bridged by moderate hardware supports. With the extension of process-level binary translation supporting, the binary translation can be achieved with much more efficiency and the design and implementation of the software are simplified.

In this paper, key factors to the quality of generated target codes and the binary translator efficiency are identified and the corresponding hardware support is designed and implemented. Based on the quantitative profile and statistics, it is found that some great semantic gaps between the source and target architecture cause the poor quality of generated target codes. To solve this problem, some target architecture hardware supports for source-architecture-only functions and partial decode unit are implemented. Besides, control transfer in translated code cache and context switch are identified as main contributors to the performance loss of binary translation system. Corresponding hardware supports including address mapping CAM, register saving and restoring acceleration, etc. are also implemented.

To verify the thoughts, the XBAR (X86 Binary translation Acceleration on RISC processors) system is designed and implemented based on Godson [9][10], a MIPS64 compatible processor. The hardware support for binary translation in Godson processors makes the translation from x86 binary to MIPS binary smooth, and significantly improves performance with little silicon area overhead and design and verification cost. Performance evaluations with the Godson-3 RTL simulation/FPGA emulation platform show that these proposed methods can speedup the benchmark programs by nearly 10 times compared to pure software-based binary translation and the system achieves about 70% native program execution performance. The Godson-3 chip is fabricated in ST 65nm CMOS technology, and the physical design results show that the chip area cost is less than 5%. In the meanwhile, the design and implementation complexity of the software part of the binary translator is reduced with these suitable hardware supports.

This paper makes the following contributions:

- A novel hardware-software co-designed method to accelerate the binary translation on an existing ISA is proposed. The method is suitable to be applied in the binary translation of any source and target architecture.
- Several key factors to the performance of binary translation, most of which widely exist in the binary translation area, are found. Corresponding low-cost hardware supports are proposed and implemented.
- The method is applied in real chips. The efficiency is proven by experiments and analyzed.

The following sections are organized as follows. Section II cites the related work. Section III gives the preliminary analysis of key factors to the performance of binary trans-

978-1-4244-5028-2/09/$25.00 ©2009 IEEE

lation system on superscalar RISC processors. Section IV introduces hardware supports and software improvements for the x86 binary translation based on an existing MIPS architecture, the Godson architecture. Section V presents the prototype implementation and preliminary performance evaluation. Conclusion and future work are given in Section VI.

II. Related Work

Emulation is the most common method to achieve the binary level compatibility. Since the compatibility is the primary objective, the performance goal is typically moderate. Hardware supports for better performance could be hardly found on these systems [3][11][12][13]. To get acceptable performance, lots of pure software-based optimization methods are developed. For example, Intel IA-32 EL [3] speculatively assumes that the TOS remains constant for all entrances to the same block, and that no stack exception occurs to solve the x86 FP stack emulation performance problem. Many systems [14][4][5] implement a form of software-based jump target prediction to save the table lookup for the translation and execution of indirect jump. Lazy evaluation [15][6] and dataflow analysis [3] are widely used by software binary translators to eliminate redundant x86 EFLAGS updates. Those methods greatly improve the software binary translation performance in many aspects. However, several problems exist in the software-based optimization methods. First, most of these methods solve the performance problem from expediency, not from principle as the hardware-based methods. They can not achieve excellent performance as the hardware does. Second, these methods (often based on the assumption, speculation or prediction mechanism) will cause great performance loss in some applications [16]. Besides, these methods greatly increase the design and implementation complexity of the software binary translator. So most of these systems need to be built with lots of time and great efforts and are hard to be maintained and debugged.

Currently most of hardware-software co-designed binary translation systems [16] have different objectives from the compatibility. The compatibility is a requirement but not the objective. They are usually designed for performance, power efficiency etc. To accomplish those goals, the whole system including the micro-architecture is custom-designed from top to bottom. So great costs will be paid on the hardware design and verification. Transmeta Crusoe [8][17][18] and IBM DAISY/BOA [7][4][19] are such systems.

