Automatic Validation for Static Binary Translation

Jiunn-Yeu Chen  
National Chiao Tung University  
jiunnyeu@cs.nctu.edu.tw

Bor-Yeh Shen  
National Chiao Tung University  
byshen@cs.nctu.edu.tw

Yuan-Jia Li  
National Chiao Tung University  
yuanchia@cs.nctu.edu.tw

Wuu Yang  
National Chiao Tung University  
wuyang@cs.nctu.edu.tw

Wei Chung Hsu  
National Chiao Tung University  
hsu@cs.nctu.edu.tw

Abstract

Binary translation is an important technique for porting programs as it allows applications for one platform to execute on another. The technique is widely used in virtual machines and emulators. However, binary translation is challenging because many delicate details, such as calling conventions and system calls, must be handled carefully to generate correct translated code. Identifying a mistranslated instruction in a program is difficult, especially when the application program is large. Therefore, it is necessary to develop an automatic tool to uncover problems incurred during translation. We have developed a new validation mechanism for static binary translation, which checks the correctness of emulated architecture state (the state of the emulated architecture) during program execution. We have also proposed additional optimizations to speed up the automatic validation process.

General Terms  Reliability, Validation.

Keywords  binary translation, validation, ARM, QEMU, architecture state

1. Introduction

Binary translation [1, pp. 49-52] is an important technique for porting programs as it allows applications for one (software/hardware) platform to execute on another. The technique is widely used in virtual machines and emulators, such as VMware Workstation [2], Microsoft Hyper-V [3], and QEMU [4]. There are two broad categories of binary translation: static binary translation, which translates code offline, and dynamic binary translation, which translates code at run time. UQBT [5] and FX32 [6] are considered static binary translators while Aries [7] and QEMU are dynamic.

Developing binary translators is challenging because many issues must be handled carefully, such as calling conventions, system calls, and the code discovery and code location problems [1, pp. 52-55]. The translated program would not work if any translation process is not handled correctly. In order to validate the correctness of a translated program, machine states are usually checked after each instruction is executed manually. The approach is both time consuming and error-prone. An automatic tool to conduct such validation steps is absolutely called for.

Traditional validation methods fall into two categories. The first approach compares the execution results of the original and the translated programs. The concept is similar to black-box testing of software engineering. This strategy is simple but unsuitable since it is difficult to accurately compare the results since there are many kinds of results, such as creating a new file or initiating a system operation. Moreover, this strategy would yield little information to assist debugging.

The second approach performs validation based on the control flow of the programs. This strategy is widely applied in the validation of optimizing compilers [8], [9]. Because the target addresses of indirect branches cannot be resolved completely at static time, it is difficult to build an accurate control flow graph from binary code. This constraint makes this strategy difficult to apply in the automatic validation of static binary translation. We have developed a new process to automatically validate statically translated programs. This mechanism is based on the first validation method but it compares the architecture states instead of the program outputs. In our mechanism, the emulated architecture states are checked after each translated instruction is executed. Instrumentation code, called a checkpoint, is inserted after each translated instruction. In our experiment, we statically translate ARM executables to equivalent x86 executable binaries. Instrumented code must be inserted in both the source ARM binary and the translated x86 binary at exactly the same places. The original ARM programs and the translated x86 programs run side by side. When the instrumentation code is executed, the emulated architecture states, including the emulated CPU registers and condition flags, are compared against the corresponding architecture states in the original program. On the surface, this strategy seems simple and intuitive, but there are many challenging issues to overcome. For example, one such challenge is that the native running program and the translated program may have inconsistent placements of their code and data in the virtual address space. Different placements will result in different values in the architecture states. This makes comparisons difficult to carry out.

In this paper, we report our experience with an automatic validation tool which can uncover mistranslated instructions for stati-
cally translated code. The tool uses QEMU to generate the runtime architecture states of the original program and then compares it against the corresponding emulated architecture states of the translated program to ensure the translated program behaves exactly as the original program. We also come up with several optimization techniques that significantly speed up the validation process.

