國立交通大學

資訊工程與科學研究所

碩 士 論 文

一個在二元轉譯中連結原生函式庫且可重定目標之方法

A Retargetable Approach for Linking Native Shared Libraries in Binary Translation

研 究 生：郭政錡

指導教授：楊武　教授

中 華 民 國　一百零一　年 七 月

一個在二元轉譯中連結原生函式庫且可重定目標之方法
A Retargetable Approach for Linking Native Shared Libraries in Binary Translation

研 究 生：郭政錡　　Student：Cheng-Chi Kuo

指導教授：楊武　　Advisor：Wuu Yang

國 立 交 通 大 學

資訊科學與工程研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2012

Hsinchu, Taiwan, Republic of China

中華民國一百零一年七月

一個在二元轉譯中連結原生函式庫且可重定目標之方法

學生：郭政錡　　　　　　　　　　指導教授：楊武 教授

國立交通大學資訊科學與工程研究所碩士班

摘　　要

二元轉譯是一種對移植應用程式到不同指令集常用的手段，動態連結共享函式庫也是經常被使用在以作業系統為主的系統上，但是在二元轉譯中，如何去處理動態連結的二元檔，尚未被廣泛討論。在二元轉譯系統中，有兩種方法可用來連結轉譯過後的可執行檔和其共享函式庫: (1) 我們可以使用轉譯過後的原始共享函式庫 (在原始平台的共享函式庫，稱作*已轉譯共享函式庫*) (2) 在我們的二元轉譯系統中，我們嘗試去連結目標平台的已經存在的共享函式庫 (稱作*原生共享函式庫*)。

連結原生共享函式庫的好處是可以增進執行效率和節省系統硬碟空間。然而，因為已轉譯可執行檔和原生共享函式庫有不同的 ABI，如何去處理它們之間的互動是一個很艱鉅的挑戰。

我們提出一個在二元轉譯中連結原生函式庫且可重定目標之方法，並實作在一個靜態二元轉譯系統 LLBT 中，它可轉譯

ARM 二元碼到 LLVM 中介碼，然後利用 LLVM 的後端產生已轉譯二元碼，這是一個無關目標平台的方法。

在我們的實驗中，已轉譯的 SPEC2006 程式並連結原生共享函式庫在 i7 的機器上跑得比其換成原本連結已轉譯共享函式庫還快，連結原生共享函式庫可以達到平均 1.18 的加速比。

A Retargetable Approach for Linking Native Shared Libraries in Binary Translation

Student：Cheng-Chi Kuo Advisor：Dr. Wuu Yang

Institute of Computer Science and Engineering
National Chiao Tung University

ABSTRACT

Binary translation is commonly used to migrate applications from one ISA to another and dynamic linking of shared libraries is widely used in OS-based systems. But how to handle dynamically linked binaries in a binary translation system has not been widely discussed. There are two ways in a binary translation system to link a translated executable with shared libraries: (1) We may use the shared libraries that are translated from the original shared libraries, which are on the source platform (which are called the translated shared libraries). (2) In our binary translation system, we attempted to link with the target shared libraries, which are compiled for and already on the target platform (which are called the native shared libraries).

Advantages of linking with native shared libraries include improving execution efficiency and saving system disk space. However, it is a significant challenge to handle the interactions between the translated executable and the native shared libraries since they may have different abstract binary interfaces (ABIs).

We present a retargetable approach for linking translated executables with native shared libraries. We implement it on a static binary translation system, called LLBT, which translates ARM binary to LLVM IR in a target-independent way and then generates the translated binary by the LLVM infrastructure. The generated IR has been available to be retargeted to different architectures, such as MIPS, without modification.

In our experiment, the translated programs in the SPEC2006 CINT benchmarks linked with native shared libraries ran faster than the ones linked with translated shared libraries on an i7 machine. Linking with native shared libraries can achieve an average speedup of 1.18.

誌　　　　謝

本論文能完成，首先衷心的感謝指導教授楊武教授，老師耐心的指導才能使得我能在碩士生涯中完成此論文，以及在課業上收穫良多。老師總是能夠以有趣幽默的態度來面對各種研究上的問題，這也使得研究生活一點都不苦悶。另外也感謝口試委員許慶賢教授、雍忠教授和游逸平教授，從教授們的建議可以看出他們對研究的態度是多麼嚴謹，以及在專業上可以用各種不同角度發現問題，這也使得此篇論文能更加完整。

此外，本論文的完成也得感謝程式語言與系統實驗室的柏皡學長與俊宇學長，在研究、課業以及學生生活上都給予很多很實用的建議。另外實驗室的原嘉、冠犟同學也都在課業上不吝的幫助。在兩年的碩士生涯中認識了很多人也接受了很多人的幫助，實在是非常感謝。

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.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Binary translation techniques have been actively researched and developed for the past two decades and have become a standard approach for migrating applications from one ISA to another. As there are many different architectures, it is desirable to develop a binary translation system that can be re-targeted to different architectures even though binary translation is highly dependent on the target machine architecture [15, 13]. In the past few years, dynamic binary translation (DBT) has been used more often than static binary translation, since there are challenging problems in static binary translation (SBT) such as the code location and code discovery problems. However, SBT could perform more aggressive optimizations [10] and have a shorter start-up time than DBT,

On most operating systems, there are two kinds of binary executables: statically linked ones and dynamically linked ones [20]. A statically linked executable is a complete binary executable that includes all the necessary code and data from libraries. Linking is done at static time. In contrast, a dynamically linked executable contains only a partial program and may require loading and linking with the libraries by a dynamic linker at run time. Although incurring run-time overhead, dynamic linking is still attractive in that (1) the dynamically linked executable has a smaller size and hence disk space is saved; (2) dynamically linked libraries can be shared among processes on a virtual memory system; and (3) it is unnecessary to recompile application programs when the shared libraries are upgraded. Nowadays, dynamic linking is widely used in OS-based systems.

Up to now, how to handle dynamically linked binaries in a binary translation system has not been widely discussed. A traditional binary translation system translates both source executables and source shared libraries. However, many common shared libraries, such as `libc`, are already compiled, optimized, and available on the target platform. It would be advantageous to use the shared libraries on the target platform instead of translating the source shared library. In this thesis, we discuss the details of linking translated executables with native shared libraries on an SBT system. The advantages of our approach are

1. Reduce translation time since only executables, but not shared libraries, need to be translated.

2. Avoid translation difficulties that are found often in special library

functions, for example, handcrafted assembly code.

In addition, the native shared libraries on the target platform are usually highly optimized for the target platform and outperform the translated shared libraries, which are taken from the source platform.

The abstract binary interface (ABI), in particular, the calling conventions, of the source and the target platforms may differ. This prevents most existing binary translators from linking with the target shared libraries. Our binary translator managed to resolve the ABI differences so that it can link the translated binary with native shared libraries on the target platform. We make use of LLVM IR [7], which is target-independent, and leave the target code generation to LLVM. In this way, our binary translator is re-targetable to many architectures as long as they are supported by LLVM.