III. Preliminary Experiments And Analysis

In binary translation system, the overall execution time consists of the translated binary code execution time, the binary translation system overhead and the cost of context switch between translation and execution [20]. To analyze the cost of binary translation, a pure software-based binary translation system is built. The translator emulates the x86 architecture and runs on MIPS hardware. SPEC CPU2000 is compiled to x86 binaries with GCC -O3 and used as the benchmarks with the *ref* input files. More detailed information about the experiment environment could be found in Section V. The breakdown of the overall execution time of this software binary translator is illustrated in Fig. 1. The execution time includes the translated binary code execution time and the binary translation system cost. The context switch overhead is distributed in the overall execution time and hard to profile. The figure shows that most of SPEC CPU2000 [21] FP applications spend more than 90% time on the translated binary code execution, and the SPEC CPU2000 INT applications spend significant time (many of them take nearly 20%) on the binary translation system cost. The statistical data shows that both of the two parts are key factors to the overall system performance and need to be optimized.

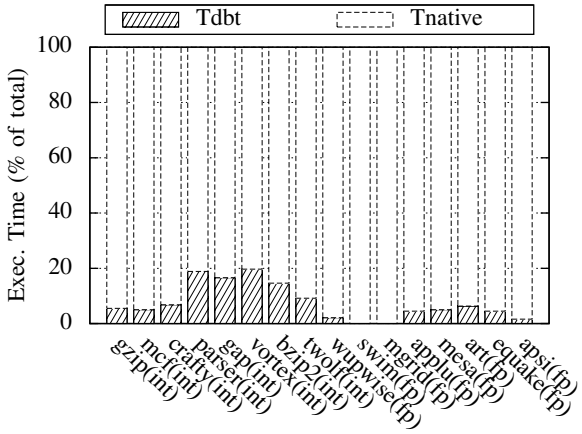

Fig. 1. Breakdown of Binary Translation Execution Time

First, it is obvious that the translated binary code execution time is closely related to the quality of generated target codes. To identify which kinds of instructions are key factors, all the x86 instructions are classified [20] according to the software binary translation method. The most frequently executed instructions which need to be translated with complicated software translation methods are recognized as the key factors to the performance of translated binary code execution.

Take the x86 floating point instructions translation as an example, the x87 FPU (floating point unit) of x86 architecture operates in a very different way from the RISC processors. The x86 FP instructions refer to eight 80-bit registers in an FP stack. The address of the floating point registers is relative to the register on the TOS (top of the stack). And the TOS is stored in the 3-bit TOP field in the x87 FPU status word. However, the MIPS processors just use a flat register file of 32 64-bit registers as floating point registers. Besides, the x87 has a 16-bit tag word to indicate the status of the 8 registers in the x87 FP stack (one 2-bit tag per register). The tag codes indicate whether a register contains a valid number, zero, or a special floating-point number, or whether it is empty. The x87 FPU uses the tag values to detect stack overflow and underflow conditions. This mechanism does not exist in MIPS processors either. As shown in Table I, when using a software-based translator, two simple x86 floating point instructions will be expanded to 22

MIPS instructions due to the FP stack and tags emulation.

TABLE I
X86 FP TRANSLATION EXAMPLE

1) FLDL (%eax) 2) FADDL 0xfffffff0(%eax)	1) LW T0, OFF_TOP(Rbase) 2) ADDIU T0, T0, -1 3) ANDI T0, T0, 0x7 4) SLL T1, T0, 0x3 5) ADDU T1, T1, Rbase 6) LDL T2, 7(Reax) 7) LDR T2, 0(Reax) 8) DMTC1 T2, $FT0 9) ADDU T2, T0, Rbase 10) SDC1 $FT0, OFF_ST0(T1) 11) SB $0, OFF_TAG(T2) 12) SW T0, OFF_TOP(Rbase) 13) ADDIU T0, Reax, 0xfff0 14) LDL T1, 7(T0) 15) LDR T1, 0(T0) 16) DMTC1 T1, $FT0 17) LW T0, OFF_TOP(Rbase) 18) SLL T0, T0, 0x3 19) ADDU T0, T0, Rbase 20) LDC1 $FT1, OFF_ST0(T0) 21) ADD.D $FT1, $FT1, $FT0 22) SDC1 $FT1, OFF_ST0(T0)

The two x86 instructions in the example are parts of the kernel loop of SPEC CPU2000 172.mgrid and will be executed frequently. This program performs poor on the binary translation system. And the result shows that it only achieves 11.4% compared to the native execution speed. These kinds of instructions take large parts in x86 floating point applications and are hard to be translated with effective software-based methods.