The remainder of this paper is structured as follows. Section 2 sketches the binary translator LLBT and the emulator QEMU. Section 3 describes the challenges of automatic validation. Section 4 illustrates the implementation details and shows how to overcome the problems of different memory address assignment, and following by the details of the method to speed up the validation process. Section 5 discusses the experimental results and section 6 summaries and concludes.

2. Background

In this research, we attempt to validate the static binary translator LLBT (LLvm-based Binary Translator) [10]. We use LLBT to translate an ARM binary into an x86 binary and compare the architecture states during the execution of the ARM and x86 binaries. We also need QEMU to emulate the ARM binary on our hardware platform (which is a x86 architecture). Two reasons let us to use QEMU to emulate the ARM binary instead of directly execute it on a real ARM machine. The first reason is performance. In this research, the two architecture states should be compared. It needs network communication to send information for validating. The overhead of network communication is more than communication between two processes in the same machine. Moreover, it is not for sure that the performance of the real ARM machine is better than that of emulation on an x86 machine. The second reason is decreasing the developing effort, we use QEMU to help us deal with the placement of data in the virtual memory rather than duplicate the function of memory management. The details of implementation will be described in section 4. Thus, it is recommended to have basic knowledge about QEMU and LLBT. The overview of QEMU and LLBT are given in the remainder of this section.

2.1 Overview of QEMU

QEMU is a generic virtual machine created by Fabrice Bellard. Two modes of emulation are supported by QEMU: process virtual machines and system virtual machines. A process virtual machine can make programs executable on platforms of different Instruction Set Architectures (ISAs). Thanks to the technique of binary translation, process virtual machines for different ISAs could be implemented on the same hardware platform. For example, an ARM program can be executed on an x86 architecture. On the other hand, a system virtual machine makes it possible to emulate a whole system including I/O devices, memory space, etc. Another operating system or environment, called guest, can be built on top of the current system, called host.

In this research, we use the process mode of QEMU. When a original program is emulated by QEMU in the process mode, QEMU first loads the original code, sets the base address of the emulated heap, and initializes the emulated CPU registers in memory. Then it allocates memory space for the code cache and the emulated stack. Finally, QEMU translates the original code and then executes the translated code. Note that the translated code is stored in the code cache temporarily, the emulated architecture state is represented by the emulated CPU registers, and the emulated heap and emulated stack are the heap and stack used by the translated code, respectively. The left side of figure 1 shows the overall memory layout of QEMU emulating an ARM program.

2.2 Overview of LLBT

LLBT is a static binary translator based on LLVM (Low Level Virtual Machine) [12]. Unlike QEMU, LLBT translates code offline, but it does not offer the function of system virtualization. LLBT translates ARM machine code into LLVM IRs (an intermediate representation in LLVM), and then uses the LLVM backend to generate target machine code. LLBT takes an ARM binary program as input and it outputs a new binary program for another ISA (Instruction Set Architecture), such as ARM, x86 and MIPS. The right side of figure 1 shows the overall memory layout of a translated program in the x86 platform. It is important to note that the emulated architecture state is maintained by the translated x86 program and the ARM stack is allocated in the stack of the process by default rather than in the heap. In general, the performance of a program translated by a static binary translator would be better than that by a dynamic binary translator because the translation time can be ignored.

3. Challenges of Automatic Validation

When an ARM program is emulated by QEMU, the process’s virtual address space is managed by the QEMU memory management instead of the host operating system. In other words, QEMU not only emulates the processor but also emulates the memory. On the other hand, whenever executing a translated program (translated by LLBT), the emulated stack and heap could be placed at any location allocated by the host operating system. Figure 1 shows the virtual memory layouts in QEMU (left) and of the translated x86 program (right), respectively. In QEMU, both the ARM stack and the ARM heap are emulated in the heap; while in the translated x86 program, the ARM heap is emulated in the heap but the ARM stack is emulated in the stack section. This causes the differences in the memory layout of two programs.

