An Emulator for Executing IA-32 Applications on ARM-Based Systems

Wei Chen
School of Computer, National University of Defense Technology, Changsha 410073, Hunan, China
Email: jizhuchenwei@hotmail.com

Zhiying Wang
School of Computer, National University of Defense Technology, Changsha 410073, Hunan, China
Email: zywang@nudt.edu.cn

Dan Chen
School of Computer Science, University of Birmingham, Edgbaston, Birmingham, B15 2TT, United Kingdom
Email: d.chen@cs.bham.ac.uk

Abstract—**Virtual Machine (VM) can not only release the processor designers from the burden of ensuring cross-platform compatibility but also provide new opportunities for innovation. Instruction Set Architecture (ISA) emulation is the key aspect of a VM. This paper describes the design and implementation of TransARM-IU, a use-level ISA emulator that supports IA-32 applications on ARM-based systems. This paper also discusses several dedicated solutions to several crucial issues related to the development of TransARM-IU, such as the Executable and Linking Format resolution and hybrid threaded interpretation. In particular, conventional interpretation is implemented in a centralized style which consumes much more execution time. A hybrid threaded interpretation method is proposed to improve the efficiency of the emulator by appending a portion of the simple decoding and dispatching code to the end of each of the instruction interpreter routines. For performance evaluation, we selected 6 benchmarks from MiBench and carried out the experiments on two real ARM-based systems. Experimental results demonstrate the correctness of TransARM-IU in terms of ISA emulation and indicate that TransARM-IU is competitive to other ISA emulators.**

Index Terms—**virtualization, interpretation, emulation**

I. INTRODUCTION

Computer architects are confronted by two fundamental issues: (1) Architecture innovations, which enable efficient system designs to achieve higher reliability, lower power consumption with lower complexity and cost, are required; (2) after decades of development, a huge amount of applications/software have been accumulated for legacy Instruction Set Architectures (ISAs), especially for x86 architecture. Unfortunately, the two issues seems to conflict with each other by nature. Software designed for one ISA can not directly execute on another because of the incompatibility problem. This has inhibited modern processor designers from developing new ISAs and limits the reuse of the large amount of existing applications.

Virtual Machine (VM) [1][2][3] is implemented by adding a software layer to an execution platform to give users the appearance of a different platform. A wide range of computer system resources, such as processors, memory, and I/O devices, including network resources can be virtualized. Thus, VMs can not only release the processor designers from the burden of ensuring cross-platform compatibility but also provide new opportunities for innovation.

There exists a wide variety of VMs developed by different groups. There are underlying technologies that are common to a number of virtual machines while each VM also has its unique characteristics. Especially, the source architecture and the target architecture have significant impact on the implementation of a VM. Computer architecture is evolving continuously, and a large amount of open problems left for VM designers. This motivates us to explore the design of the VMs crossing specific source and target architectures.

This study aims to implement a VM which can enable IA-23 [4] (the most widely used x86 architecture) applications to run on the ARM-based platforms. ARM is the de facto industry standard for 32-bit embedded processors, which can be found in billions of mobile phones, Apple iPods, and other products and systems. On the other hand, the x86 architecture dominates personal computer systems and has the support of the most software resources comparing to other architectures. It is desirable to directly execute software resources the x86 architecture on ARM-based platforms. We anticipate that this will be feasible with the support of the virtual machine technology.

There exist a number of VM approaches, such as process VMs and system VMs, co-designed VMs [3]. Despite the variety of VMs, any cross-platform VM needs to provide the ability of ISA virtualization so as to support a binary executable compiled for one ISA on the target platform which has a different ISA. ISA virtualization/emulation is the first step to implement a full system VM across x86 and ARM. Virtualization at the instruction set architecture level is a basic method for implementing a VM. To distinguish it from the more powerful 'full' virtualization that takes all the hardware resources into account, we used instruction architecture emulation to improve accuracy. This paper discusses the implementation of an ISA emulator named TransARM-IU ("I" and "U" stands for interpretation and user level respectively), which adopts interpretation support IA-32 applications on ARM-based systems at user level.

A complete ISA consists of instructions, register architecture, memory architecture, trap and interrupt architecture. TransARM-IU focuses on emulating the operation of user-level instructions in the absence of exceptions (traps and interrupts). ISA emulation can be carried out using a variety of methods that require different amount of computing resources and offer different performance characteristics. At one end of the spectrum is Dynamic Binary Translation (DBT), while on the other is interpretation. Though the interpretation technique may incur a larger execution overhead than DBT does, it involves a much smaller cost of initial emulation [3]. Furthermore, the implementation of interpretation is much easier than DBT's. It enables an easier architecture mapping without concerning issues like context switching and control transfer chaining [5] in contrast to DBT. Therefore, TransARM-IU adopts interpretation as its basis. In our previous work [6][7], we identified that conventional interpretation may cause larger execution cost because of the redecoding operations, and this problem has been properly addressed via a Pcache solution. TransARM-IU uses a hybrid threaded interpretation infrastructure evolved from the Pcache approach to minimize the interpretation overhead.

ISA emulation also involves issues like architecture mapping (register mapping, memory space mapping), executable file resolving. These issues have been properly addressed in the implementation of TransARM-IU.