Besides, several other kinds of x86 instructions including EFLAGS related instructions, MMX instructions, strings instructions and bytes operation instructions are also identified as key factors to the performance with the same method.

Second, the binary translation system overhead includes the source to target code translation cost (translation cost), translated code control transfer cost (control transfer cost) and system call and exception cost (system service cost) for the process-level binary translation system. One of core concepts of binary translation system [16] is that a code block is translated once and it will be executed for a great lot of times later for most application programs, so the translation cost should be minimal. The profiling data of the software-based binary translation system also confirms that. The system service also costs little. However, in the pure software-based binary translation system there is an obvious performance cost [22][23] in the steady state because the dispatch table for the source PC to target address is accessed every time when there is a branch or jump from one translated block to the next. Typically, this would require several instructions. Fortunately, the translation and execution of direct branches can be optimized by the software direct block chaining technique [15] effectively. The translation and execution of indirect branch instructions are much more difficult to be optimized because the branch target register only includes the address of x86 branch targets. According to the address of the x86 branch targets, the binary translator has to look up the corresponding translated MIPS target address dynamically by some kinds of mapping mechanisms such as hash table [15], and that will cost tens of MIPS instructions. And the statistics in Fig. 2 show that 12.21% of branch target looking up is caused by indirect branch instructions when running SPEC CPU2006 programs. Since branch instructions are so intensive, the binary translator will encounter the indirect branch instruction within less than 100 x86 instructions [20]. So the inefficient translation and execution of indirect branches will cause significant performance loss.

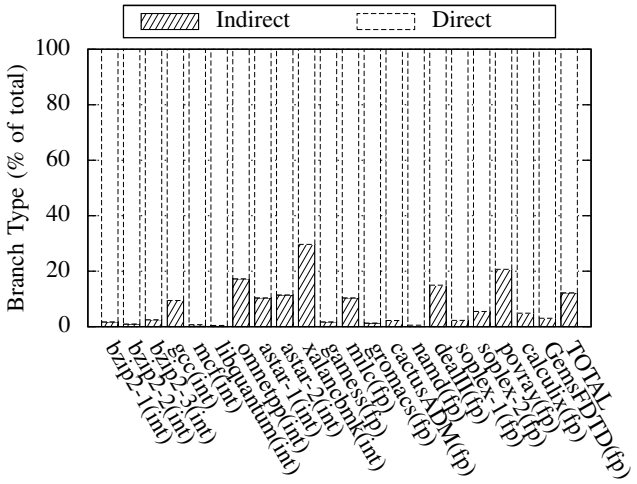

Fig. 2. SPEC CPU2006 X86 Binary Branch Types

IV. HARDWARE SUPPORT FOR BINARY TRANSLATION

As stated in the previous section, several kinds of source-architecture instructions including the x86 floating point instructions, EFLAGS related instructions, etc. as well as the control transfers are identified as the main contributors to the overall binary translation overhead. For the x86 floating point problem, the TOS and x87 tag hardware emulation mechanism is proposed and implemented to accelerate the x86 floating point emulation. And a partial x86 decode unit can be used to handle the rest of the complicated instructions emulation overhead. For the control transfer problem, a hardware CAM is used to accelerate the indirect branch target address lookup. Besides, the hardware optimization for the context switch is also introduced to accelerate the emulation. This section will detail the design and implementation of these hardware supports.

A. X86 Floating Point Unit Support

For the stack emulation problem, the Godson processor maintains a TOS value in hardware dynamically to solve this problem. The TOS value is added to the floating point register number in the decode stage [9]. The new register number is then used as the logical register number to look up the physical register number in the register renaming stage. Some instructions are also defined to increase or decrease the TOS value. The mechanism of restoring the old TOP value is implemented in Godson when branch misprediction or exception occurs. There are hardware flags in

the MIPS floating point registers to indicate whether the register number of the floating point instruction is relative to TOP or not. Only MIPS instructions translated from x86 instructions are affected by the TOP pointer, while normal MIPS instructions are not affected by the TOS value. With the TOS value modification and x86/MIPS FP mode switch instructions, the software binary translator can maintain the FP stack in a very simple way and with little cost. For the tag emulation problem, the Godson processor provides dedicate instructions to simulate the x87 tag with general purpose registers and defines a new exception to reflect stack overflow or underflow exception.