We implemented our method on the LLVM-based static Binary Translator (LLBT)[21], which translates ARM binary into LLVM IR. Currently, our implementation can work for binaries compiled from C code. For the discussion in this thesis, we use the Executable and Linking Format (ELF) as the format for the dynamically linked binaries, ARM/Linux as the source platform and x86/Linux as the target platform but notice that our translator has been available to be retargeted to other architectures, such as MIPS.

The rest of this thesis is organized as follows. Section 2 briefs the related work. The LLBT overview is described in section 3. The issues related to binary translation and their implementation are discussed in section 4. The experiment results are shown in section 5.

Chapter 2

Related Work

In this section, we will brief some existing static binary translators. The focus is on the interoperability with native binaries and the retargetability to different architectures. We will also discuss binary tools related to LLVM IR.

2.1 Static Binary Translator

VAX Environment Software Translator (VEST)[22] is an SBT, which translates OpenVMS VAX images to OpenVMS Alpha images. The translated images can run just like native images on OpenVMS Alpha systems with the help of the translated images environment(TIE). Besides, VEST provides interoperability between native and translated images by *jacket routines* that are created automatically except certain cases that needs to be written by hand.

FX!32[19, 11] is an emulator/binary translator that migrates x86 Win32 applications to Windows NT/Alpha platforms. It combines emulation and static binary translation which uses the execution profiles, and provides interoperability with native Win32 API by jacketing native Win32 API and translated callback functions. Most jacket routines for native Win32 API are generated automatically at static time based on the API documentation and header files, and are embedded in FX!32 runtime library.

In contrast to VEST and FX!32, which can only handle the interoperation between specific source and target architectures (such as Windows NT on x86 and Alpha platforms in FX!32), LLBT uses a machine-independent IR, i.e., LLVM IR, to represent the translated code that can be retargeted to different platforms and can cooperate with native binaries.

UQBT[14, 16] is a retargetable SBT. It uses the high-level register-transfer language (HRTL) as the intermediate representation. HRTL can be translated into various forms, such as low-level C code, depending on the translation purposes. In addition, UQBT recovers functions from binary to high-level IR with explicit arguments and return values that are represented by four low-level types: integers, floating-point values including the sizes and signs, and pointers to data or to code. The number and types of arguments are obtained from an analysis of the source binary or are extracted from the header files. Once procedure calls are recovered, the translated code may use the native calling convention of the target platform rather than emulating the unclear source calling convention. Compared with UQBT, LLBT uses not only the target calling convention but also the target library code.

2.2 Decompilation to LLVM IR

SecondWrite[24] is a static binary rewriter that decompiles x86 binary into LLVM IR and then generates x86 binary by the LLVM backend. The advantage of LLVM IR as the intermediate representation is that SecondWrite may leverage the rich set of existing LLVM optimizations and transformations. Furthermore, SecondWrite recovers functions with symbols, arguments, and return values from binary and replaces register and memory locations by LLVM symbols. The high-level LLVM IR helps SecondWrite to do the security checks.

RevGen[12] is a tool that also statically converts x86 binary into LLVM IR. Its purpose is to analyze legacy binaries indirectly by analyzing the translated LLVM IR with existing LLVM tools.

Both SecondWrite and RevGen convert x86 binary into LLVM IR and generate the binary that is also x86. Our translator handles the issues particularly encountered in binary translation.

Chapter 3

LLBT Overview

This section will describe an overview of LLBT and the new components for linking with native shared libraries.

LLBT is an SBT system which uses several existing tools in order to improve retargetability and speed up development. The translation flow is shown in Figure 3.1. The ELF reader and the disassembler disassembles the source ELF binary. The translator translates the assembly code to LLVM IR. Following that, LLBT leverages the LLVM infrastructure to optimize LLVM IR and generate the target assembly. Finally, LLBT uses the target assembler and linker to generate a binary executable for the target platform.

Some parts of source (ARM) binary contain data that would be used in execution, such as the `.rodata` and `.data` sections. These sections are included in the target binary and will be directly used by the translated binary.

In the original binary, we may use or calculate a value that is actually an address that points to somewhere in the data sections (including `.data`, `.rodata`, etc.). The translated binary will calculate exactly the same value. In order to avoid the difficulties of mapping the addresses in the two binary executables, the data sections in the translated binary are placed at exactly the same location as the data sections in the original ARM binary (through a specification in the linker script).

The `.text` section contains data as well as instructions. The instructions are all translated into the corresponding instructions for the target platform. The data could be jump tables or are addressed through program-counter-relative (PC-relative) addressing mode. The jump tables are recovered and represented directly in the LLVM instructions while pc-relative addressing is in-lined in the LLVM instructions. Therefore, the `.text` section need not be kept in the translated binary.

There are a few sections that are used for dynamic linking, such as `.interp`, `.dynsym`, and `.hash` in the source binary. These sections are simply discarded. New information for dynamic linking on the target platform will be generated by the target linker.

3.1 Overview of the LLVM IR Generated by LLBT

The LLVM IR generated by LLBT (see Figure 3.2) consists of three kinds of LLVM functions:

1. All the instructions in the source binary are translated and put into an LLVM internal function called `unexported_text_section`. Each source instruction is translated into a sequence of LLVM instructions that follow an LLVM label named with the corresponding source address, such as `L_8308`.

2. The function `main` is the entry point of the translated executable. In the beginning of `main`, there are several instructions that perform allocation and initialization for emulating the source architecture state. Once the emulated architecture state is ready, it passes a pointer of the emulated architecture state to the `unexported_text_section` function, which starts executing the translated code.

3. For every exported function, which would be called by functions in the native library, LLBT creates a wrapper function. In Figure 3.2, `compare_wrapper` is the wrapper for the exported `compare` function. Similar to the above `main` function, this wrapper function will perform allocation and initialization and then invoke the above `unexported_text_section` function. The signature of the wrapper function is obtained from the

header file.

3.2 Registers and Stack

The registers and stack of the source architecture are emulated by LLVM local variables. We use the `alloca` instruction to allocate an `i32`-type local variable for each 32-bit register. Since only `load` and `store` operations are performed on these local variables, LLVM optimization may promote as many local variables into registers as possible, which makes the execution much faster. On the other hand, because these variables are local variables rather than global or static variables, the translated binary is reentrant.

3.3 Handling Indirect Branches

Unlike direct branches, whose branch targets are specified by constant value thus are known at static time. The branch target of an indirect branch is unknown only until it is about to be executed. For handling indirect branches, LLBT prepares an *address mapping table* that contains pairs of a source address and the LLVM label of the corresponding translated instruction. An indirect branch is translated to a branch to a piece of code that searches the address mapping table for a source address. If found, the corresponding LLVM label, instead of the source address, becomes the target of the indirect branch in the translated binary.