When we validate the behaviours of the original ARM program (which is emulated by QEMU) and of the translated x86 program in Figure 1, we compare their architecture states at various steps during program execution. The architecture state includes the contents of the registers and the four condition flags (NZCV). The contents of a register could be either a value or an address. When
the contents of a register, say R1, is a value, then R1 in the original ARM program and R1 in the translated x86 program should be identical. When the contents of a register, say R2, is an address, then R2 in the original ARM program and R2 in the translated x86 program could have different contents if memory allocation method is different in the two programs.

ARM is a register-based load-and-store architecture. All memory operations are performed by load or store instructions. Before an instruction accesses the memory, the address of the memory cell must be saved in the registers. For example, when a program operates on the process stack, the stack-pointer register SP holds the address of stack’s top cell. During validation, the contents of the stack-pointer registers in the emulated ARM program and in the translated x86 program registers may be different even if they actually point to the same stack-top cell. Similar situations may happen to other registers.

In comparing architecture states, there are two approaches: (1) We may allow different memory allocations and adjust the addresses in registers accordingly. (2) If we always allocate the ARM stack and heap at the same virtual addresses, all corresponding registers in the emulated ARM program and in the translated x86 program must always hold the same contents, be it a value or an address. In the former solution, adjusting the addresses in the architecture states is cumbersome because it is difficult to track the contents of registers. Therefore, we adopt the later approach.

In comparing architecture states, there are two approaches: (1) We may allow different memory allocations and adjust the addresses in registers accordingly. (2) If we always allocate the ARM stack and heap at the same virtual addresses, all corresponding registers in the emulated ARM program and in the translated x86 program must always hold the same contents, be it a value or an address. In the former solution, adjusting the addresses in the architecture states is cumbersome because it is difficult to track the contents of registers. Therefore, we adopt the later approach.

4. Design and Implementation

This section illustrates the various methods that are used to ensure that memory allocation in the translated x86 program is exactly the same as that in the emulated ARM program. Figure 2 shows the overall structure of the validator. Before program execution, the translated x86 program is generated from the original ARM program by LLBT. LLBT also produces a register-defining list from the generated LLVM intermediate form (IR). For each ARM instruction, the register-defining list contains the registers that are modified by that ARM instruction. In order to validate the LLVM IR generated by LLBT, we use another register-defining analyzer, which also produces a register-defining list from the original ARM binary code. The two register-defining lists should be identical otherwise there are problems in LLBT.

4.1 Allocating identical virtual memory

In order to simplify the comparison of the architecture states, all variables and dynamic memory in the translated x86 program are allocated in the same virtual addresses as those in the emulated ARM program.

When QEMU emulates the original ARM binary, it first allocates a block of memory that will serve as the stack of the ARM binary. When the translated x86 binary executes, it also needs to allocate a block of memory that will serve as the stack of the ARM binary. Moreover, when a program is executed, the program loader will push the addresses of the system arguments and environment variables into the stack. Figure 3 shows the initial contents of the ARM stack. The ARM stack in the translated x86 binary should be located at the same virtual memory address and should have the same content as the ARM stack in the emulated ARM binary.

When QEMU forks the first new process, the new process automatically copies the entire virtual memory from the original QEMU process. Then the translated x86 code is loaded as a shared library and starts execution. The translated x86 binary can use the ARM stack directly.

The ARM heap is handled similar to the ARM stack. The ARM heap in the translated x86 binary is located at exactly the same virtual memory address as the ARM heap in the emulated ARM binary because the new process is forked from the original QEMU process. Memory in the heap is allocated piece by piece. Since memory allocation is done with system calls, such as brk or

![Figure 2: The overall structure of the validation consists of the offline phase and the runtime phase. The translated x86 program, the register-defining list, and the results of coarse-instruction analysis are prepared during the offline phase. At runtime, three processes cooperate to perform validation.