With the hardware support for the x86 floating point emulation, the software binary translator only needs to deal with some corner cases. These situations are hard to be handled by hardware supports or need to be handled with expensive hardware cost. With this hardware support, the software binary translator is also simplified and can achieve very good performance.

B. Partial X86 Decode Unit

Although the x86 architecture is a kind of CISC which is very different from the RISC in instructions, they also share certain similarity with each other. Lots of MIPS instructions and function units share the same functions as the x86 architecture in principle. However, they have small differences in formats, functions and so on. With software-based binary translation, the format transformation, corner case processing etc. will cause the great performance loss. To solve this problem, a partial x86 decode unit based on the existing MIPS decode unit is implemented. This unit will decode instructions which are closer to corresponding x86 instructions in formats and/or semantics. With this method, little modification needs to be made on the function units. The logic design work of the new x86 decode unit is also easy to be done and verified. And fortunately most of the performance limit factors which were identified earlier could be eliminated by this method. For example, the EFLAGS instructions, MMX instructions, strings related instructions and bytes operation instructions can be optimized by this hardware method. Furthermore, the partial decode unit mechanism could be used in the binary translation of any source and target architecture.

Take the x86 EFLAGS as an example, Godson defines the "EFLAGS counterpart" instruction for each fix-point arithmetic instruction by adding a "SETFLAG" prefix to the original instruction. For example, adding a "SETFLAG" prefix to the "SUB R1, R2, R3" instruction turns the SUB instruction to its "EFLAGS counterpart" which does the same calculation as the original SUB instruction but generates x86 EFLAGS instead of the difference of R2 and R3. Though the "SETFLAG" prefix does not consume issue slots, it consumes instruction fetch slots as well as instruction cache spaces. To further increase the execution efficiency, Godson directly defines the "EFLAGS counterpart" instructions for most frequently used instructions. For example, a new instruction X86ADD is defined in Godson to generate EFLAGS of the addition operation. The advantage of the "EFLAGS counterpart" mechanism also lays on the simplicity of its implementation. The mechanism can reuse most data paths of the original instructions, such as the register renaming logic, the reorder logic, the issue logic, the write back logic, etc. Only the decode logic and execution unit need to be minor adjusted.

C. Control Transfers Optimization

To optimize the control transfers caused by indirect branch instructions, Godson implements a hardware-software co-designed method to accelerate the mapping process. A CAM (content associated memory) [24][22] which caches the mapping relationship between the source and target addresses is implemented in the processor hardware. The CAM can be managed and accessed by the software. The structure of the CAM is shown in Fig. 3.

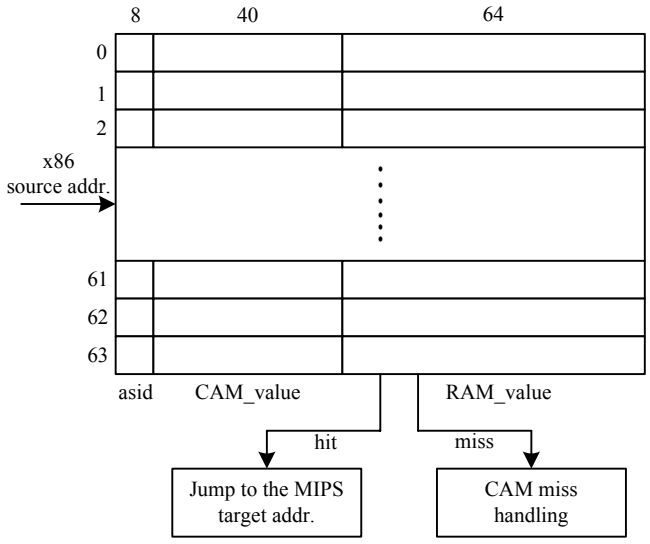

Fig. 3. Hardware-based Target Addresses Lookup Table

All fields of the CAM entry can be written by the CAMWI instruction, and the RAM_value part of each entry can be read by the RAMRI instruction. The CAMPI and CAMPV instruction probe the CAM. In the case of the CAM hitting, CAM generates the index and value of the hit entry respectively. When the software binary translator comes across an indirect branch, the x86 address is fed to the CAM to do the full associative search. If hit, the control transfers to the corresponding MIPS target address. If not, the CAM miss handling subroutine is called.

