Hardware/software co-design of Dynamic Binary Translation in X86 Emulation

Hongqi He
Department of Computer Science and Technology
NDSC
Zhengzhou, China
hhq@ndsc.com

Haifeng Chen
Department of Computer Science and Technology
NDSC
Zhengzhou, China
Haifengchen.work@gmail.com

Liehui Jiang
Department of Computer Science and Technology
NDSC
Zhengzhou, China
jiangliehui@163.com

Weiyu Dong
Department of Computer Science and Technology
NDSC
Zhengzhou, China
cidentifier@yahoo.com.cn

***Abstract*—X86 emulation is an effectively method to solve the problem of software compatible between X86 and RISC processors, such as ARM, PowerPC, Alpha and so on. Dynamic Binary Translation (DBT) in X86 emulation translates the X86 binary codes to RISC binary code dynamically so that the software based on X86 platform could execute undifferentiated on RISC platform. However, the DBT based on software is one of the performance bottlenecks nowadays. In this case, this paper discusses a new method for DBT with hardware/software co-design.**

A hardware unit is designed to accelerate the DBT system, including Instruction Decoder, RISC Code Table, Translation Cache and Cache Query Unit. Instruction Decoder analyses the meaning of X86 binary codes and then looks up RISC Code Table to obtain the corresponding RISC binary codes. Translation Cache stores the recently translated RISC binary codes to reduce the repeated instruction translation. Cache Query Unit is used to determine whether cache hit or not. Finally, we achieve the hardware unit using Verilog HDL. Experiment showed that the co-design DBT system could work accurately.

Keywords- X86 Emulation; Instruction Translation; Co-design; Dynamic Binary Translation; Hardware Unit

I. INTRODUCTION

RISC processors have been tremendous growth recently, such as ARM, PowerPC and so on. However, processors based on X86 Architecture keep their dominant place in the CPU market share, and users have been accustomed to using X86 software. To expand the market, RISC processors have to support compatibility for software based on X86. A new solution is proposed to solve this problem, which is called X86 emulation [1]. X86 emulation provide a virtual X86 machine for users on RISC processors so that the software based on X86 platform could execute undifferentiated on RISC platform.

Dynamic Binary Translation (DBT) is one of the most important components in X86 emulation. It translates the X86 binary codes to RISC binary code dynamically. Recently, there are many open-sources DBT system, such as QEMU [2], Bochs [3] and so on. However, they are both programed by software and the efficiency is far from practical application. Experiment showed that the DBT based on software could reach only 10%-15% of the actual performance [4]. This paper discusses a new frame for DBT with hardware/software co-design. A hardware Unit is designed to accelerate the DBT system.

II. OVERVIEW OF X86 EMULATION

X86 emulation provide an entire X86 hardware system, which is called Guest Machine (GM) including GM Memory, GM CPU and GM I/O. And the RISC computer is called Host Machine (HM). GM CPU record a state of X86 registers. It is not a physical processor but a structure of variables in HM's memory space. GM memory is also a block of continuous region allocated from HM's memory space. All of the X86 instruction and data will be loaded here. GM I/O maintains a series of I/O interface registers. The In and Out instruction may change the context of the registers. HM's OS monitors the changes and then deals with the I/O request.

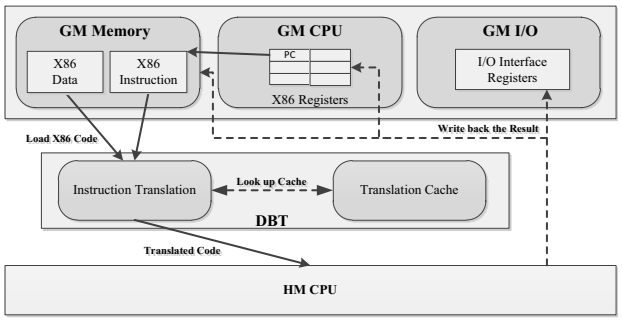

Figure 1. The Structure of X86 Emulation

As shown in Figure 1, the DBT system includes Instruction Translation and Translation Cache. Instruction

978-1-4673-0089-6/12/$26.00 ©2012 IEEE

Translation reads the address of next instruction from PC register and loads the X86 binary codes from GM Memory. The X86 binary codes are translated to Alpha binary code dynamically and then sent to HM CPU. The result is written back to GM CPU, GM Memory and GM I/O. However, according to the Principle of Locality, the translated codes may be accessed again. Thus, Translation Cache is designed to record the recent translated codes. Before translating, the cache is looked up firstly.