A naive address mapping table would contain one pair for *every* instruction in the source binary. This makes the address mapping table very big. A bigger table also takes longer time to search. We should remove as many pairs from the address mapping table as possible.

Our method is to remove the pairs which represent instructions that will *never* be the target of an indirect branch. The instructions that are function entry points, return points, function pointers are potential targets of an indirect branch. Their addresses are kept in the address mapping tables. The addresses of other instructions will not.

3.4 Linking with Native Shared Libraries

In order to link with native shared libraries, we modified LLBT and added some components (which are in italic font in Figure 3.1). The header parser extracts the prototypes of library functions from the header files. The function prototypes are used in the subsequent translation. The LLBT runtime library is responsible for the translation that cannot be performed at static time. It contains subroutines such as the wrapper of variadic functions. The details will be described in section 4.

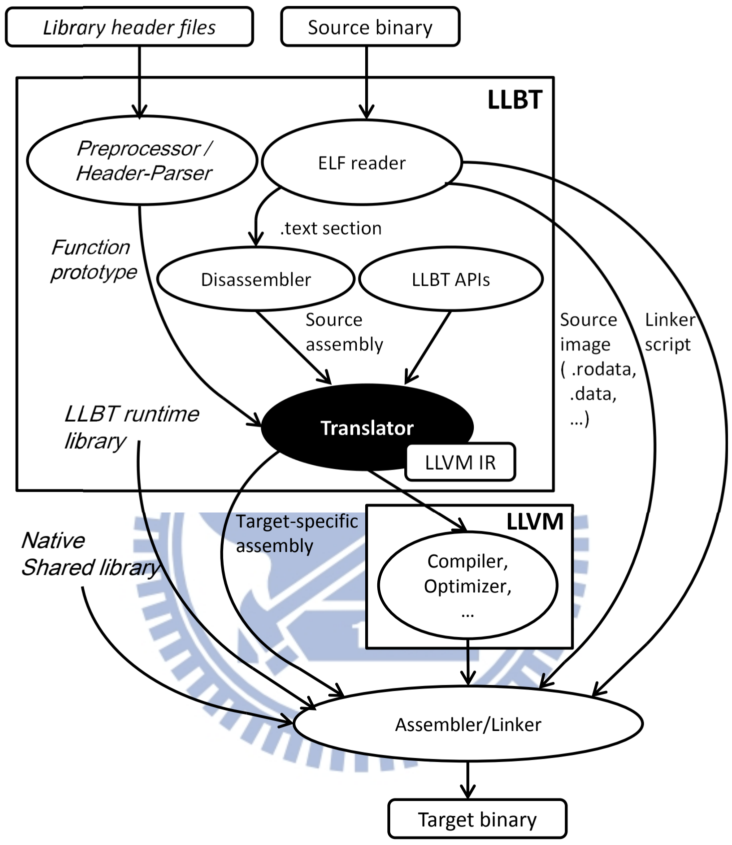


Figure 3.1: Translation flow of LLVM-based static Binary Translator (LLBT). The components in the italic font are intended for linking with native shared libraries.

LLVM IR:

```
%regType = type { i32, i32, ... , i32}

define internal void
@unexported_text_section(%regType* %regs_0) {
entry:
  ;Allocate the new local variables for emulating the ARM archicture
state.
  %regs = alloca %regType
  ;Copy the ARM architecture state from %regs_0 to %regs.
  ...
  br label %indirect_branch_stub

L_8308:
  ;The translated instructions from ARM binary at the address 0x8308.
  ...
return:
  ;Restore the ARM architecture state to %regs_0.
  ...
indirect_branch_stub:
  ;A stub for looking up the address mapping table.
  %branchTarget = load i32* %ARM_pc
  %tmp_0 = lshr i32 %branchTarget, 3
  %key = and i32 %tmp_0, 31

  ;A two-level table for speeding up lookup time.
  switch i32 %key, label %look_up_failure [
    i32 1, label %table_1
    ...
  ]
table_1:
  switch i32 %branchTarget, label %look_up_failure [
    i32 33544, label %L_8308
    ...
  ]
table_2:
  ...
look_up_failure:
  ...
}

define i32 @main(i32 %argc, i8** %argv, i8** %envp) {
entry:
  ;Allocate local variables for emulating the ARM architecture state.
  %regs = alloca %regType
  ...
  ;Initialize the emulated architecture state.
  store i32 33748, i32* %ARM_pc ;Set the entry point
  ...
  ;Pass control to the unexported_text_section for starting to execute
the translated code.
  call void @unexported_text_section(%regType* %regs)
  %ret_0 = load i32* %ARM_r0
  ret i32 %ret_0
}

;Other exported functions like callback wrappers
define i32 @compare_wrapper(i32 arg_0, i32 arg_1) {
  ;Allocate and initialize as well as main.
  ...
  ;Move incoming arguments to emulated registers/stack.
  ...
  call void @unexported_text_section(%regType* %regs)
  ;Return the result from emulated registers
  ...
}
```

Figure 3.2: An overview of the LLVM IR generated by LLBT

Chapter 4

Issues in Implementation and Their Soulutions

If the libraries on the source and target platforms share the the same application programming interface (API), e.g. μClibc[4] and glibc[2], but are possibly implemented with different application binary interfaces (ABI), our system can link the translated executable with native shared libraries by resolving the differences in ABI. Obviously, native shared libraries are more efficient than the translated ones.

ABI is a standard or convention in software. It details the rules that must be followed by binaries when they interoperate with one another. The rules may involve the calling convention, data format (i.e. type, size, and alignment), system call number, binary format (e.g. ELF), and so on. To link translated executables with native shared libraries, the most challenging issue is to resolve the differences between two different ABIs (for the source and target platforms, respectively). We will present a retargetable approach to make the translated binaries cooperate with native binaries.

Unfortunately, even if we resolve all the ABI differences, there are still certain translated shared libraries or library functions which could not be replaced by the corresponding native ones in some situations. We will discuss this issue later.

Despite this, we hope that we can still use as many functions from native shared libraries as possible. To this end, the translated binary is linked with both translated and native shared libraries at the same time. The translated binary will invoke the function from the native library if it is available. Otherwise it will use the one from the translated library.

4.1 Header Parser

It is possible to recover the arguments and return value of a function invocation by analyzing the register and memory accesses in the binary[25]. These methods are complicated and *incomplete*, especially for variable-length argument lists.