To get the suitable item number of CAM, the CAM miss ratios with different CAM sizes are evaluated. The experiments are performed on a fast Godson simulator and the SPEC CPU2006 programs with the *ref* input files are used as the benchmarks. The experiment results are shown in Fig. 4.

The CAM with 64 items is implemented in the Godson processor. The CAM with such size reaches a very good hit ratio and is the result of tradeoff between the performance

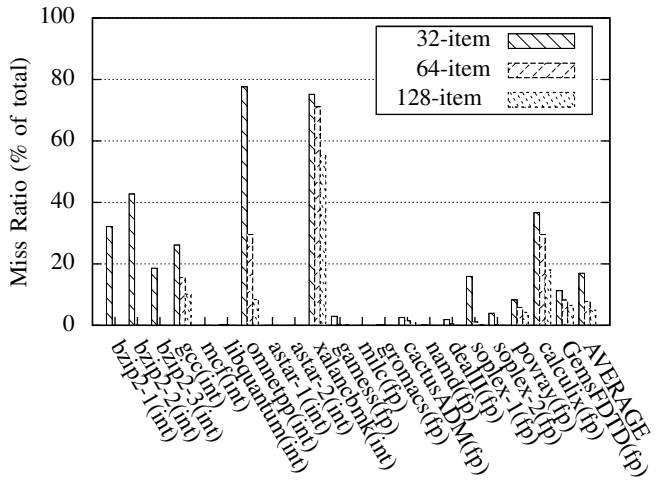

Fig. 4. Miss Ratio with Different CAM Sizes

and the physical design and cost. This co-designed method, which optimizes the direct branch with software direct block chaining method and optimizes the indirect branch by hardware and software, significantly reduces the control transfer overhead to a considerable extent.

D. Context Switch Optimization

The overhead of context switch between the binary translator and translated codes also contributes to the performance loss greatly. The instruction cache flushing and register contents saving and restoring are identified as most costly ones. First, the binary translator dynamically generates the translated binary codes on the target machine. Those binary codes which are generated during the runtime are the data results of the binary translation. Because the binary translator stores these codes in data cache, the execution requires flushing them from the data cache and loading them into the instruction cache. However, flushing the data cache through software to keep coherence between the data and instruction caches is time consuming. Therefore, Godson keeps coherence between the data and instruction caches, as well as the L2 cache, through hardware automatically.

Second, generally the binary translator maps the registers of the source machine to the registers on the target machine with a certain method. When the binary translator performs the context switch between binary translator codes and translated codes, the target registers which are mapped as the source registers should be reserved or restored before switching. To reduce the register reserving and restoring cost, the Godson processor implements 128-bit load and store instructions, with which up to four x86 registers can be saved or restored at one time.

V. Experiment Platform And Performance Results

This section describes the infrastructure and the benchmarks used for the experiments. Experimental results and the analysis are also presented.

A. Experiment Platform

Godson-3 processors [10] are used in the experiments. Godson-3 is the third generation of the Godson microprocessor series, a project of the Institute of Computing Technology of the Chinese Academy of Sciences. Godson-3's scalable and distributed on-chip network connects processor cores and globally addressed level-two (L2) cache modules. Each core of Godson-3 is a MIPS64 compatible, 4-issue, out-of-order processor core [9].

The hardware supports for binary translation which are mentioned above have been integrated into the Godson-3 multi-core processor. The first four-core Godson-3 design is fabricated in ST 65nm CMOS technology and has been taped out. The physical design result shows that the chip area cost is less than 5%. Before the Godson-3 chips return from the fabrication, the performance experiments of the Godson XBAR system are performed on two platforms: a register-transfer-level (RTL) simulation platform and a field-programmable-gate-array (FPGA) prototyping platform. In the RTL simulation environment, the processor core clock frequency is set to 1 GHz, the DDR2/DDR3 clock frequency to 333 MHz, and the HyperTransport clock frequency to 800 MHz. To speed up the simulation, Cadence's Xtreme-3 simulation accelerator [25], which can achieve a speed of 200,000 to 400,000 cycles per second, is used. Because of the difficulty of building a full-scale Godson-3 FPGA prototype system, a partial-scale prototype is built to evaluate the single processor core's performance. The prototype system includes one processor core, one 1 MB L2 cache, one DDR2/DDR3 controller, and one HyperTransport controller. The FPGA prototyping speed is 50 MHz, which is much faster than RTL simulation. Because the run speed ratio between the FPGA prototype's core and I/O clocks differs from those in the real system, the I/O latency of the FPGA prototyping system is carefully adjusted to obtain accurate performance results.