III. RELATED WORK

In order to Intel converts the CISC ISA into what is essentially a RISC ISA running on a dynamic superscalar microarchitecture starting from Pentium processor. And this CISC-to-RISC conversion is done entirely in hardware. As shown in Figure 2.

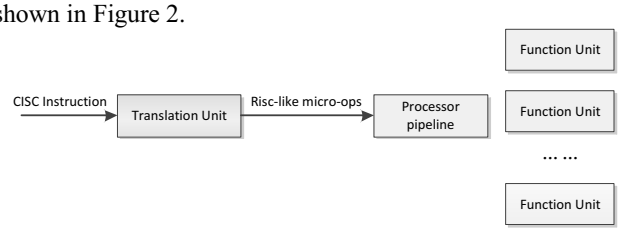

Figure 2. CISC-to-RISC conversion by hardware

Also, this conversion provides a new solution for X86 emulation. We can use hardware to accomplish the instruction translation in the DBT system. Recently, the related work is researched by many research institutions, Universities and Companies.

Transmeta Cruso is a very long instruction word (VLIW) CPU [5]. And the hardware is surrounded by a software layer, which is called Code Morphing (CM). The CM software dynamically translates x86 instructions into VLIW instructions so that the software based on X86 platform can be executed on VLIW platform. To improve the performance, Cruso designs special hardware to support the CM software such as shadow register, alias hardware and so on.

Godson-3 is a MIPS64 RISC processor. To support x86 emulation, It provides dedicated hardware to supports efficient x86 to MIPS binary translation. Godson-3 defines EFlag counterpart instruction and runtime environments to bridge the gap between the x86 and MIPS64 ISAs [6]. The DBT for Godson-3 is an improved system based on QEMU.

However, these solutions depend mainly on software. Hardware is only used as an assist. For example, many registers and instructions are added in to support the Profiling, which gathers the trace of a program at run-time.

In this paper, we propose a new frame of DBT with co-design. In our design, most of the function is realized by hardware. Software is used to do some data transmission such as reading data from GM Memory and sending them to the hardware unit and so on.

IV. CO-DESIGN OF DBT

Our co-design of DBT includes a software-based Data Loader and a hardware unit. Data Loader reads X86 binary codes to be executed from GM Memory and send them to the hardware unit to be translated. The hardware unit is composed of 4 parts, which are Instruction Decoder, RISC Code Table, Translation Cache and Cache Query Unit, as shown in Figure 3. In this secession, we will talk about the design of the hardware unit.

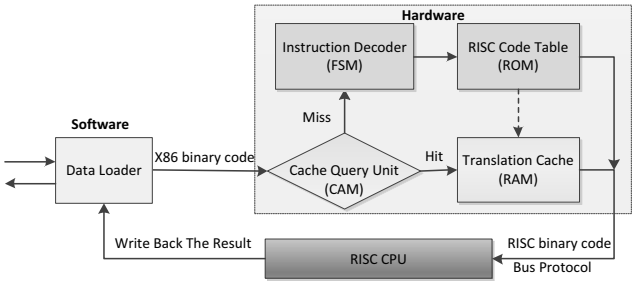

Figure 3. Co-design of DBT system

A. Instruction Decoder

Instruction Decoder fetches a series of X86 binary codes from Data Loader and decodes them to intermediate structure according to the format of X86 instructions.

As shown in Figure 4, X86 instructions consists of Prefixes, Opcode, ModR/M, SIB, Displacement and Immediate. The intermediate structure contains all information of X86 instructions, such as operation, source code, destination code, addressing and so on.

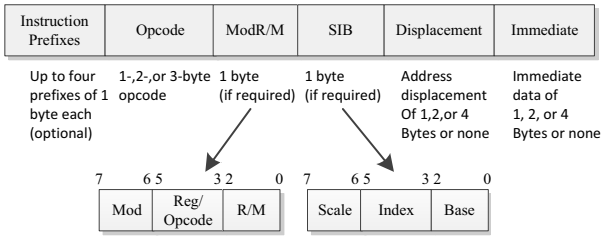

Figure 4. X86 Instruction Foramt

However, X86 instructions are not fixed length. An X86 instruction may not include Prefixes, ModR/M and SIB. The shortest has only 1 byte, but the longest is up to 11 bytes. Thus, Instruction Decoder couldn't decode them with the help of data location.