The goal of this study is to provide a transparent, high performance instruction emulator and to explore the platform-dependent issues as well as the common underlying issues in implementing the cross-platform VMs. The scope and objectives of this paper can be as follows:

- To design a user-level instruction emulator that supports IA-32 applications on ARM-based systems;
- To speed up the instruction emulation process;
- To enable architecture mapping and executable file resolving.

The rest of this paper is organized as follows. We first introduce some related works (Section II). We then present an overview of the TransARM-IU infrastructure (Section III). We discuss several crucial issues involved in TransARM-IU (Section IV). We then present our experimental results (Section V), and finally conclude (Section VI).

II. Related Work

A software emulation system can be interpretation based, translation based, or hybrid. For example, Transmeta Crusoe [8] processor's underlying architecture is VLIW (Very Long Instruction Word) while the source ISA is the Intel IA-32. It employs a "Code Morphing Software" (CMS) layer logically surrounding the hardware engine. CMS is a two-stage emulation software which uses interpretation to emulate x86 code initially and translates the frequently executed portions of the source code into native VLIW code, no matter the source code originally works at user level or system level.

Bochs [1][10] is a highly portable open source IA-32 emulator written purely in C++. It is a full system emulator includes emulation of the CPU engine, common I/O devices, and custom BIOS. Bochs can be compiled to emulate any modern x86 CPU architecture, including most recent Core2 Duo instruction extensions. Bochs uses pure interpretation to emulate the x86 ISA. The interpretation infrastructure is organized as a central loop of fetching, decoding and dispatching instructions. To reduce the emulation time, Bochs utilizes some optimization techniques such as lazy flags updating, staged memory reference resolving, decoded instruction trace cache, et al.

QEMU [11][12] is another fast machine emulator which emulates several ISAs (x86, PowerPC, ARM and Sparc) on multiple targets (x86, PowerPC, ARM, Sparc, Alpha and MIPS). It is the unique emulation system reported that can emulate x86 instructions on ARM. The QEMU itself runs on the target operating system (OS) and supports full system emulation. The kernel of QEMU is a portable dynamic translator, which performs a runtime conversion of the source CPU instructions into the target ISA. Each of the source instructions are first split into several simpler instructions called micro operations. A micro operation is first implemented by a small piece of C code and then compiled by GCC to an object file. However, QEMU does not apply in emulating x86 architecture on ARM platform.

Other well-know software emulation systems include: (1) IA-32 EL [13], which converts IA-32 instructions into Itanium instructions via dynamic binary translation; (2) Digital FX!32 [14], which supports running x86 Windows applications on DEC Alpha platforms; and (3)DAISY [15], which interprets PowerPC instructions before invoking "tree-region" translation.

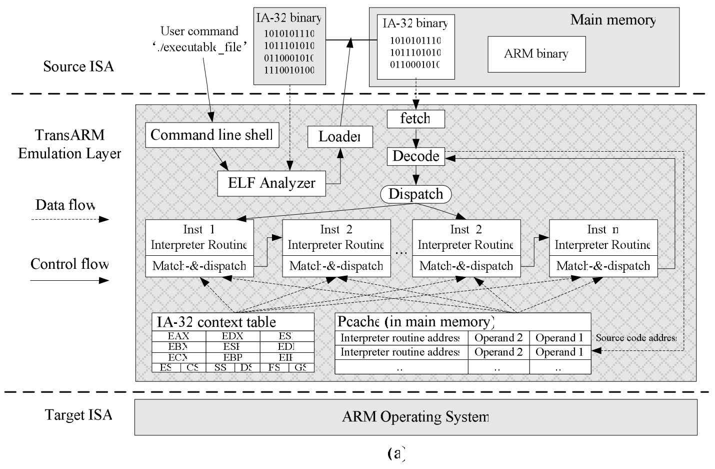
Existing emulation systems employ optimization techniques to accelerate the emulation process and to simplify implementation, i.e., software interpretation and/or dynamic binary translation. Interpretation is portable but also much slower than direct execution on the target processor. On the other hand, dynamic translators are usually difficult to be ported because the whole code generator must be rewritten according to the target ISA. Binary translation has a smaller execution cost but a larger initial translation cost than the interpretation does. An appropriate method should be selected according to the characteristics and the performance requirements when designing an emulation system. The proposed emulation system in this study (1) aims at emulating complex IA-32 instructions on ARM based platforms, (2) bases on interpretation to offer portability, acceptable start-up performance, and low memory requirement, (3) considerably reduces the interpretation overhead thus to improve the performance of the entire emulation system, and (4) can handle most user-level instructions.

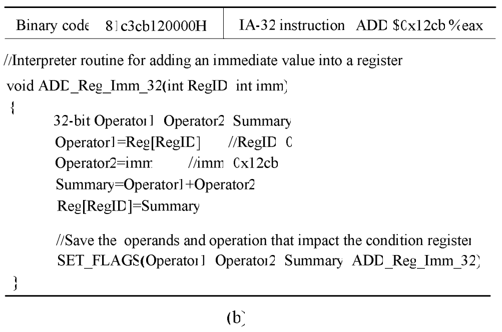
III. General Architecture of TransARM-IU

TransARM-IU is targeted for user-level emulation only and is implemented in C to ensure portability. TransARM-IU is loaded to the same user space as the native application. The entire emulation process is transparent to the user who feels like operating on a normal ARM-based environment. An overview of the infrastructure of TransARM-IU is shown in Fig. 1(a). From a user's perspective, TransARM-IU looks like a sandbox where the inputs are user-level executables compiled on the IA-32 platform with the option "-static" and "-msoft-float". The first option "-static" appends the source code of the library calls to the executable file and the second option "-msoft-float" coverts the floating-point instructions to the software routines emulating the behaviors of the floating-point instructions.