The software architecture of Godson binary translation system is shown in Fig. 5 (in the next page). The software part of XBAR performs staged binary translation [16], using both simple binary translation and optimized translation. During the initial run of an x86 binary program, it performs simple translation and the execution profile information is collected. Then the translator can use the profile information to find the hot regions. Once the execution count of code blocks reaches the certain optimization threshold, the hot region is optimized by some simple and common compiler techniques, such as dead code elimination [16], address alignment fix and so on. Some common problems, such as self modifying code [15], exception support are also handled in the software part of XBAR.

B. Benchmarks

Table II (in the next page) shows the benchmarks which are evaluated on the Xtreme-3 and FPGA platforms. 11 typical application kernels or full applications are chosen to evaluate the hardware improvements and the software

TABLE II
BENCHMARKS

Name	Source	Language	Experiment Platform
Floating point IDCT	Microbench	C	Xtreme
Floating point FFT	Microbench	C	Xtreme
General Control	Microbench	C	Xtreme
Fixed point IDCT	EEMBC	x86 assembly	FPGA
Fixed point FFT	EEMBC	x86 assembly	FPGA
OS booting codes	Microbench	C and x86 assembly	Xtreme/FPGA
401.bzip (train)	SPEC CPU2006	C	FPGA
403.gcc (train)	SPEC CPU2006	C	FPGA
429.mcf (train)	SPEC CPU2006	C	FPGA
450.soplex (train)	SPEC CPU2006	C++	FPGA
454.calculix (train)	SPEC CPU2006	Fortran and C	FPGA

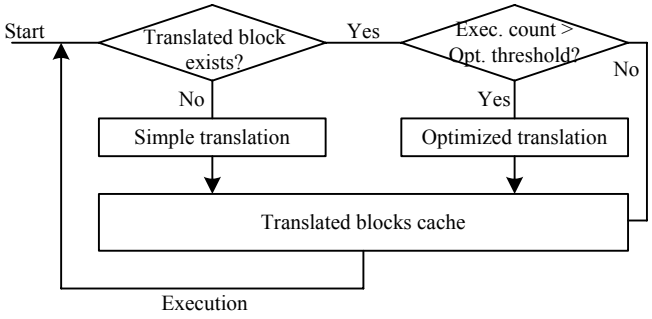

Fig. 5. Software Architecture of Binary Translator

translation efficiency. All benchmarks are compiled with the GCC -O3 flag.

All these benchmarks are executed in three modes: 1. Native MIPS mode, in which benchmarks are directly compiled into MIPS binaries and run on MIPS hardware; 2. Basic translator mode, in which benchmarks are compiled into x86 binaries and run on the standard MIPS hardware and the software-based binary translator. The binary translator is described in the previous section and pure software-based. There isn't any hardware binary translation support in the MIPS hardware either; and 3. Improved translator mode, in which benchmarks are compiled into x86 binaries and run on MIPS hardware using the improved binary translator with partial or full x86 binary translation acceleration methods on RISC processors (XBAR) hardware support.

C. Results and Analysis

Fig. 6 (in the next page) shows the relative performances of the pure software-based translator and different hardware improved translators, which are compared to the native MIPS mode. Floating point IDCT and FFT are handwritten microbenches to verify the common floating point translation and execution performance. As expected the hardware support for x86 floating point brings most of the speedup to these programs. General control program uses lots of EFLAGS related instructions. And fixed point IDCT and FFT contain lots of x86 MMX instructions. They benefit the most from the partial decode support for x86 architecture. The operating system booting codes mix lots of different kinds of instructions and most of the translated codes are used only once. And the support for context switch still brings it relative much speedup. The last five benchmarks are from the SPEC CPU2006 [21]. For 401.bzip, 403.gcc and 450.soplex, as illustrated in Fig. 4 the CAM can catch most of the indirect branch address translations, so the control transfer hardware support improves the performance of the two benchmarks greatly. The x86 floating point support also improves the performance of 450.soplex and 454.calculix greatly, because they are floating point applications. Because of the complexity of these SPEC CPU2006 applications, the context switch and partial decode hardware support also bring some benefits to them.