We implement a *header parser* that, together with the GCC compiler[1] (`gcc -E`), parses header files of the shared library to collect type and function declarations. The output of the header parser is a text file containing function prototypes that are composed of primitive types (e.g. `void`, `int`, float, double, etc.)[18] and derived types (e.g. struct(int, int)). Furthermore, in order to generate wrappers for callbacks (which will be discussed in section 4.4.3), the type of a function pointer argument must be recorded precisely. For example:

```
void qsort(addr, ulong, ulong,
                funcaddr(int compar(addr,addr)));
```

Here funcaddr is a new keyword for simplifying parsing. The above string means that the function qsort returns void and takes four arguments whose types are pointer to data (i.e. addr), unsigned long, unsigned long and function pointer, respectively. The fourth argument, compar, points to a function that takes two pointers to data and returns an integer.

The header parser extracts the function prototypes of the shared library for both the source and target platforms. The function prototypes on both platforms are compared. If they are compatible, LLBT will translate calls to external functions into calls to native library functions using the function prototypes. Otherwise, calls to external functions are translated into calls to the corresponding functions in the translated library.

4.2 External Function Calls

In order to link with the native libraries for arbitrary target platforms, external function calls are translated into the LLVM call instructions. The LLVM call instruction is a machine-independent IR that takes the function name and a list of explicitly typed arguments. The LLVM backend will generate function calls following the calling convention of the target platform. The resulting object will be linkable with other objects with the target linker.

Consider the ELF dynamically linked ARM binaries[8]. An external function call is a (direct or indirect) branch instruction that jumps to one of the entries in the *Procedure Linkage Table* (PLT) instead of the actual address of the external function. There are two cases to consider:

1. The function call is a direct branch.

2. The function call is an indirect branch.

4.2.1 Direct External Function Calls

Because the branch target address of a direct external function call is known at static time, we can find the corresponding function name based on the address from the table of dynamic symbols [1] The function name is used in the LLVM `call` instruction. Furthermore, the LLVM `call` instruction needs a list of explicitly typed arguments. LLBT builds a list of arguments. The arguments are taken from emulated registers/stack according to the calling convention of the source architecture[9] and its function prototype (obtained from the header parser). Similarly, the return value must be moved from an LLVM variable to an emulated register after the external function returns.

[1]The table of dynamic symbols is preserved even in a stripped binary since the dynamic linker will need it to resolve function names at run time.

Lines 11-20 in Figure 4.1 is an example of a direct external function call in LLVM.

Dynamic linking is done with the target dynamic linker (rather than the translated source dynamic linker). This approach saves a lot of overhead emulating the linkage operations if the translated source dynamic linker is used. Specifically, a `bl` (branch-and-link) instruction for an external function call in the source (ARM) binary will save the return address in a register. This return address is useless when the `bl` instruction is translated into an LLVM `call` instruction directly (by LLBT). The `call` instruction is compiled into target instructions which will not use the return address saved by `bl`. This brings several benefits:

1. On the call site, the instruction that stores the return address to the link register can be eliminated (line 17 in Figure 4.1).

2. The pair of source address and the corresponding LLVM label of the next source instruction (line 22 in Figure 4.1) become useless. It is unnecessary to put the pair in the address mapping table. This results in a smaller table and reduction of its lookup time.

3. When the external function returns, the return is an indirect branch with native instructions. Compared with a return of an internal function call that is also an indirect branch but is emulated by several instructions which load the branch target from the emulated register and look up the corresponding LLVM label in the address mapping

table, the native return is much faster.

In summary, using the native `call` instruction is certainly much more efficient than emulating the source `call` instruction.

4.2.2 Indirect External Function Calls

As we discussed in section 3.3, the address of an indirect branch is unknown until it is about to be executed. LLBT prepares an address mapping table for indirect branches. An indirect external function call is also an indirect branch instruction, such as `blx    r3` in Figure 4.1 except that its branch target is one of the entries in PLT. Unlike direct external function calls, indirect external function calls do not have the branch target address at static time. So neither the function name nor the function prototype is known.

Further, an indirect branch could be either an external function call or an internal branch. Therefore, we do not translate indirect function calls to LLVM `call` instructions. Instead, they are translated to indirect branches. In other words, an indirect branch will reach the corresponding LLVM label of its branch target via the address mapping table. If the indirect branch is an external function call, it will reach the PLT entry of the external function after emulating the indirect branch. As a result, instead of translating the original source instructions in PLT, we translate each PLT entry into a sequence of instructions emulating the corresponding direct external function call, setting up the argument list, and emulating the function return via the emulated link register (i.e. %ARM_lr). See lines 27-30 in Figure 4.1.

4.3 Tail Call Elimination Handling

Tail call elimination is an optimization often used in a compiler. For a tail call, its `return` is combined with a previous `return` into a single `return`. The combined `return` may drop off multiple stack frames at once. This saves several run-time overhead.

Note that translated functions and native library functions do not share the same registers/stack: The translated functions use an emulated registers/stack while the native functions use the native registers/stack. Consider Figure 4.2(a). A translated function, say `foo()`, calls another translated function, say `bar()`. Then `bar()` makes a tail call to a native library function `external_function()`. When the native function returns (which is done via the native link register), it does not return to the intended caller `foo()`. But rather, it returns to the original translated function `bar()`. In this case, LLBT needs to add code to return from `bar()` to `foo()`. In Figure 4.2(b), Lines 4-6 are the added instructions for the emulated `return`. These instructions move the register `lr` to `pc` and indirectly branches to the address of `pc`. On the other hand, the stack frame of the source stack has not been adjusted as well (the callee adjusts only the target stack). In the case of ARM source binary, we only have to deal with the return problems. It is not necessary to adjust the stack frame since the ARM stack is adjusted before a tail call.

The indirect branch of an emulated return is still necessary even if `external_func` is a translated library function. Therefore, the overhead for handling tail call elimination is merely the native return. The native return uses the target address with target instructions, so the overhead is very small.

4.4 Arguments

When translating an external function call, LLBT adds instructions to move the arguments from the emulated registers or the stack to LLVM variables (according to the calling convention of source platform and the function prototype obtained from header parser). In this sections, we discuss how to handle special arguments that might raise issues while linking with native shared libraries.

4.4.1 Variable-length Argument List

A *variadic function* function may accept a different number of arguments at different call sites. To recover an external function call with a variable-length list of explicitly typed arguments, we need to figure out the length and types of the arguments at the call site. But it is difficult to determine the argument list by static analysis[17]. Although the arguments passed in registers can be easily determined by data flow analysis, the arguments passed in the stack cannot. Therefore, we replace a call to a variadic function with a wrapper function that has a fixed-length argument list and translates the calling convention at run time.

In the standard C library, a variadic function usually has a corresponding function that has a fixed-length argument list and one of the arguments determines the number of arguments and argument types if necessary. For example:

```
int printf(const char *format, ...);
int vprintf(const char *format, va_list ap);
```

The variadic function `printf` has a corresponding function `vprintf`, which has a `va_list` type argument `ap`, which is actually a pointer to an array that holds the variable-length arguments. The first argument `format`, which is a string, determines the number and types of arguments.