Calls to the library routines such as 'printf' are assumed by many emulation systems, such as UQDBT [16] and FX!32 [14], existing on the source as well as the target machine. This assumption is not restrictive as long as there is a mapping from a source library routine to its equivalent on the target machine, i.e., libraries of the source architecture can be reproduced on the target machine via translation or rewriting. In fact, it is not trivial to establish a map from the IA-32 C library to the standard ARM C library. Therefore, only statically compiled programs are taken into account in TransARM-IU. The compiling option '-static' can appends the source code of the required library routines to the executable file.

Currently, TransARM-IU mainly concerns the emulation of integer instructions. There are two reasons for ignoring floating-point instructions: First, the Vector Floating-Point (VFP) architecture [17] is a coprocessor extension to the ARM architecture. Some ARM-based systems may not provide the floating-point instruction set. Secondly, most floating-point instructions can be converted into software functions of the integer instructions by setting the option '-msoft-float' during compiling, i.e., the software function can be exploited to 'emulate' the floating-point instruction.





Binary code 81c3cb120000H	IA-32 instruction ADD $0x12cb %eax

```
//Interpreter routine for adding an immediate value into a register
void ADD_Reg_Imm_32(int RegID  int imm)
{
        32-bit Operator1  Operator2  Summary
        Operator1=Reg[RegID]      //RegID  0
        Operator2=imm          //imm  0x12cb
        Summary=Operator1+Operator2
        Reg[RegID]=Summary

        //Save the  operands and operation that impact the condition register
        SET_FLAGS(Operator1  Operator2  Summary  ADD_Reg_Imm_32)
}
```

(b)

Figure 1. General Architecture of TransARM-IU. (a) The infrastructure of TransARM-IU; (b) An example of an instruction interpreter routine.

User-specified commands intended to run the executable files are intercepted by a modified command line shell, which invokes the ELF (Executable and Linking Format) analyzer (see Fig. 1) when necessary. An executable file's ELF format (details see Section IV) will first be resolved to identify its target architecture. The executable file can get direct execution if it's architected for ARM. When the executable file's source architecture is IA-32, TransARM-IU will load the executable file to the memory space following a direct mapping strategy, and it starts the kernel process of the emulation.

Interpretation is the key aspect in our design. An interpretation based system normally reads source instructions one at a time, performing each instruction interpreter routine in turn on software maintained version of the source architecture's state, including all architected registers and main memory. The resource holding the state does not necessarily have to be the same type as it does on the source platform. The key point of state mapping is that for all the resource holding the source state which is necessary for the execution of a source program, the associated resource in the target is readily defined. Hence, we use a straightforward mapping in TransARM-IU which means that all of the useful registers of IA-32 are mapped to the target memory. A context table will be created to contain the image of various components of the IA-32 architecture state, such as general-purpose registers (e.g., EAX, EBX), the program counter (EIP), condition codes (EFLAG), and miscellaneous control registers (e.g., ES, CS) as shown in Fig. 1(a). Architecture mapping of interpretation is much flexible as it does not occupy any hardware resource of the target architecture, which can not be avoided in DBT in contrast.

The mapped state will be updated during the emulation of each source instruction in order to emulate the results of the source program's native execution. However, computing all the condition code for a given source instruction needs many target instructions which can slow down the emulation considerably. In fact, although the condition register (EFLAG) is updated frequently, they are seldom used. Thus, updating the emulated condition register at an instruction granularity is not necessary. TransARM-IU adopts the lazy evaluation on the condition flag register, where the operands and operation that impact the condition register are saved instead of the condition register settings themselves [10]. This is shown in the example of an interpreter routine in Fig. 1(b). An interpreter routine contains one or several target instructions emulating a source instruction. Fig. 1(b) is an example of an interpreter routine (written in C language) corresponding to the IA-32 instruction that adds an immediate value to a general purpose register. TransARM-IU contains hundreds of interpreter routines, each corresponding to a dedicated type of the source instruction. For those simple user-level instructions such as arithmetic instructions, the interpreter routine is quite simple: the source instruction's behaviour is emulated straightforward like the example in Fig. 1(b).

Furthermore, IA-32 processors are "little-endian" machines [4]. Fortunately, since most ARM processors provide both the little-endian and big-endian modes [17]. The endianness issue can be ignored in TransARM-IU.

In summary, when a user runs an executable file, the TransARM-IU fist intercepts the commands and analyzes the ELF format of the executable file in order to identify the required architecture. If the executable targets on an ARM platform, the native executions can be directly performed. Otherwise, the executable file is loaded into the memory space following a direct mapping strategy. The emulation system then fetches the source code, decodes it and dispatches an interpreter routine according the instruction type in order to achieve the same effect as the source instruction does.