The overall relative performance of basic and improved translator modes compared to native MIPS mode is shown in Fig. 7. The hardware supports for Godson-3 binary translation significantly accelerate the binary translation from x86 to MIPS, and the binary translator achieves 73.5% of the native execution performance on average. The results show that several key factors to the performance of binary translation architecture are successfully identified. And the proposed hardware-software co-designed methods are very effective to improve the translation and execution performance.

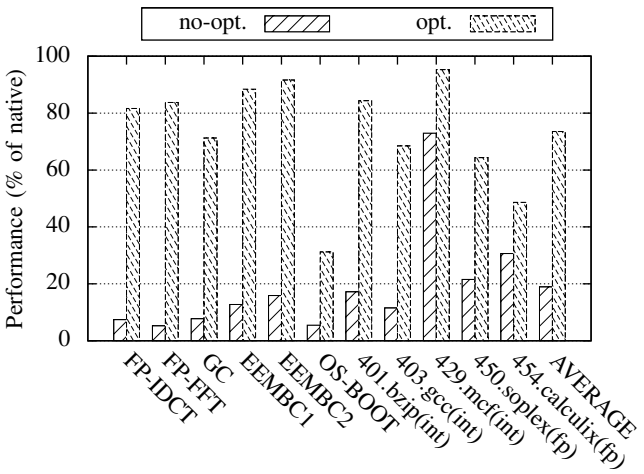

Fig. 7. Overall Performance Improvement

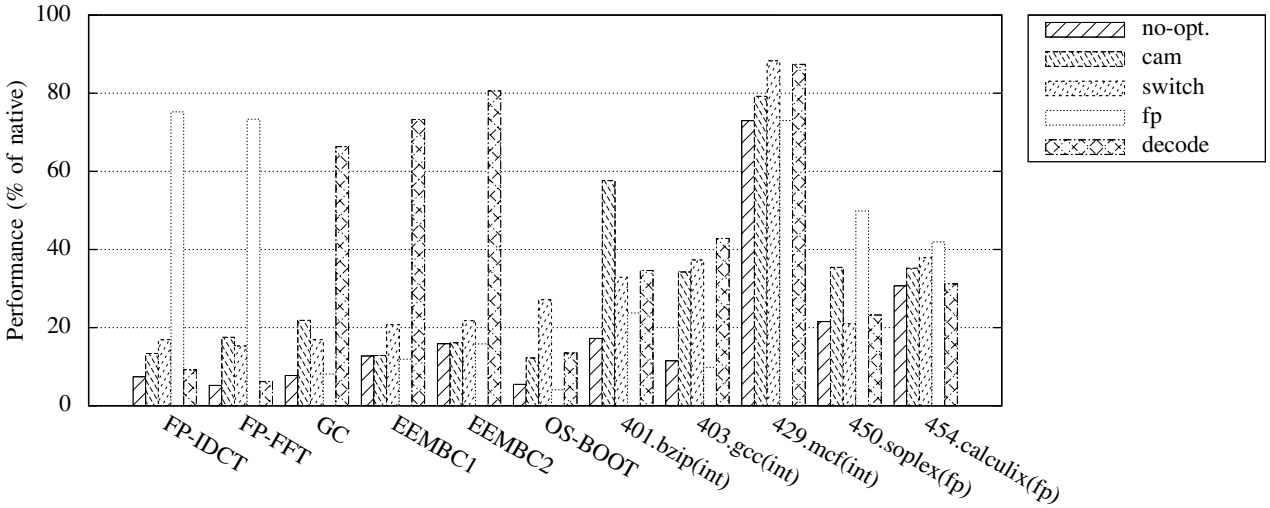

Fig. 6. Performance Improvements by Different Hardware Supports

VI. Conclusion And Future Work

This paper presents a novel hardware-software co-designed method to accelerate the binary translation on an existing ISA and the method is suitable to binary translation of any source and target architectures. Based on the method, several key common factors to the performance of binary translation are identified and corresponding hardware supports are proposed. Experiments on the Godson-3 RTL simulation and FPGA emulation show that this method helps the binary translation system to achieve high performance and simplify the software design. Also the hardware supports cost little.

Our recent work will focus on the research of ISA level compatibility and more common hardware supports to achieve native execution speed on the binary translation system. Further researches on parallel binary translation and optimization in the Godson multi-core environment also will be performed.