For variadic functions, we create a wrapper that uses an argument (e.g. the `format` string in the `printf`) to determine the number of arguments and how to copy them from the emulated registers/stack, and then copies the rest of arguments to a variable (e.g. `ap`) which will be passed to the corresponding function (e.g. `vprintf`).

The above method is useful only if we know how to determine the number of arguments of the variadic function. If a function has an unknown way to determine the number of arguments, the wrapper cannot be created correctly. In this situation, we have to call the translated library function (rather than the function in the native shared library), which emulates the calling convention of the source platform.

4.4.2 64-bit Data Type

In the 32-bit architecture, the double-precision floating-point type (`double`) occupies two registers. There are two ways to compose a 64-bit data value: the first register holds either the lower or the higher 32 bits; the rest is in the second register. In LLBT, the translator recovers each `double` argument to a `double` LLVM variable, and then the arguments in the translated program will be passed to the callee obeying the appropriate calling convention. Recovering `double` arguments needs not only the function prototypes but also the source architecture's floating-point format. All other primitive types whose lengths are more than 32 bits, such as `long long`, are treated similar to `double`.

Different hardware platforms may enforce different *double-word alignments*. In translating the source binaries to LLVM IR, it is necessary to consult the double-word alignment on the specific hardware platform when a double-word argument is encountered.

4.4.3 Callbacks

Callback is a function usually located in the executable. It is passed as an argument (i.e, a *function argument*) when another function calls a library function. The library function then calls (directly or indirectly) the callback via the argument. A function argument is passed as a pointer to the callback function. We need to be careful that this pointer contains the address of the callback function in the *source* binary, which is not the address in the *translated* binary.

When linking the translated binary with the native libraries, a native-library function cannot invoke the callback directly through the function argument. This situation is similar to an indirect jump instruction.

For each callback function, a *wrapper function* is created. The function argument actually contains the address of the wrapper, not the address of the callback function. The wrapper function has the same function prototype as the corresponding callback function. Hence, when a library function invokes a callback, it actually jumps to the corresponding wrapper function. The wrapper function will transform the arguments (which are in the target-platform's calling convention) to the calling convention of the source platform and then invoke the actual callback. Return from the callback function is handled analogously.

There is another method to handle callbacks[23], which places the instructions that redirect the program to where the translated callback is located at all the callback source addresses. It's straightforward, but it needs to find out all the callback addresses at static-time for the placement of redirection instructions with a SBT system, or it needs run-time help. Moreover, the analysis of finding callbacks at static-time is not trivial and the redirection instruction are target-dependent and required being written in assembly, thus LLBT chooses another method.

4.5 Architecture-specific Functions

Some functions are architecture-specific in that they make use of unique features in the architecture. Examples are `setjmp` and `longjmp`. These architecture-specific functions on the source platform cannot be replaced by the corresponding architecture-specific functions on the target platform directly; we must always execute their translated binaries. For example, the `setjmp` and `longjmp` functions save and restore a program's calling environment (i.e. registers) to the `env` argument for non-local jumps, respectively. If the translated program calls the native `setjmp`, the program's calling environment saved by the `setjmp` is the target registers, but not the emulated source registers, which means that the emulated program's calling environment is lost and subsequent program execution is meaningless.

An alternative way to avoid translating shared libraries is to write an emulated function by hand. The benefit is that the hand-written shared library is smaller since only architecture-specific functions are included.

4.6 Linking with Translated Shared Libraries

As discussed previously, there are still translated shared libraries or library functions that could not be replaced by the corresponding native ones, which occur in the following situations:

1. The API of the library is unknown. Our implementation will not work

without the header files of the library.

2. The corresponding native shared library is not available on the target platform. For example, the library has not been ported to the target platform.

3. The function has a variable-length argument list but we do not know how to determine the number of arguments. In this case, a wrapper for it cannot be created.

4. The function is architecture-specific and its result depends on the source architecture.

In our implementation, we link the translated executables with both the translated shared library and the native shared library. It will attempt to use the functions in the native library whenever possible.

In translated shared libraries, the prototype of all exported functions is identical to that of the `unexported_text_section` function. All function names end with a suffix (i.e. `_LLBT`). For example, the prototype of the `printf` function is

```
void @printf_LLBT(%regType* %regs)
```

This translated `printf` takes only one argument `regs`, which is a pointer to a structure of type `regType` that contains all the variables for emulating the source architecture state. That is to say, we pass the emulated source-architecture state to the callee when the caller calls a translated library function. Once the translated library function can access the emulated architecture state, it executes with the caller's emulated architecture state. In particular, it could retrieve the arguments from the emulated registers/stack. The return values are handled similarly.

Unfortunately, because the emulated architecture state are stored not in the local variables but in the memory pointed by an argument, the callee's translated instructions that operate on emulated architecture state cannot be promoted into registers as we discussed in section 3.2. Performance of the translated shared libraries hurts.

As an improvement, the translated library function may allocate local variables, copy the architecture state to local variables, do its normal work, and finally copy the contents of the local variables back to the architecture state upon return. This approach creates extra overhead but may save time in the execution of the translated library function.

Function porototype of puts in C:

```
int puts(const char *s);
```

ARM Assembly:

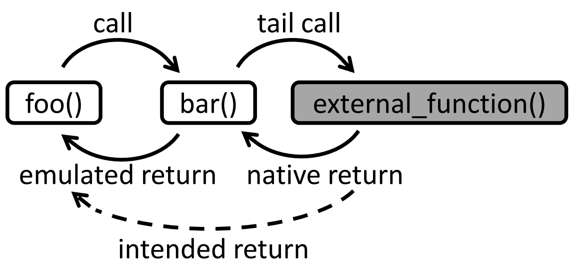
```
; the PLT entry of the function puts.
82a8: add     ip, pc, #0
82ac: add     ip, ip, #32768  ; 0x8000
82b0: ldr     pc, [ip, #528]! ; 0x210

;A direct external function call.
839c: ldr     r0, [pc, #20] ;load data at 0x83b8
83a0: bl      82a8 ;branch to the PLT entry
83a4: ...
...
;An indirect external function call via r3.
83ac: blx     r3
...
;The address of a constant string.
83b8: .word   0x83e4 ;pc-relative data
```

LLVM IR:

```
 1 ;PLT entry of the function puts.
 2 L_82a8:
 3   %arg_0 = load i32* %ARM_r0
 4   %ret_0 = call i32 @puts(i32 %arg_0)
 5   ;emulated return
 6   store i32 %ret_0, i32 *%ARM_r0
 7   %lr_0 = load i32* %ARM_lr
 8   store i32 %lr_0, i32* %ARM_pc
 9   br label %indirect_branch_stub
10 ...
11
12 ;A direct external function call.
13 L_839c:
14   store i32 33764, i32* %ARM_r0 ;store 0x83e4
15   br label %L_83a0
16 L_83a0:
17   store i32 33700, i32* %ARM_lr ;store 0x83a4
18   %arg_1 = load i32* %ARM_r0
19   %ret_1 = call i32 @puts(i32 %arg_1)
20   store i32 %ret_1, i32 *%ARM_r0
21   br label %L_83a4
22 L_83a4:
23 ...
24
25 ;An indirect external function call via r3.
26 L_83ac:
27   store i32 33712, i32* %ARM_lr
28   %pc_0 = load i32* %ARM_r3
29   store i32 %pc_0, i32* %ARM_pc
30   br label %indirect_branch_stub
31 ...
```

Figure 4.1: A translation example of the external function calls to the library function *puts*. Lines 1-8 shows an entry of the PLT. Lines 11-20 shows a direct external function call. Lines 24-29 shows an indirect external function call.