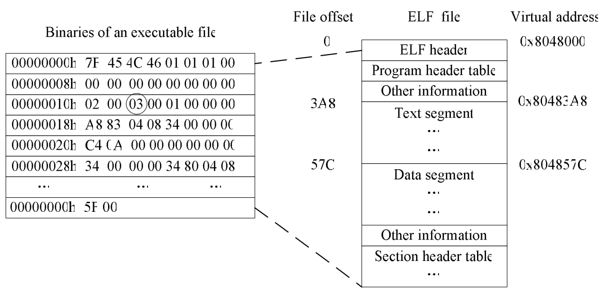
IV. Design and Implementations

How to organize the emulation procedure and dispatch the interpreter routines is crucial to the performance of an emulator. This section discusses the organization of the emulation procedure and other techniques in details.

A. ELF Analyzer and Loader

The first task of TransARM-IU is to identify the required ISA architecture of the executable file/program and to load the code and data into memory. This task is performed through the analysis of the Executable and Linking Format (ELF) [18] of the executable file.

An ELF file is the binary representation of a program. It contains information which participates in program linking (building a program) and program execution (running a program), such as instructions, data, symbol table, and relocate information. Fig. 2 shows an executable file's ELF organization: the left sub-figure indicates the binaries of the entire executable file; the right sub-figure shows the corresponding ELF organization of these binaries.



Binaries of an executable file
00000000h 7F 45 4C 46 01 01 01 00
00000008h 00 00 00 00 00 00 00 00
00000010h 02 00 03 00 01 00 00 00
00000018h A8 83 04 08 34 00 00 00
00000020h C4 CA 00 00 00 00 00 00
00000028h 34 00 00 00 34 80 04 08
... ...
00000000h 5F 00
File offset
0
3A8
57C
ELF file
ELF header
Program header table
Other information
Text segment
Data segment
Other information
Section header table
Virtual address
0x8048000
0x80483A8
0x804857C

Figure 2. Executable and linking format of an executable file

When an executable file begins to execute on the target ARM processor, it is the OS that resolves the file's ELF and loads the file into memory. Normally, the OS first checks the machine architecture the executable file requires. If the required architecture is ARM, the OS creates the memory image (code, data etc) and invokes the dynamic linker. Otherwise, an error occurs. In our design, the OS can be modified to identify the IA-32 executable file. To ensure user transparency and to avoid the efforts of modifying the OS kernel, we customized a dedicated command line shell to identify the executable file's machine architecture.

A user still executes an application/program by typing a command like "./executable_file_name" as normal. The command line shell (also known as command line interpreter) intercepts the command and invokes the ELF analyzer to identify the required architecture through "e_machine" of the ELF header. "e_machine" specifies the required architecture for an individual file as shown in Fig. 3. The values 3 and 40 represent the IA-32 architecture and ARM respectively. The circled value ("e_machine") in Fig. 2 indicates that this ELF file's required architecture is ARM. If the value is 40, the command line shell directly leaves the executable file with the OS. A value "3" means that IA-32 is required, the entire binaries of the file will be loaded into memory without the need for loading and linking.

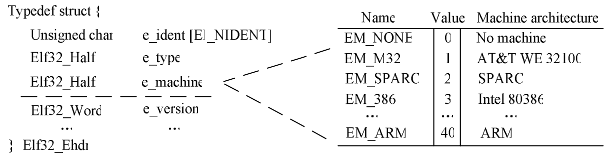


Figure 3. ELF header specifying the required architecture of an individual executable file.

The information specified for dynamic linking and loading in the original IA-32 executable file is no longer valid in the ARM environment. Therefore, TransARM-IU ignores the normal linking and loading steps. Instead, it loads all the binaries of the executable file (with the ELF information) into the virtual memory space according to the entry point (the beginning address of the executable file in the virtual memory) specified in the ELF header. In fact, TransARM-IU system treats all the binaries of an executed file as ordinary data, no matter they are text, data, or stack. As such, a straightforward mechanism has been established for "loading" the executable file without losing any necessary information.

There is an interesting observation in the implementation of TranARM IU that there is no conflict while loading the executable file into the virtual memory space of the ARM system at the address declared in the executable file. Both the data and text from the source IA-32 program can be mapped exactly at the same virtual address as it would normally have. According to the specification of the Application Binary Interface (ABI) [19] for Intel IA-32 architecture, a process's text segment typically resides at the virtual address of 0x8048000. This is also the beginning of an executable/ELF file. On the other hand, according to the ABI for ARM [20], address 0x8000 is usually used as the begging of an executable file. TransARM-IU itself is a user-level application and will be loaded into the memory with the beginning at 0x8000. Therefore, for most source applications, there will be no conflict if we load the executable file into the memory space of ARM from its original beginning address at 0x8048000. In fact, the straightforward memory mapping method avoids the overhead of address resolution. The source application's code and data remain unchanged, similar to their layouts on the original IA-32 platform.

B. *Pcache-Based Hybrid Threaded Interpretation*

The interpreter takes over on the completion of ELF analyzing and loading steps. A typical interpretation process, referred to as decode-and-dispatch interpretation, involves a cycle of fetching a source instruction, decoding and dispatching it to a required instruction interpreter routine. A conventional interpretation routine process can be viewed as a central loop (as shown in Fig. 4(a)), which is likely to be quite slow [3]. TransARM-IU adopts a hybrid threaded interpretation method to address this drawback based on our previous work [6][7].