To decode X86 instruction accurately and efficiently, we design a finite-state machine (FSM) for Instruction Decoder. According to the format of X86 instructions, 6 states are defined to describe the FSM including *Pre*, *Opc*, *ModR/M*, *SIB*, *Disp* and *Imm*. Also, a *Start* and *End* state is used to mark the beginning and end of the FSM.

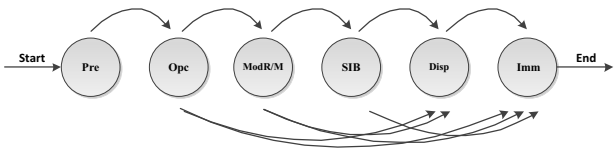

Figure 5. FSM for Instruction Decoder

Figure 5 shows the State-transition Diagram of the FSM. And the regulation of state transition is listed below.

- When to start, Instruction Decoder gets 1 byte from the X86 binary code queue and checks that whether it is equal to Repeat/lock prefix, String manipulation prefixes, Segment override prefixes, Operand override prefix or Address override prefix. If the result is true, the next state is *Pre*, else it is *Opc*.
- If the state is *Pre*, the next state must be *Opc*.
- If the state is *Opc*, *ModR/M*, *Disp*, and *Imm* should be checked whether they are existed. A according to the result, the next state may be *ModR/M*, *Disp*, *Imm*, or *End*.
- If the state is *ModR/M*, the addressing of source code and destination code could be obtain. Also, the *SIB* may be needed to provide an extended addressing. The next state may be *SIB*, *Disp* or *Imm.*
- If the state is *SIB*, a kind of addressing just like [Scale* index + base] is provide. The next state may be *Disp* or *Imm.*
- If the state is *Disp*, address displacement of 1, 2, or 4 bytes could be obtain. The next state may be *Imm* or *End.*
- If the state is *Imm*, it means an X86 instruction is decoded completely, and the next state is *End.*
- In order to decode the X86 binary codes continuously, we define the *Start* state and *End* state as the same state.

In our design, Instruction Decoder fetches 1 byte per clock cycle. Thus, an instruction's decoding process take up to eight clock cycles at most.

Instruction Decoder analyzes the information of X86 binary codes and records them into the intermediate structure. The operation, source operand and destination operand of the codes have been obtained.

B. RISC Code Table

A DBT based on software could take a variety of optimization measures to improve the performance. However, hardware is not as flexible as software. If the co-design DBT takes the same approach as what software does, it is very difficult for hardware to achieve the same function. Additionally, the advantage of the hardware could not been embodied.

TABLE I. ADD INSTRUCTION FORMAT IN X86 ISA

Opcode	Dest	Src	Sizebyte
add	Reg	Reg	2
add	Mem	Reg	2-4
add	Reg	Mem	2-4
add	Reg	Imm	3-4
add	Mem	Imm	3-6
add	Accum	Imm	2-3

For these reasons, we design a RISC Code Table for our co-design DBT. RISC Code Table is a look-up table which records the equivalent RISC codes for each kind of X86 instructions. For example, the "Add" instruction in X86 ISA has 6 different kinds of format, as listed in Table 1. We program the corresponding RISC instructions for each format and store them into RISC Code Table.

An X86 instruction may be translated to number of RISC instructions. So, we set a *Number Flag* in RISC Code Table. This flag records the number of the rest translated instructions which have not been read.

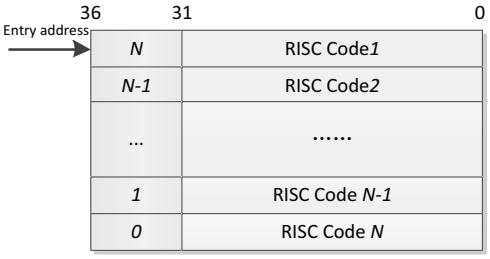

Figure 6. RISC Code Table

As shown in Figure 6, there are *N* RISC instructions for an X86 instruction. Instruction Decoder sends the first RISC code's *entry address* to start read the corresponding entries one by one. RISC Code Table checks the *Number Flag* of the current entry. If *Number Flag* is above 0, the reading operation is going on. When *Number Flag* is equal to 0, it means this entry is the last of the equivalent RISC codes and the reading operation should be stopped.