![Figure 3: The initial layout of the ARM stack.](image)
**Figure 4.** An example of coarse instructions. The colored registers in the right are the registers defined by the coarse instruction.

In order to make the two ARM heaps (one in emulated ARM code and one in the translated x86 code) identical, it is necessary to modify all the system calls which are related to memory allocation. But we cannot change the kernel code when we modify the system calls because other programs unrelated to validation would be influenced as well. Therefore, a user-level’s solution should be adopted.

Before explaining the solution, we describe the system-call mechanism in ARM architecture. There are two steps in a system call in an ARM program:

(a) Store a system call number in the register R7.
(b) Raise software interrupt by swi (or svc) instruction.

The brk system call is implemented as follow:

\[
\text{mov R7, #45}
\]
\[
\text{swi 0x00000000}
\]

The first mov instruction stores the constant 45 into the register R7. 45 is the system-call number of the brk system call. The second instruction swi(or svc) raises a software interrupt. In handling the software interrupt, the OS will switch to the kernel mode, allocate the required memory, and store the address of the allocated memory into the register R0.

In our solution, LLBT translates the svc instruction into a function call to a wrapper which performs the system call according to the value of R7. The wrapper uses mmap to allocate a block of memory at a specific address. In fact, the QEMU memory management is implemented in a similar way.

Although adapting the system-call wrapper can avoid modifying the kernel code, it still needs a memory management utility that behaves exactly the same as the QEMU memory management. This is achieved automatically when a new process is forked from the original QEMU process. The QEMU memory management automatically sits on the virtual memory of the new process. Because the translated program is loaded into the process (which is forked from QEMU) as a shared library, the code sections in QEMU and translated x86 code exist in the same memory space. It is possible for the translated x86 code to use QEMU memory management. So, we initialize a function pointer which points to the code of QEMU memory management and then the system-call wrapper of the translated x86 program can adopt the memory management by calling through the function pointer.

Therefore, the strategy, executing the translated x86 code by loading it into the new process, which forked from QEMU, as a library, is a good solution with less development effort to makes the ARM stack and ARM heap identical.

### 4.2 Performance of validation

The performance of the straightforward validation is a serious challenge. In the straightforward validation, the instrumentation code is inserted immediately after every translated ARM instruction. This means that the instrumentation code is executed once for every ARM instruction that is executed. The instrumentation code will collect and deliver the architecture state to the validator. The cost of the instrumentation code is much more than that of a single ARM instruction. In our experiment, the execution time with validation was 27 times larger than without validation.

The instrumentation code needs to deliver the architecture state to the validation framework. We use the shared memory for the delivery, which is the fastest IPC strategy because no memory copy between user space and kernel space needed.

We propose two methods to reduce the overhead of validation.

Optimizations In this section, we proposed two methods to speed up the validation. First we use coarse instruction, which reduces the number of instrumentation code and indicates the mistranslated instruction without losing the precision. The second is quick validation, which attempts to validate each ARM instruction only once even if that instruction is executed several times.

### 4.3 Coarse instruction

A coarse instruction is a sequence of instructions. It consists of one or more adjacent machine instructions in a program. We called the grouping method as coarse-instruction analysis. Figure 4 shows an example coarse instruction. The first instruction add r5, sp, #40 defines the register R5. The remaining instructions define \{R6\}, \{R1, R2, R3\}, \{R0\}, \{IP\}, respectively.

A coarse instruction is a longest sequence of consecutive instructions that satisfy the following conditions: (1) All the instructions will be executed once the first instruction is executed; and (2) No two instructions define the same registers; and (3) Every machine instruction belongs to exactly one coarse instruction. The first condition implies only the last machine instruction of coarse instructions can be a jump instruction. The second condition implies every register is modified at most one time in a coarse instruction. We add the instrumentation code after each coarse instruction, instead of after each machine instruction. When any register goes wrong during validation, we can still trace the offending machine instruction. They reduce the instrumentation code by 80% in the example in Figure 4.