In a conventional interpretation process, each interpreter routine finally turns back to the central decode and dispatches routines. These branches tend to deteriorate performance as they alternate the control flow. Threaded interpretation [21][22] reduces some of the branches by appending a portion of the decoding and dispatching code to the end of each of the instruction interpreter routines as illustrated in Fig. 4(b). This approach works efficiently with RISC ISAs. Because a modern RISC ISA such as the PowerPC has the regular instruction format (all instructions have the same length, and each field in the instruction is fixed), the interpreter can extract the opcode and then immediately dispatch it to the indicated instruction interpreter routine through a relatively small amount of code. On the other hand, a CISC ISA has a much more complicated format with varied instruction lengths and even field lengths. Appending the decoding codes (for a CISC instruction) into each interpreter routine will dramatically increase the interpreter routine's size, which may offset the performance improvement. For example, it costs TransARM-IU 650 lines of C code to decode and dispatch one IA-32 instruction (for the RISC ISA, the decoding and dispatching may consume only several lines of C code). Therefore, threaded interpretation is not appropriate to be directly adopted by interpretation of the CISC ISA.

TransARM-IU defines a Pcache for saving the decoded instruction information. In our previous work [6][7], it has been observed that instruction fetching and decoding contribute about 50% to the interpretation overhead, especially in the case of CISC ISA. It has also been noted that most instructions are reinterpreted many times, and this causes a considerable redundancy. Hence, the repeated decoding of source instructions forms a bottleneck of interpretation. We then proposed using an interpreted code cache (Pcache) to avoid most of the replicated decode operations by saving the interpreted instruction information for reuse. Each Pcache line contains the opcode, operand's addressing modes and the address of the dedicated interpreter routine of a source instruction, as depicted in Fig. 1(a). We have demonstrated that Pcache can dramatically reduce the re-decode operations thus improve the performance of the interpretation. Pcache can be implemented in the form of both software and hardware. TransARM-IU simply organizes Pcache as a hash table which contains 1024 Pcache lines. Each Pcache line contains the decoded information of a source instruction. Thus, the CISC ISA can be converted into an intermediate form with fixed length and fields. Saving the decoded instructions in Pcache can not only avoid most re-decoding operations [6] but also enable the threaded interpretation of the CISC ISA. Different from the threaded interpretation which replicates the decode-and-dispatch code, our method appends a small amount of match-and-dispatch code at the end of each of the interpreter routine. During interpretation, the interpreter will access the Pcache first according to the address of the source instruction. If the required instruction is found in Pcache, the interpreter will bypass the decoding process and directly turn to the next interpreter routine whose address is specified in the related Pcache line. If the required instruction is not found, the interpreter will return to the central decode-and-dispatch routine. The central decode-and-dispatch routine fetches the source code from the main memory through the data cache, and it performs a complex decoding operation and finally saves the decoded instruction information in Pcache for reuse. The hybrid threaded interpretation method can be illustrated in Fig. 4(c).

Fig. 4 highlights the differences in control and data flow between three different interpretation methods. For the traditional decode-and-dispatch method (Fig. 4(a)), control flow continuously exits from, and returns to, the central decode-and-dispatch routine. For the traditional threaded interpretation method (Fig. 4(b)), code pieces are replicated at the end of each of the interpreter routines.

For those CISC ISAs, the complicated decoding process always consumes tens even hundreds lines of C codes which dramatically increases the size of each of the interpreter routine. For our hybrid threaded interpretation method based on Pcache (Fig. 4(c)), simple match-and-dispatch code is appended into each of the instruction interpreter routine. For most common cases, required source instruction's information can be found in Pcache, thus one interpreter routine can directly turn to the next with the complicated decoding process avoided. The complex centralized decode-and-dispatch routine is seldom executed. An evaluation of our hybrid threaded interpretation method can be found in Section V.

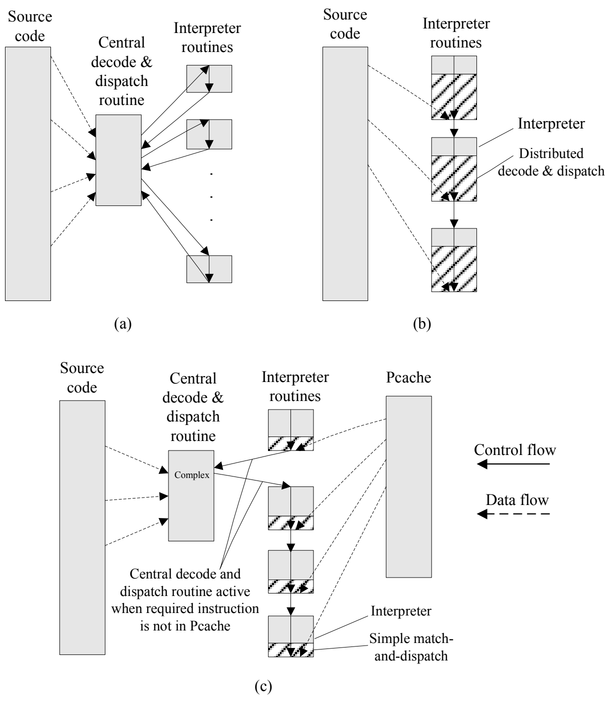

Source code
Central decode & dispatch routine
Interpreter routines
(a)
Source code
Interpreter routines
Interpreter
Distributed decode & dispatch
(b)
Source code
Central decode & dispatch routine
Complex
Interpreter routines
Pcache
Control flow
Data flow
Central decode and dispatch routine active when required instruction is not in Pcache
Interpreter
Simple match-and-dispatch
(c)

Figure 4. Control flow and data flow of different interpretation methods. (a) Central decode-and-dispatch interpretation; (b) normal threaded interpretation; (c) threaded interpretation based on Pcache, data flow from the decode-and-dispatch routine to Pcache is omitted.