C. Translation Cache

The process of instruction translation is one of the performance bottlenecks in the DBT system. To reduce the numbers of instruction translation, we design a Translation Cache to record the RISC codes translated in the most recent period.

To a cache, the most important evaluation is the hit rate. And the hit rate is affected by cache management algorithm. In our design, we divide Translation Cache into 2 areas, the Hot Code area and the Cold Code area, as shown in Figure 7.The size of Cold Code area is twice bigger than Hot Code area.

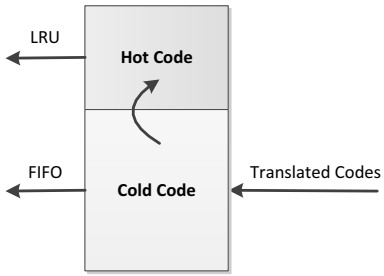

Figure 7. Management of Translation Cache

While an X86 program is running, all of the translated Codes would be sent to the Cold Code area firstly. Because of large number of translated code, we prefer the FIFO algorithm to manage the Cold Code area. However, many translated code may be accessed again, but the FIFO algorithm could not keep the again-accessed entries in the cache. Therefore, we set a *Counter flag* for each entry. The *Counter flag* records the number of an entry's accessed times.

When the entry is accessed again, the counter is increased by 1. Before updating the Cold Code area, the *Counter Flag* should be checked firstly. If the counter is up to the threshold value, that means this entry is a hot entry, the entry would be transferred to the Hot Code area.

Ang-Chih *et al* researched the number of entries' accessed times of code cache in DBT system [7]. According to their experiment, we define the threshold value as 2.

The Hot Code area records the high frequency accessed entries. And each of them may be accessed again. Thus, we prefer the LRU algorithm to manage this area.

A problem is that times of entries' transfer would bring about a large of overhead. To avoid entries' transfer, we divide the Translation Cache only in logic but not in physical. An *Area Flag* is used to distinguish the two areas for each entry. If an entry's counter is up to the threshold value, we only need to change the *Area Flag*.

Additionally, a *Valid Flag* and a *Length Flag* are added in. The *Valid Flag* is used to sign whether the entry is useful for the DBT system. If an entry is swapped out of Translation Cache, the *Valid Flag* would be cleared. Also, self-modifying codes would clear the *Valid Flag*. The *Length Flag* has the same function as the *Number Flag* in RISC Code Table. The format of the Translation Cache entry is shown in Figure 8.

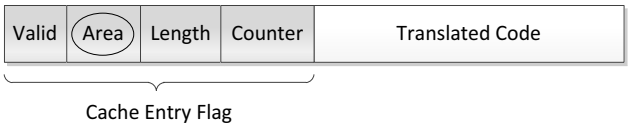

Figure 8. Translation Cache Entry

D. Cache Query Unit

Cache Query Unit is used to make sure whether Translation Cache is hit or not. Before the instruction translation, the DBT system looks up Cache Query Unit. If the current value of PC Register in GM CPU has been found in the unit, it means the current X86 codes have been translated and the cache entry would be output. With the help of cache entry, the equivalent RISC codes could be read easily from Translation Cache. However, if the current value of PC has not been found, the DBT system has to start Instruction Decoder and RISC Code Table to accomplish the X86-to-RISC conversion.

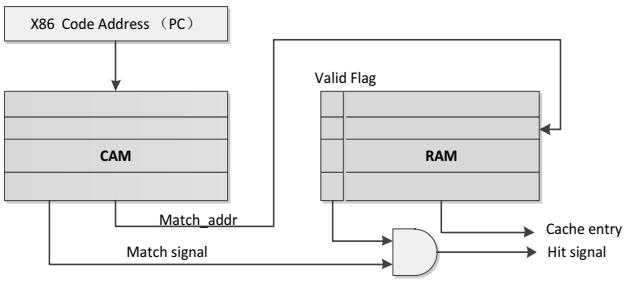

Figure 9. The Structure of Cache Query Unit

The Cache Query Unit consists of A CAM and A RAM, as shown in Figure 9. The CAM is defined as content-addressable memory. Unlike a common RAM, CAM produces an address for a given data word. Also, CAM searches all addresses in parallel.