(a)



(b)

ARM Assembly:

```
;A tail call to external_function()
8438: b        0x82cc
```

LLVM IR:

```
1 L_8438:
2   call i32 @external_function()
3   ;For handling tail call elimination
4   %lr_0 = load i32* %ARM_lr
5   store i32 %lr_0, i32* %ARM_pc
6   br label %indirect_branch_stub
```

Figure 4.2: Tail call elimination. (a) A translated function `bar` makes a tail call to a native library function `external_func`. When `external_func` returns, it returns to `bar` but not intended `foo`. Thus, in (b), we add additional instructions (lines 4-6) for the emulated return to the intended `foo` function.

Chapter 5

Experimental Result

In the following experiments, we use ARM as the source architecture and x86 as the target architecture, and we run the SPEC2006 CINT [6] benchmarks on a 3.07GHz 4-core Intel i7 PC running Ubuntu 11.10. The ARM binaries and x86 binaries were both compiled with gcc version 4.4.6 using optimization flag -O2 and linked with μClibc library. The translated binaries were generated by LLVM 3.0 using optimization flag -O2. In our experiments, LLBT (as well as QEMU[5]) cannot handle the ARM binary `400.perlbench`. In addition, our LLBT cannot handle C++ programs. Therefore, the benchmarks `471.omnetpp`, `473.astar`, and `483.xalancbmk` are C++ programs and are excluded in our experiment. The results in this section were obtained from the remaining 8 benchmarks of SPEC2006 CINT: `401.bzip2`, `403.gcc`, `429.mcf`, `445.gobmk`, `456.hmmer`, `458.sjeng`, `462.libquantum`, and `464.h264ref`.

5.1 Native Shared Libraries vs. Translated Shared Libraries

We compare two approaches in translating dynamically linked ARM binaries. The first is to link with native x86 shared libraries (call this the *native configuration*) and the other is to link with translated ARM shared libraries(call this the *translated configuration*). In order to demonstrate the performance improvement obtained from linking with native shared libraries, we added some changes:

1. The names of all helper functions in the translated ARM executables were replaced with the names of the equivalent functions in the x86 *libgcc* or x86 *CompilerRT* library. For example, the function `__aeabi_fmul` in ARM *libgcc* is essentially the function `__mulsf3` in x86 *libgcc*. Only its names in the two libraries differ. We used a table to translate one name to the other in such cases.

2. A hand-written wrapper function is added for each variadic function as we discussed in section 4.4.1.

3. We add an emulated version of the architecture-specific functions (i.e. `setjmp` and `longjmp`).

With the above three changes, it is possible to work only with the native x86 shared libraries. The translated ARM libraries can be no longer needed.

5.1.1 Execution Time

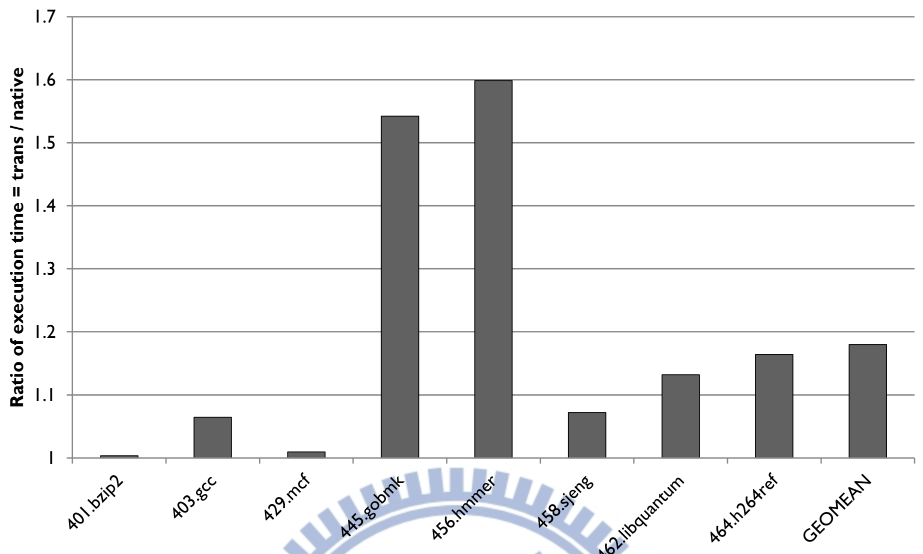


Figure 5.1: Ratio of execution times in the translated and native configurations.

We measured the execution time of the native and translated configurations. On the average, the ratio of the execution time of the translated configuration to that of the native configuration (i.e. speedup) is 1.18 (See Figure 5.1). For `401.bzip2` and `429.mcf`, there is almost no speedup. We also break down the execution time in different parts of the benchmarks (i.e. executable and libraries) using the *performance analysis tools for Linux* (perf). Figure 5.2 is the breakdown of execution time in the translated configuration and Figure 5.3 is for the native configuration. From Figure 5.2, we can see that, for `401.bzip2` and `429.mcf`, the time spent in the translated shared libraries were very small. That is why it is not possible to obtain much

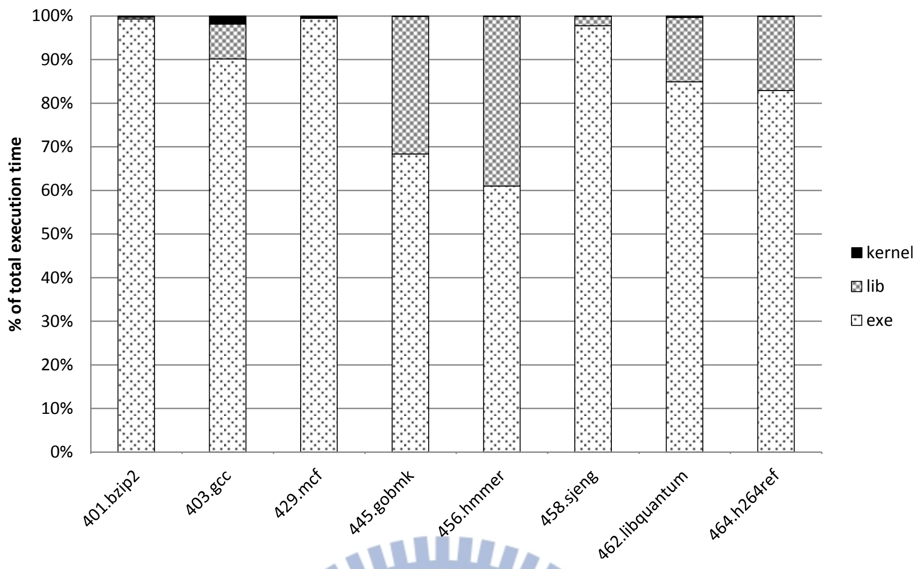


Figure 5.2: Breakdown of execution time in the translated configuration.

improvement by replacing the translated libraries with the native shared libraries.