There might be more than one way to partition a program into coarse instructions. We will show a simple algorithm that can identify coarse instructions later.

A coarse instruction can be represented as \(C = (I_1, I_2, \ldots, I_n)\) where each \(I_i\) is a machine instruction in the program. The registers defined by \(C\) is \(D = (D_1, D_2, \ldots, D_n)\) where \(D_i\) is the set of registers defined by \(I_i\). Note that \(D_i \cap D_j = \emptyset\), where \(i \neq j\).

We may analyze a program and identify the coarse instructions. The instrumentation code is added after each coarse instruction. As shown in figure 2, the register-defining analysis and coarse-instruction analysis are performed offline.

A coarse instruction is not a basic block for two reasons: A basic block has exactly one entry point and one exit point. In contrast, a coarse instruction can have multiple entry points and one exit point. It is allowed to jump into the middle of a coarse instruction. This definition could potentially enlarge a coarse instruction. Hence there are fewer coarse instructions (and less instrumentation overhead). Different instructions in the same basic block can define the same registers. This is not allowed in a coarse instruction because we need to trace the offending instruction in a coarse instruction once a register goes wrong.
In order to group coarse instructions, we analyze the instructions to collect the registers and condition flags defined by each instruction and to find all jump instructions. The analysis is performed and the instrumentation code are inserted during translation. A condition flag is treated like a register for our purpose. All instructions that may change the straight line control flow are considered as jump instructions.

In the ARM architecture, there are three types of jump instructions:

1. The branch instructions such as `b` (branch), which accepts a signed offset as the relative address of the next instruction, and `bl` (branch and link), which saves the program counter PC in the linkage register LR before jumping to the target address. These instructions can be identified by opcode of the instructions.

2. The instructions that define the program counter PC such as `mov PC, LR`. These instructions can be identified easily.

3. In the ARM architecture, some of instructions such as `cmp` may set the condition flags (NZCV) of the processor. The N flag is set when the result is negative; the Z flag is set when the result is zero; the C flag is set when the result of an unsigned operation overflows; and V flag is set when the result of a signed operation overflows. Some ARM instructions could be conditionally executed. We called these instructions are predicated instructions. A predicated instruction is considered as a conditional branch. However, in order to make a coarse instruction as large as possible, a sequence of predicated instructions with the same predicate will be grouped into a single coarse instruction. For example, the following sequence of instructions is divided into 5 coarse instructions:

```
add R1, R2, R3
------------------
adlt R0, R1, #1
------------------
adreq R2, R2, R1
adreq R4, R4, #1
------------------
adreq R5, R5, R4
adreq R6, R6, R4
add R7, R7, R4
add R8, R8, R4
add R9, R9, R4
------------------
adreq R3, R1, R2
```

Certain instructions may change the condition flag, for example, in `addeqs`, the suffix `s` means the condition flag will be modified according the result of the add operation. A coarse instruction will stop immediately after such an instruction. Certain instructions do not carry a flag. These instructions will always be executed. A sequence of non-predicated instructions could be appended to the end of the previous predicated coarse instruction.

The algorithm for coarse-instruction analysis is shown in figure 5. This algorithm will group the coarse instructions by marking the last machine instruction of each coarse instruction as a checkpoint. Lines 9 to 11 identify the first two kinds of jump instructions. Lines 12 to 14 ensure that no register is defined more than once in a coarse instruction. Lines from 15 to 22 process predicated instructions. The instrumentation code for validation is inserted after the instructions marked as checkpoints.