VII. Acknowledgments

We would like to thank all members of the Godson research group. And our work is supported by the National Basic Research Program of China under Grant No. 2005CB321600, the National High-Tech Research and Development Program of China under Grant No. 2008AA010901, the National Natural Foundation Key Program of China under Grant No. 60736012 and the National Natural Science Foundation of China under Grant No. 60673146. We would also like to thank the anonymous reviewers for their feedback.

References

[1] E. Altman, D. Kaeli, and Y. Sheffer, "Welcome to the opportunities of binary translation," *IEEE Computer*, vol. 33, pp. 40–45, 2000.

[2] E. Altman, K. Ebcioglu, M. Gschwind, and S. Sathaye, "Advances and future challenges in binary translation and optimization," *Proceedings of the IEEE, Special Issue on Microprocessor Architecture and Compiler Technology*, vol. 89, no. 11, pp. 1710–1722, 2001.

[3] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 Execution Layer: A Two-Phase Dynamic Translator Designed to Support IA-32 Applications on Itanium-Based Systems," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, vol. 36, 2003, pp. 191–204.

[4] K. Ebcioglu, E. Altman, M. Gschwind, and S. Sathaye, "Dynamic binary translation and optimization," *Computers, IEEE Transactions on*, vol. 50, no. 6, pp. 529–548, 2001.

[5] E. Witchel and M. Rosenblum, "Embra: fast and flexible machine simulation," in *Proceedings of the 1996 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. New York, NY, USA: ACM, 1996, pp. 68–79.

[6] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. Bharadwaj Yadavalli, and J. Yates, "FX!32 a profile-directed binary translator," *IEEE Micro*, vol. 18, no. 2, pp. 56–64, 1998.

[7] K. Ebcioglu and E. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," in *Proceedings of the 24th annual International Symposium on Computer Architecture*, 1997, pp. 26–37.

[8] A. Klaiber *et al.*, "The Technology Behind Crusoe Processors," Transmeta, Tech. Rep., 2000.

[9] W. Hu, F. Zhang, and Z. Li, "Microarchitecture of the Godson-2 Processor," *J. Computer Science and Technology*, vol. 20, no. 2, pp. 243–249, 2005.

[10] 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.

[11] R. Hookway and M. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation," *Digital Technical Journal*, vol. 9, pp. 3–12, 1997.

[12] C. Zheng and C. Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," *IEEE Computer*, vol. 33, pp. 47–52, 2000.

[13] M. Probst, "Fast machine-adaptable dynamic binary translation," in *Proceedings of the Workshop on Binary Translation*, vol. 9, 2001.

[14] S. B. Vasanth Bala, Evelyn Duesterwald, "Transparent Dynamic Optimization: The Design and Implementation of Dynamo," HP Laboratories Cambridge, Tech. Rep., 1999.

[15] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *USENIX 2005 Annual Technical Conference*, 2005.

[16] J. E. Smith and R. Nair, *Virtual Machines - Versatile Platforms for Systems and Process*. Elsevier, 2005.

[17] T. R. Halfhill, "Transmeta breaks x86 low-power barrier," *Microprocessor Report*, vol. Feb., pp. 1–11, 2000.

[18] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson, "The Transmeta Code Morphing Software: using speculation, recovery, and adaptive retranslation to address real-life challenges," in *ACM International Conference Proceeding Series*, vol. 37, 2003, pp. 15–24.

[19] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "Dynamic and transparent binary translation," *IEEE Computer*, vol. 33, pp. 54–59, 2000.

[20] J. Wang, “Co-design and Co-optimization of x86 virutal machine on general RISC platform,” Ph.D. dissertation, Graduate University of Chinese Academy of Sciences, 2008.
[21] “SPEC CPU benchmark suite,” http://www.spec.org/benchmarks.html.
[22] H.-S. Kim and J. E. Smith, “Hardware support for control transfers in code caches,” in *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2003, p. 253.
[23] H. Kim, “A co-designed virtual machine for instruction-level distributed processing,” Ph.D. dissertation, University of Wisconsin, 2004.
[24] M. K. Gschwind, “Method and apparatus for determining branch addresses in programs generated by binary translation,” IBM, Tech. Rep., 1998.
[25] “Incisive Xtreme Series Datasheet,” http://www.cadence.com/rl/Resources/datasheets/Cadence_6569_DS_R2.pdf.