V. EVALUATION

This section presents an evaluation on the performance of TransARM-IU.

A. Experiment Environment

We first compiled the source programs on an IA-32 platform with the option '-msoft-float' and '-static'. The executable files were then executed on two ARM platforms (named ARM platform 1 and platform 2) respectively. TransARM-IU ran on Linux on top of the ARM platform with user transparency supported. Detailed configuration settings of the source and target platforms are provided in Table II.

We carried out experiments on MiBench [23][24], a free and representative embedded benchmark suite. The large dataset provideed by MiBench for 'stringsearch' were used. MiBench totally consists of six categories of 38 benchmarks. We select one benchmark form each of the categories. Table III gives the brief description of the selected benchmarks.

TABLE I. MACHINE CONFIGURATION SETTINGS

IA-32		
Processor	Intel Pentium Dual CPU E2140 1.6GHz	
Linux version	Linux Red Hat Enterprise3, Linux kernel 2.4.21-4.EL	
GCC	3.2.3	
Compiling option	-msoft-float -static	
ARM		
	ARM platform 1	**ARM platform 2**
Processor version/ ARM architecture	OMAP 3530/ CortexTM-A8	Samsung S3C2440/ ARM920T
Frequency	600 MHz	400 MHz
Linux	ubuntu 8.04, Linux kernel 2.6.28-r18	Linux kernel 2.6.13
GCC version	4.2.1	3.4.1
I-Cache	16 KB	16 KB
D-Cache	16 KB	16 KB
L2 Cache	256 KB	None
Memory system	64 KB shared SRAM on chip, 112 KB ROM on chip, 256 MB, 166 MHz mDDR, 512 MB Nand Flash	64 MB SDARM, 64 MB Nand Flash, 2MB Nor flash

B. Performance Evaluation

We first executed the benchmarks directly on the ARM platforms for reference. The benchmarks were compiled by the native compiler on the ARM platform. The second column in Table IV lists the native execution times required by each of the selected benchmarks. Since modern IA-32 processors run definitely much faster than the ARM processors, the IA-32's native execution times were bypassed when evaluating the performance of TransARM-IU.

TABLE II. DESCRIPTION OF BENCHMARKS

Benchmark	Category	Description
adpcm (encode)	telecomm	takes 16-bit linear Pulse Code Modulation samples and converts them to 4-bit samples
dijkstra	network	calculates the shortest path between every pair of nodes in an adjacency matrix (representing a large graph) using Dijkstra's algorithm
qsort	automotive & industrial control	sorts a large array of strings into ascending order using the well known quick sort algorithm
sha	security	produces a 160-bit message digest for a given input
stringsearch	office	searches for given words in phrases using a case insensitive comparison algorithm
typeset	consumer devices	captures the processing required to typeset an HTML document

We then compiled all the selected benchmarks on IA-32 platform and evaluate the emulation performance of these benchmarks on TransARM-IU. The last column of

Table IV list the number of the source instructions of each benchmark. As these entire source instructions were interpreted, the last column also reveals the numbers of the interpreted source instructions. The executable files with the input data files were then migrated to the ARM platform. All benchmarks are successfully emulated by TransARM-IU on the two ARM platforms. This demonstrates the correctness of our emulation method.

TABLE III. NATIVE EXECUTION TIME AND INTERPRETED INSTTUCTIONS OF EACH OF THE BENCHAMARKS

Benchmark	Native execution time on ARM platform 1	Native execution time on ARM platform 2	Source instruction No.
adpcm.enc	0.92 s	1.47 s	38458788
dijkstra	0.65 s	0.99 s	49534305
qsort	0.52 s	0.80 s	43890779
sha	0.22 s	0.35 s	12904042
stringsearch	0.02 s	0.03 s	4345763
typeset	0.41 s	0.62 s	29354972

(s: second)

Comparing to the native execution on ARM, emulation definitely introduces extra overhead which causes performance degradation. Typically, the emulation speed of a software interpreter is 10X to 100X slower than native execution [1]. Since performance is a crucial issue an emulation system. The emulation time of the benchmarks on TransARM-IU which adopts Pcache-based hybrid threaded interpretation method, is on average 35 times as much as their native execution time on the 600MHz ARM platform and 44 times on the other ARM platform. Fig. 7 shows the ratio of the emulation time to the native execution time for each benchmark. For evaluation of TransARM-IU on the ARM platform 1, the reference is the native execution on platform 1. Similarly, for evaluation of TransARM-IU on the ARM platform 2, the reference is the native execution on platform 2. TransARM-IU causes much larger performance degradation on the ARM platform 2 than on the ARM platform 1. The main reason is that platform 2 does not have a L2 cache. Cache misses have a big impact on the emulation performance.