We use an example to demonstrate the coarse-instruction analysis algorithm.

```
i = ary[1]; j = ary[2];
k = ary[3]; m = ary[4];
if ( i == 0 ) {
i += 100; j += 200;
} else {
k += 200; m += 100;
}
i += k;
i = m;
print(i);
return;
```

Initially, the value of variable acc_regs is NULL. When the instruction in line 1 is analyzed, the value of variable acc_regs becomes {R1}. Because the defined registers are different from

```
add R1, R2, R3
------------------
adlt R0, R1, #1
------------------
adreq R2, R2, R1
adreq R4, R4, #1
------------------
adreq R5, R5, R4
adreq R6, R6, R4
add R7, R7, R4
add R8, R8, R4
add R9, R9, R4
------------------
adreq R3, R1, R2
```
line 1 to line 5, the value of acc_regs is \{R1, R2, R3, R4, CPSR\} when the instruction in line 5 is analyzed. When the instruction in line 6 is analyzed, because its condition code is different from the previous instruction, the previous instruction (line 5) should be marked as a checkpoint and the variable acc_regs is set to \{R1\}. The instruction in line 8 is similar to the one in line 6 so the instruction in line 7 should be marked as a checkpoint. Note that when line 10 is analyzed, line 9 is not a checkpoint because the condition code in line 10 is AL. However, line 11 should be a checkpoint because the defined register is the same as the one in line 10. Line 12 and line 13 both would change the program flow so they are marked as checkpoints. In the example, the checkpoints are in line 5, line 7, line 10, line 12 and line 13.

### 4.4 Quick validation

The coarse-instruction technique reduces the number of checkpoints to one-third of the number of instruction. However, the resulting instrumented program is still 9 times slower than the original program. Note that certain code in a program may be repeatedly executed. The code will also be validated repeatedly. In order to speed up the validation, we adopt the code-coverage technique in our validator. In short, each instruction will be validated only when it is executed for the first time. After a piece of instrumentation code is executed, it is turned off so that it will not be executed for the second time.

Code coverage is a measure of the degree that the source code of a program has been tested. It has been used in white-box software testing widely [13], [14]. In practice, many coverage criteria can be adopted, in a trade-off between the performance and the accuracy in testing.

- **function coverage**: There is instrumentation code in each function to indicate whether it has been tested.
- **statement coverage**: There is instrumentation code in each block of code compiled from a statement in source code to indicate whether it has been tested.
- **decision coverage**: There is instrumentation code in each decision (or edge) to indicate whether it has been tested.
- **instruction coverage**: There is instrumentation code in each instruction to indicate whether it has been tested.

In order to offer the instruction-level coverage, we adopt the instruction coverage criterion.

According to the 80/20 rule, 80 percent of the execution time is spent on 20 percent of the executed code. For long-running programs, most of the other 20 percent of executed code is contained in loops. If the code in loops is validated at most once, the remaining time spent in the instrumentation code is extremely lesser than that in the normal code. Hence, the performance of validation can be improved significantly. We use a short example to illustrate the quick validation.

```plaintext
; add R4, R4, #1
L_00000062: ; no need to check condition code
  br label %L_00000062_2
L_00000062_2:
  %tmp_op1_39 = load i32 * %ARM_r4
  %tmp_result_int_80 = add i32 %tmp_op1_39, 1
  store i32 %tmp_result_int_80, i32* %ARM_r4
  call @helper_trace(...)
  br label %L_00000063
```

The above code is the LLVM IR which are generated by LLBT from the ARM instruction `add R4, R4, #1`. The line `call @helper_trace(...)` is the instrumentation code which calls an external function for validation. The last line `br label %L_00000063` transfers control to the next instruction. When the code from `br label %L_00000062_2` to store i32 `%tmp_result_int_80, i32* %ARM_r4` is executed, the function `helper_trace` should be executed and it could record that the code has been tested.

According to our experimental results, total execution time (including normal program execution and validation) in this validate-once strategy is only 1.4 to 1.9 times the execution time of the original program. This validate-once strategy is faster but may produce less accurate results.