In addition, we examined the execution time in different parts of the benchmarks. The result is shown in Table 5.1 and Table 5.2. We can see that the time spent in executables in the two configurations is almost the same; since the LLVM IR generated by our translator in the two configurations are very similar. In Table 5.1, native configuration is slightly faster because argument passing is handled differently in the two configurations. In the native configuration, every external function call has an explicit argument list and several instructions for copying the arguments. The LLVM optimizer has a chance to optimize the call site. In contrast, in the translated configuration, all external function calls have an argument (which points to

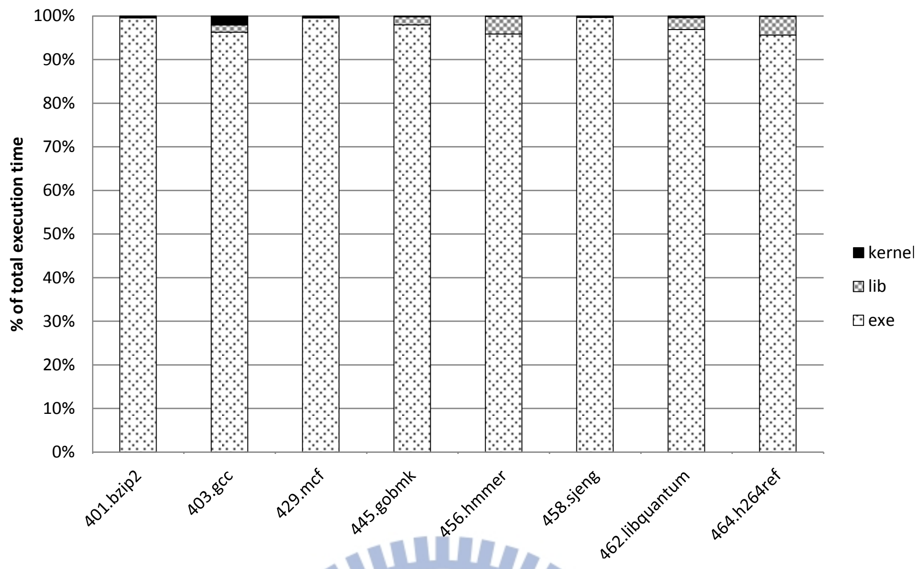


Figure 5.3: Breakdown of execution time in the native configuration.

the emulated architecture state), as we discussed in section 4.6. If the caller's emulated registers have been promoted into target registers, the caller needs to move them back from target registers to the memory where the emulated architecture state is located. Furthermore, the callee in translated shared libraries also needs to restore the emulated registers before it returned for a similar reason. In summary, external function calls incur more calling overheads in the translated configuration.

In Figure 5.1, the speedups for `445.gobmk` and `456.hmmer` are 1.54 and 1.6, respectively, in the native configuration. The large speedup is due to the following reason: The two benchmarks spent a large portion of time in the shared libraries (See Figure 5.2). This part of the execution has been reduced by 26.22 and 15.59, respectively for `445.gobmk` and `456.hmmer` (see

Table 5.2).

There are two reasons why the two benchmarks takes so much time in the translated configuration:

benchmark	trans conf	native conf	trans/native
401.bzip2	899.73	899.02	1.00
403.gcc	1468.07	1471.65	1.00
429.mcf	321.25	318.55	1.01
445.gobmk	1323.93	1230.57	1.08
456.hmmer	1173.63	1153.48	1.02
458.sjeng	1367.63	1300.82	1.05
462.libquantum	575.78	580.39	0.99
464.h264ref	1651.19	1636.12	1.01

Table 5.1: The time (sec) spent in the executable and the time ratio of the translated configuration and the native configuration.

1. The two benchmarks contains many calls to helper functions that execute floating-point operations. Helper functions in the ARM library emulate floating-point operations in software and the library is emulated again by our translator, which makes the translated helper functions much slower than the native ones, in which the floating-point operations are executed with x86 hardware instructions directly.

2. As we discussed in section 4.6, the translated library functions needs to

benchmark	trans conf	native conf	trans/native
401.bzip2	3.71	1.44	2.57
403.gcc	128.77	25.84	4.98
429.mcf	0.45	0.16	2.83
445.gobmk	609.19	23.24	26.22
456.hmmer	746.32	47.88	15.59
458.sjeng	29.38	2.35	12.51
462.libquantum	99.67	16.41	6.07
464.h264ref	337.64	72.38	4.67

Table 5.2: The time (sec) spent in the shared libraries and the time ratio of the translated configuration and the native configuration.

copy/restore the emulated registers to/from the caller's registers. This increases the calling overhead in each external function call. Unfortunately, the two benchmarks make a lot calls to external functions (see Table 5.3, which shows the numbers of external function calls including helper function calls and total library function calls. The statistics is collected with the library-call tracer *ltrace*[3] with `-c` flag) and most of the called library functions run too short to compensate the calling overhead.

benchmark	helper	total
401.bzip2	0	305,403
403.gcc	0	25,160,973
429.mcf	37,197	470,604
445.gobmk	458,547,244	723,632,501
456.hmmer	5,932,217,722	6,489,272,725
458.sjeng	0	273,340,052
462.libquantum	1,242,026,165	1,294,635,444
464.h264ref	42,787,106	1,079,500,458

Table 5.3: Number of external function calls: helper function calls and total library function calls.