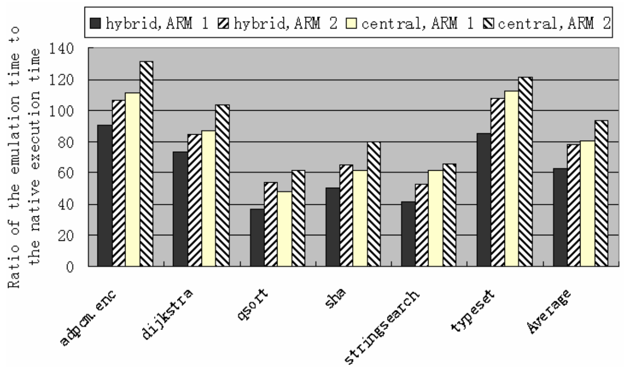


Figure 5. Performance evaluation of TransARM-IU. "hybrid" means the TransARM-IU system adopts the Pcache-based hybrid threaded interpertation method and "central" presents the conventional central looped interpretaion method.

We also evaluated the performance of TransARM-IU which uses the conventional centralized interpretation method with Pcache used. Experimental results indicate that the emulation time is on average 63 times of the native execution time on ARM platform 1 and 78 times on ARM platform 2. This indicates that the Pcache-based hybrid threaded interpretation method can dramatically reduce the emulation overhead.

A well-known interpretation-based emulation system was utilized for reference. We derived the ISA emulation part from Bochs 2.4.1 where the source ISA and the target ISA are the same IA-32. We measured the emulation time of the 6 MiBench benchmarks on Bochs running on the IA-32 platform. We also measured the native execution time of the benchmarks on the same IA-32 platform. The average ratio of the emulation time to the native execution time is 87, as shown in Table V. Because the source ISA and target ISA are the same, the emulation process of Bochs is much simpler than a cross-platform emulator. This indicates that TransARM-IU, especially when adopting Pcache-based hybrid interpretation, is competitive to Bochs.

TABLE IV. RATIO OF THE EMULATION TIME ON BOCHS TO THE NATIVE EXECUTION TIME

Benchmark	Ratio	Benchmark	Ratio
adpcm.enc	108	sha	81
dijkstra	96	stringsearch	48
qsort	73	typeset	115
Average	87		

In TransARM-IU, Pcache is defined as a software hash table, which consumes memory resource as it is actually stored in the main memory. A software Pcache with 1024 entries need at least 12KB memory space. Software Pcache can not only consumes the memory resource, but can also add considerable burden to the data cache of the underlying ARM processor. Moreover, the size of the data cache is limited. Sometimes, the processor has to access the Pcache from the main memory which consumes much more time than access from the data cache. A solution to this problem is to implement a hardware Pcache[6][7]. A hardware Pcache do not consume the resource of the data cache and is more faster. However, hardware Pcache has to modify the underlying hardware as special Pcache accessing instructions are needed [6][7]. This is easy to be achieved as most ARM platform use customized ARM processor IP cores.

VI. CONCLUSIONS AND FUTURE WORK

ISA emulation is a key aspect of a cross-platform virtual machine.TransARM-IU, an ISA emulator, has been developed to support user-level IA-32 applications on ARM-based systems. TransARM-IU adopts an ELF analyzer to resolve the executable file's format, thus to identify the required ISA architecture and other useful information for execution. The source architecture resource has been mapped to a software maintained context block with interpretation of each source instruction on the mapped architecture enabled.

As the conventional centralized interpretation method causes much emulation overhead, especially when interpreting the instructions of the CISC ISA, like IA-32. Aiming at this problem, a hybrid threaded interpretation method has been developed which appends a simple decode-and-dispatch routine, optimized for the common cases, to each instruction interpretation routine and use a centralized decode-and-dispatch routine for more complex cases. We selected six benchmarks from MiBench for performance evaluation of TransARM-IU. The experimental environments are two real ARM platforms with different configuration settings. All the selected benchmarks which are pre-compiled on the IA-32 platform have been successfully emulated by TransARM-IU on both target platforms. This demonstrates the correctness of our work. The experimental results also indicate that the emulation can be dramatically accelerated by our novel hybrid interpretation method. TransARM-IU is competitive to other ISA emulators such as Bochs.

For future work, we will explore effective solutions to floating point instruction emulation and library calls emulation which are now addressed through indirect method. Our ultimate goal is a full system VM across IA-32 and ARM.

ACKNOWLEDGEMENT

This work is supported by China National Basic Research Program under grant No.2007CB310901 and National Nature Science of China under grant No.60804036.

REFERENCES

[1] M. Rosenblum, T. Garfinkel, “Virtual Machine Monitors: Current Technology and Future Trends,” Computer, Vol.38, No.5, pp39-47, 2005.

[2] W. Huang, J. Liu, A. Bulent, D. K. Panda, “A Case for High Performance Computing with Virtual Machines,” Proceedings of the 20th Annual International Conference on Supercomputing, New York, USA:ACM, pp125-134, 2006.

[3] J. E. Smith, R. Nair, Virtual Machines: Versatile Platforms for Sysetms and Processes, Beijing: Publishing House of Electronics Industry, 2006.

[4] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer's Manual, vol.1: Basic Architecture, 2009.