In the quick validation mode, all instrumentation code is executed at most one time. We use dynamic binary instrumentation in QEMU (a dynamic binary translator) and self-modifying code (SMC) [15] for the programs translated by LLBT (a static binary translator) to remove the instrumentation code dynamically. Many research focused on the similar issues. Mustafa [16] used dynamic code patching to achieve dynamic instrumentation for code-coverage testing. Naveen [17] proposed an efficient way to do program monitoring and profiling. It is available to build customized program analysis tools with dynamic instrumentation by PIN [18]. Most research used dynamic code patching or dynamic binary instrumentation to add or remove instrumentation. However, in our research, the instrumentation code is added during binary translation and would be removed later.

Figure 6 illustrates the process of removing dynamic instrumentation code. In QEMU (shown as Figure 6a), the basic unit of the translated code is a translation block (TB). When an ARM instruction is executed, QEMU would search for the corresponding TB first. If the TB is found (this implies the instruction has been translated) and is valid, the TB is executed; otherwise, QEMU translates the instructions, generates a TB, and then executes it. In our implementation, each TB would contain one or more pieces of instrumentation code. After a TB is executed, it will be marked as invalid. When the TB is encountered the next time, QEMU would check whether the TB is valid. If the TB is invalid, the TB will be generated again; however, instrumentation code will not be included this time.

On the other hand, for x86 code that is generated by LLBT (shown as Figure 6b), we cannot use the validate/invalidate mechanism in QEMU. Thus, a strategy based on self-modifying code is adopted. The steps are shown as follows: (1) When an ARM instruction is executed, the corresponding instrumentation code will be executed. (2) Because the instrumentation code is essentially a function call, the caller will obtain the function return value, which is the address of next instruction. Before the caller...
5. Experimental evaluation

Our validator operates in five modes:
1. **Full mode**: Validate after each ARM instruction.
2. **Coarse mode**: Validate after each coarse instruction.
3. **Branch mode**: Validate only when the branch occurs.
4. **Full quick mode**: Validate after each instruction is executed the first time.
5. **Coarse quick mode**: Validate after each coarse instruction is executed the first time.

Note that, the first two modes are accurate validation which guarantee all the inconsistent states would be found. The remaining three modes are less accurate.

Our validator is a good debugging tool. Several hidden bugs in the well-developed LLBT were found with our validator. We also conducted two experiments on the validator. The first experiment showed that coarse instructions can significantly reduce the number of checkpoints. The second experiment showed the execution time when validating the translated programs in the five modes.

The experimental environment is an x86 machine equipped with four 12-core AMD Opteron 6172 processors, and 45 GB memory.

The benchmark is EEMBC 1.1 [19] which is compiled as ARM statically linked binaries by arm-eabi-gcc 4.4.6 and linked with uClibc library 0.9.30.2 [20]. The version of QEMU underlying the current validator is 0.14.0.

5.1 Bugs in LLBT discovered by the validator

The validator discovered three bugs in the 4-year-old LLBT before writing this paper. The three bugs were fixed and listed below.

- **Load and store double words**: This bug was found when validating the math program which is a test case for the math library. In the ARM architecture, two categories of base-register addressing mode can be used in load and store operations. (1) Pre-indexed addressing mode. Consider the two instructions: *ldr R0, [R1, #4]* and *ldr R0, [R1, #4]!*. In both instructions, the value in the word whose address is *R1 + 4* would be loaded into the register *R0*, but in the second instruction, the notation *!* means to increment the base register *R1* by 4, that is *R1 + 4*. The second instruction is equivalent to two instructions: *ldr R0, [R1, #4]* and then *add R1, R1, #4*. (2) Post-indexed addressing mode. For example, *ldr R0, [R1]*. *#4* loads the value in the word whose address is *R1* to the register *R0*, and then increment the value of register *R1* by 4.

This bug occurs in the instruction *ldr R0, [R1]* (load double words) and *strd* (store double words) on pre-indexed addressing mode with updating base register (1). When updating a double word data, two registers would be used to store the value in memory address *Addr* and *Addr + 4*. The cause of the bug is that the value of base register is updated to *Addr + 4* rather than *Addr*. For example, after executing the instruction *strd R0, [R6, #4]*, the value of *R0* and *R6* are stored into *mem[R6 + 4]* and *mem[R6 + 8]*, and the value of *R6* should be updated to *R6 + 4*. In the mistranslated x86 code, the value of *R6* would be updated to *R6 + 8*.