5.2 Binary Translation vs. Recompilation of Source Code

It is interesting to investigate the quality of our LLBT binary translator by comparing the performance of the translated code against the best possible performance. The best possible performance is usually achieved by recompiling the source code directly for the target platform. For the comparison, we measured the execution time of the recompiled x86 program, the translated ARM executable linked with native x86 shared libraries and the translated ARM executable but linked with translated ARM shared libraries. The inverse of the execution time is considered as the performance. Figure 5.4

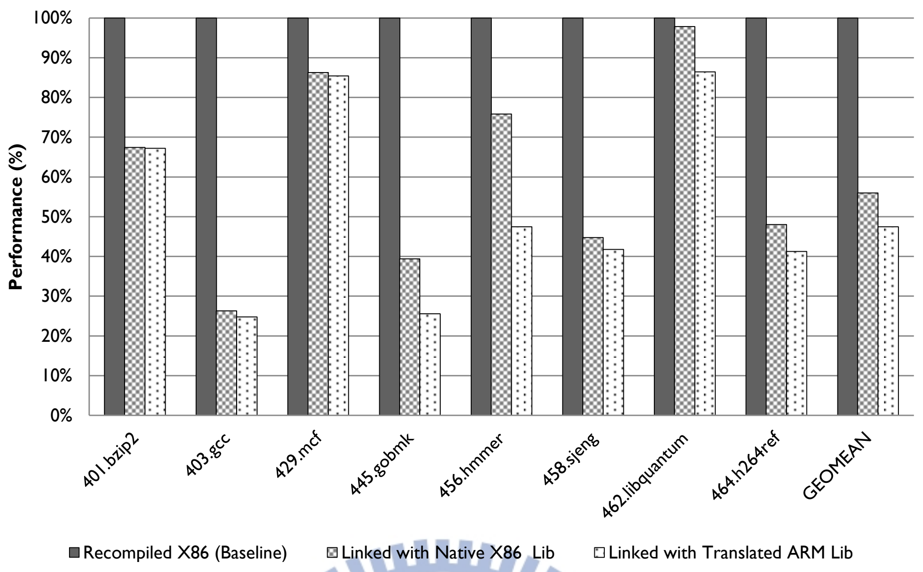


Figure 5.4: Performance compared with the recompiled x86 program

shows the performance comparison using the recompiled x86 program as the baseline.

For the benchmark `462.libquantum`, the performance of the translated code linked with the native shared libraries is 97.8% of the recompiled code. In comparison, the performance of the binary translation that links with the translated shared libraries is 86.4% of the recompiled code. The average performance of the two approaches of binary translation is 56.0% and 47.4%, respectively.

5.3 Limitation and Future Work

The focus of our study is dynamic linking with the native shared libraries in binary translation. Currently our implementation does not support C++ programs because in a C++ program, a member function might be invoked by a native library function through a class pointer. This is an implicit callback. Our implementation cannot handle this implicit callback.

Our implementation also requires the header files of the shared libraries in order to determine the function prototypes. If the files are unavailable, our implementation cannot make use of the native libraries.

Chapter 6

Conclusions

In this thesis, we studied linking translated executables with native shared libraries in binary translation. Intuitively, native shared libraries is faster than the translated one. Our approach is *retargetable* because we use LLVM IR as an intermediate representation of the binary code. Part of the work in retargeting to different platforms is delegated to existing LLVM framework. We identified several problems and proposed their solutions related to linking. Our experiments show that the translated programs linked with native shared libraries can achieve an average speedup of 1.18 when comparied with the ones linked with the translated shared libraries and kept 55.9% performance when compared with recompiling the source programs for the target platform.

Bibliography

[1] GCC, the GNU Compiler Collection.

[2] GLIBC, the GNU C Library.

[3] ltrace.

[4] μClibc.

[5] QEMU.

[6] Standard Performance Evaluation Corporation.

[7] The LLVM Compiler Infrastructure.

[8] ARM Compiler Tools Group. *ELF for the ARM Architecture*, October 2009.

[9] ARM Compiler Tools Group. *Procedure Call Standard for the ARM Architecture*, October 2009.

[10] Jiunn-Yeu Chen, Wuu Yang, Jack Hung, Charlie Su, and Wei Chung Hsu. A Static Binary Translator for Efficient Migration of ARM based

Applications. In *Proceedings of the 6th Workshop on Optimizations for DSP and Embedded Systems*.

[11] Anton Chernoff, Mark Herdeg, Ray Hookway, Chris Reeve, Norman Rubin, Tony Tye, S. Bharadwaj Yadavalli, and John Yates.

[12] Vitaly Chipounov and George Candea. Enabling sophisticated analyses of x86 binaries with RevGen. In *Intl. Conf. on Dependable Systems and Networks*, June 2011.

[13] Cristina Cifuentes, Cristina Cifuentes, Mike Van Emmerik, Mike Van Emmerik, Norman Ramsey, Norman Ramsey, Brian Lewis, and Brian Lewis. Experience in the design, implementation and use of a retargetable static binary translation framework. Technical report, Mountain View, CA, USA, 2002.

[14] Cristina Cifuentes and Mike Van Emmerik. UQBT: adaptable binary translation at low cost. *Computer*, 33(3):60 –66, mar 2000.

[15] Cristina Cifuentes and Vishv Malhotra. Binary Translation: Static, Dynamic, Retargetable? In *Proceedings of the 1996 International Conference on Software Maintenance*, Washington, DC, USA, 1996. IEEE.

[16] Cristina Cifuentes and Doug Simon. Procedure Abstraction Recovery from Binary Code. In *Proceedings of the Conference on Software Maintenance and Reengineering*, Washington, DC, USA, 2000. IEEE Computer Society.

[17] Wen Fu, Rongcai Zhao, Jianmin Pang, and Jingbo Zhang. Recovering Variable-Argument Functions from Binary Executables. In *Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science*, ICIS '08, pages 545–550, Washington, DC, USA, 2008. IEEE Computer Society.

[18] I. Guilfanov. A Simple Type System for Program Reengineering. In *Proceedings of the Eighth Working Conference on Reverse Engineering*, WCRE '01, pages 357–, Washington, DC, USA, 2001. IEEE Computer Society.

[19] Raymond J. Hookway and Mark A. Herdeg. DIGITAL FX!32: Combining Emulation and Binary Translation. *Digital Tech. J.*, 9(1):3–12, Jan 1997.

[20] John. R. Levine. *Linker and loader*. Morgan Kaufmann, 2000.

[21] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, and Wuu Yang. LLBT: An LLVM-based Static Binary Translator. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES'12, Tampere, Finland, October 2012.

[22] Richard L. Sites, Anton Chernoff, Matthew B. Kirk, Maurice P. Marks, and Scott G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, February 1993.

[23] Matthew Smithson, Kapil Anand, Aparna Kotha, Khaled Elwazeer, Nathan Giles, and Rajeev Barua. Binary Rewriting without Relocation Information. Technical report, University of Maryland, November 2010.

[24] Padraig O' Sullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D. Keromytis. Retrofitting Security in COTS Software with Binary Rewriting. In *Proceedings of the 26th IFIP International Information Security Conference*, 2011.

[25] Jingbo Zhang, Rongcai Zhao, and Jianmin Pang. Parameter and return-value analysis of binary executables. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, volume 1, pages 501 –508, july 2007.