[5] H. Kim, J. E. Smith, “Hardware Support for Control Transfers in Code Caches,” Proceedings of the 36th International Symposuim on Microarchitecture, Washington, USA: IEEE CS, pp253-264, 2003.

[6] Wei Chen, Hongyi Lu, Li Shen, Zhiying Wang, Nong Xiao, “A Hardware Approach for Reducing Interpretation Overhead,” Proceedings of the 9th IEEE International Conference on Computer and Information Technology, Washington, USA: IEEE CS, pp98-103, 2009.

[7] Wei Chen, Li Shen, Hongyi Lu, Zhiying Wang, Nong Xiao, “A Light-Weight Code Cache Design for Dynamic Binary Translation,” Proceedings of the 15th International Conference on Parallel and Distributed Systems, Washington, USA: IEEE CS, 2009.

[8] J. C. Dehnert, et al., “The Transmeta Code MorphingTM Software: Using Speculation, Recovery, and Adaptive Retranslation to Address Real-Life Challenges,” Proceedings of 1st Annual IEEE/ACM International Symposium on Code Generation and Optimization, Washington, USA: IEEE CS, pp15-24, 2003.

[9] Bochs, http://sourceforge.net/projects/bochs/files/bochs/.

[10] D. Mihocka, S. Shwartsman, “Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure,” Proceedings of the 1st Workshop on Architectural and Microarchitectural Support for Binary Translation, http://bochs.sourceforge.net/Virtualization_Without_Hardware_Final.pdf.

[11] QEMU, http://www.qemu.org/download.html.

[12] F. Bellard, “QEMU, a fast and portable dynamic translator,” Proceedings of the USENIX 2005 Annual Technical Conference, Berkeley, CA, USA: USENIX Association, pp41-46, 2005.

[13] L. Baraz, et al., “IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems,” Proceedings of the 36th International Symposium on Microarchitecture, Washington, DC, USA: IEEE CS, pp191-204, 2003.

[14] R. J. Hookway, M. A. Herdeg, “Digital FX!32: Combinning Emulation and Binary Translation,” Digital Techncal Journal, Vol. 9, No. 1, pp3-12, 1997.

[15] K. Ebcioglu, E. R. Altman, “DAISY: Dynamic Compilation for 100% Architectural Compatibility,” Proceedings of 24th International Symposium on Computer Architecture, New York, USA: ACM, pp26-37, 1997.

[16] D. Ung, C. Cifuentes, “Machine-Adaptable Dynamic Binary Translation,” Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization, New York, USA: ACM, pp41-51, 2000.

[17] ARM Limited, ARM Architecture Reference Manual, http://www.arm.com/miscPDFs/14128.pdf.

[18] UNIX System Laboratories, Executable and Linkable Format, http://www.skyfree.org/linux/references/ELF_Format.pdf

[19] The Santa Cruz Operation Inc, SYSTEM V APPLICATION BINARY INTERFACE Edition 4.1, http://www.sco.com/developers/devspecs/gabi41.pdf.

[20] ARM Limited, Application Binary Interface for the ARM® Architecture (The Base Standard), http://infocenter.arm.com/help/topic/com.arm.doc.ihi0036b/IHI0036B_bsabi.pdf.

[21] P. Klint, “Interpretation Techniques,” Software-Practice and Experience, Vol. 11, pp963-973, 1981.

[22] K. Casey, M. A. Ertl, D. Gregg, “Optimizing indirect branch prediction accuracy in virtual machine interpreters,” ACM Transactions on Programming Languages and Systems, Vol.29, No.6, pp1-36, 2007.

[23] R. G. Matthew, et al., “MiBench: A free, commercially representative embedded benchmark suite,” Proceedings of the IEEE 4th Annual Workshop on Workload Characterization, 2001, http://www.eecs.umich.edu/mibench.

[24] MiBench, http://www.eecs.umich.edu/mibench/.

[25] Shiliang Hu, J. E. Smith, “Reducing Startup Time in Co-Designed Virtual Machines,” Proceedings of the 33rd Annual International Symposium on Computer Architecture, Washington, DC, USA: IEEE CS, pp277-288, 2006.

Wei Chen was born in China in 1982. She is a Ph.D. candidate in National University of Defense Technology, P. R China (NUDT). She obtained her B.Sc. and M.Sc. in computer science from NUDT in 2004 and 2006 respectively. Her research interests include computer architecture, binary translation, and computer virtualization.

Zhiying Wang was born in China in 1956. He is a Professor with the School of Computer, National University of Defense Technology, P. R China. He is a Senior Member of China Computer Federation, and a member of IEEE. He received a B.Sc., a M.Eng. and a Ph.D. from NUDT. His main research interests include high performance computing, computer architecture, asynchronous microprocessor, and information security.

Dan Chen was born in China, in 1973. He is a joint HEFCE Fellow with the University of Birmingham and the University of Warwick (United Kingdom). He is also a Professor with the Institute of Electrical Engineering, Yanshan University (China). He received a B.Sc. in applied physics from Wuhan University (China) and a M.Eng. in computer science from Huazhong University of Science and Technology (China). After that, he received another M.Eng. and a Ph.D. from Nanyang Technological University (Singapore). His research interests include computer-based modelling and simulation, high performance computing.