**BLX.** This bug was found when validating the benchmark djpeg in EEMBC’s consumer test suite. It is a bug in translating *blx* instruction and occurs when the operand is the register *FP* (*blx FP*). LLBT would regard the operand *FP* as an immediate value when translating. The misunderstanding makes the instruction *blx FP* be translated incorrect.

**Set condition flags.** This bug was found when validating the cast program which is a test case for casting floating points to integer numbers. The value of the carry flag (*C*) was wrong after executing the *SBC* (subtract with carry) instruction. The *C* flag should be set when a carry occurs in the add operations or “no borrow occurs” in subtract operations, but the code translated from the *SBC* instruction sets the *C* flag when “a borrow occurs”. However, this bug was solved by using LLVM overflow intrinsic, which is intrinsic for some arithmetic with overflow operations provided by LLVM, instead of the old one to update the condition flags.

5.2 The number of times instrumentation code is executed

Figure 7 shows the number of times instrumentation code is executed in the coarse mode and branch mode relative to that in the full mode. The result of the data shows the consequent of coarse instructions in coarse mode. In coarse mode, the ratio is lower for programs with fewer conditional and branch instructions. In our experiment, the ratio is between 22% and 52%. The average ratio is 36%. In other words, it can eliminate 64% cost in executing instrumentation code. In branch mode, the average ratio is 21%. 

![Figure 7](image-url)
The ratios in full quick mode and coarse quick mode are too small, so they are not shown in our experiment.

5.3 Execution time

Figure 8 (a) and (b) show the execution time for EEMBC applications in the five modes. All the execution time are normalized by adjusting the execution time of the original benchmarks emulated by QEMU (i.e., ones without the instrumentation code) as 1. From figure 8a, we can see the execution of a naive implementation of the instrumentation can be 59 times that of the original program.

In figure 8a, two observations are listed.

• The execution time is between 10 and 59 in the full mode. The reason for the large variation is the differences of program behaviours. For a CPU-intensive program, the execution time of the instrumentation code is huge relative to the CPU-computation operations. For a memory-intensive process, the execution time of the instrumentation code is relatively small.

• The overhead of validation is very huge so that the execution time decrease significantly when we reduce the number of times instrumentation code is executed. In the figure 7, it shows the executed instrumentation code in coarse mode is approximately one-third, and the figure 8a also shows the execution time is approximately one-third compared to full mode in average.

On the average, the execution time is 26.7 in the full mode and 9.5 in the coarse mode. In other words, the performance of validation in the coarse mode is about 3 times better than in the full mode. The fastest validation is in the branch mode, which has an execution time in 5.3. However, the branch mode is less accurate in that it does not guarantee to identify the mistranslated instructions.

The experimental results of the full quick mode and the coarse quick mode are shown in figure 8b. As shown in the figure, the
execution time of the validation with the coarse-instruction mechanism is smaller than that of the one without coarse-instruction mechanism. The validation time is 0.3 for the coarse quick mode and 0.8 for the full quick mode on average.

6. Conclusion
We developed a validator for binary translation which can indicate the mistranslated instruction if it discovers the program is translated incorrect. Furthermore, in order to speed up validation, we propose the coarse instructions, which reduce the number of instrumentation code. The results also show the number of times the instrumentation code is executed in the coarse mode is only 36% of that in the full mode. Finally, the code-coverage technique enables the quick validation. The validation time is only 30% of the normal execution (i.e., without validation).

Acknowledgments
The work reported in this paper is partially supported by National Science Council, Taiwan, Republic of China, under grants NSC 100-2218-E-009-010-MY3 and NSC 100-2218-E-009-009-MY3.

References