The CAM records the address of translated codes. Cache Query Unit compares the address of current X86 codes which need to be translated. If CAM hit, the *Match Signal* would be set, and the *Match_addr Signal* gives the corresponding address for RAM.

The RAM records the cache entries. From *Match_addr Signal*, the equivalent RISC Code entry is obtained. Then, the translated codes could be red.

A *Valid Flag* is used to sign the availability of the cache entry. And this flag is synchronized with the *Valid Flag* in Translation Cache.

E. Data Loader

Data Loader is the only component based on software in our co-design DBT system. It is running on the OS of HM. The main function of Data Loader is to fetch the X86 binary codes from GM Memory and to send them to the hardware Unit.

Data Loader maintains the interface between software and hardware. It writes the data to a special memory space and this memory space is mapped to a buffer in the hardware Unit.

V. Implement of Co-design DBT

We implement the hardware of the co-design DBT using Verilog HDL. And the ISim of Xilinx ISE is used for functional simulation. The chosen RISC processor is Alpha. Experiment showed that the co-design DBT system could realize the instruction translation correctly.

Figure 10 and Figure 11 shows the process of Instruction Translation. The X86 instruction "26h 81h 0Eh 65h 10h FCh FEh" (OR WORD PTR ES: [1065h], FEFCh) are translated up to 74 Alpha Instructions. That is because this instruction has complex addressing and several times memory access are needed. But Alpha provides only Load and Store instructions to deal with the memory access operations.

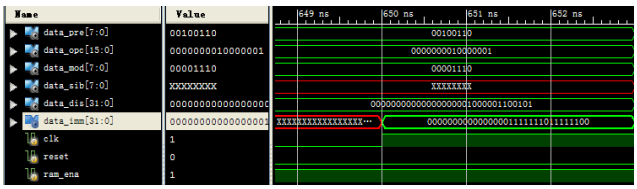

Figure 10. X86 Binary Codes Decoding

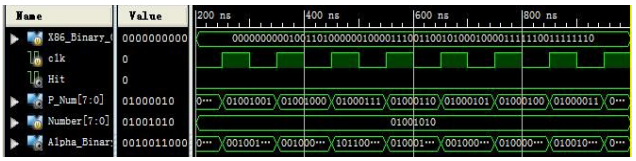

Figure 11. The Translated Alpha Binary Codes

VI. CONCLUSION

X86 emulation on RISC processor could support compatible for software based on X86 ISA. And the DBT system is one of the most important components in X86 emulation. To improve the performance, a co-design DBT is designed to accelerate the instruction translation. The co-design DBT system includes Instruction Decoder, RISC Code Table, Translation Cache, Cache Query Unit and software-based Data loader. Experiment showed that the co-design DBT system could work accurately.

Our recent work is to reduce the clock cycle in Instruction Decoder. Research on high performance interface between software and hardware is also in progress.

REFERENCES

[1] Liehui Jiang, Haifeng Chen, Jianping Lu and et al. Prefetching Strategy for Address Translation in IA-32 Emulation[C]. Advances in Intelligent and Soft Computing, 2012, Volume 124/2012, pp.703-708. DOI: 10.1007/978-3-642-25658-5_83.

[2] F. Bellard. Qemu, a fast and portable dynamic translator. In USENIX Annual Technical Conference,FREENIX Track, April 2005.

[3] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. Linux J.,1996(29es):7, 1996.

[4] Chen Hai-feng, Jiang Lie-hui, Dong Wei-yu and et al. An Emulation Model of IA-32 Memory Management[C]. 2011 International Conference on Intelligence Science and Information Engineering (ISIE), wuhan, 20-21 Aug. DOI: 10.1109/ISIE.2011.41.

[5] Alexander Klaiber, "The Technology Behind the Crusoe Processors," http://www.charmed.com/PDF/CrusoeTechnologyWhitePaper_1-19-00.pdf, Dec. 2011

[6] Hu W, Wang J, Gao X, Chen Y, Liu Q, Li G J. Godson-3: A scalable multicore RISC processor with X86 emulation.IEEE Micro, 2009, March/April, pp.17-29.

[7] Ang-Chih Hsieh,Chun-Cheng Liu,TingTing Hwang. Enhanced Heterogeneous Code Cache management scheme for Dynamic Binary Translation.2011 16th Asia and South Pacific Design Automation Conference, pp.231 - 236. DOI: 10.1109/ASPDAC.2011.5722189.