Dynamic Binary Translation for Multi-Threaded Programs with Shared Code Cache

Student: Chia Lun Liu

Advisors: Wuu Yang

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

June 2013

Hsinchu, Taiwan, Republic of China

blank

blank

blank

Contents

List of Figures

Chapter 1

Introduction

Binary translation emulates one ISA through another ISA. It has applications such as program instrumentation [18], fast simulation [17] and security investigation [19]. In the thesis, the emulated ISA is called **guest**, and the emulating ISA is called **host**. Our work is based on mc2llvm [13]. mc2llvm is a process-level ARM-to-x86_32 hybrid binary translation system which is able to do static and dynamic binary translation. Currently, it is only able to emulate sequential programs. Our work focuses on translating issues on emulation of multi-threaded binary code using dynamic binary translation. DBT (dynamic binary translator) takes the guest executable file as data. At run time, guest instructions are translated to host instructions whenever needed. The host instructions are cached in the code cache so further emulation can be done by directly executing codes in the code cache without repeated translation. In the system, the action of a guest thread is emulated

by an individual host thread called **emulating thread**. The task of the emulating thread can be briefly described as follows:

1. Given the address of an guest basic block, it looks up the address mapping table for the starting address of the corresponding host instructions. If the thread fails to find it, goto (3). Otherwise, goto (2).

2. It executes the host instructions stored in the code cache. After that, it will be given the address of the next guest basic block then goto (1).

3. It translates the guest basic block to the host instructions, keeps host instructions in the code cache and stores the mapping of the address of the guest basic block and the starting address of host instructions into the address mapping table. Then goto (2).

When DBT is emulating multi-threaded binary code, each emulating thread would access to the same components in the translation system such as the code cache, the instruction translator and the address mapping table. One thing to do with is the synchronization of them because threads may access to them simultaneously. This issue would be discussed later.

To further speed up the emulation, we do trace compilation. A trace has a single entry and multiple exits. In binary translation, frequently successive emulated guest basic blocks are collected to become a trace. We split trace compilation into 2 parts: trace selection and trace generation. The emulating threads would do trace selection. The form of the collected trace is a

vector of guest addresses. The emulating thread pushes its collected trace in a queue called trace queue. Since trace generation is a time consuming task, the system has 3 extra threads called optimizing threads dedicated to trace generation. They periodically try to pull the trace out from the trace queue. Once the optimizing thread succeeds in pulling a trace, it does code generation for the trace.

In summary, we study the issues on translation of multi-threaded binary code and implement a scalable and efficient binary translator for emulating multi-thread binaries. In this thesis, We make the following contribution:

- turn mc2llvm to emulate multi-thread ARM binaries
- design the shared code cache
- craft efficient lock-free address mapping tables
- speed up emulation speed by trace compilation
- support x86_64 backend

In the following sections, some terms are used very often and they are stated here for clarity.

Translation Block: It is a container to store LLVM IR that corresponds to guest instructions in a basic block. A translation block has two addresses. One is the guest address. It is the address of the guest basic block. The other is the host address. It is the starting address of the host instructions that emulate the actions of guest basic block.

Emulating Thread: It is a host thread that emulates the actions of the guest thread.

Optimizing Thread: It is a host thread dedicated to trace compilation.

The rest of the paper is organized as follows: Chapter 2 discusses related work. Chaper 3 discuss the background knowledge on multi-threaded programs. Chaper 4 describes the design and implementation. Chapter 5 presents the optimization technique of the system. Chapter 6 talks about the synchronization in the translation system. Chaper 7 presents the experiment result. Chapter 8 concludes the work and talks about the future work.

Chapter 2

Related Work

This chapter introduces some existing systems related to binary translation and optimization techniques used in dynamic binary translation.

2.1 Binary Translation

Dynamo [20] is a dynamic optimization system. It states that some optimization opportunity is not available to the static compiler. For example, optimization of executables with dynamic linked libraries as a whole is not possible. To speed up emulation speed, Dynamo picks the trace. The destination address of a backward branch is a candidate of the trace head. The candidate becomes the trace head when it has been visited over a certain times and the rest of the trace is selected based on the execution flow once the trace head is found. After a trace is selected, Dynamo optimizes the trace and emits it in the cache. Thus, subsequent encounter of trace entry address

would cause control transfer to the trace. Trace compilation has several advantages. For example, due to its larger scope, it gains much optimization opportunity, and there is no joint point in it. The experiment result delivers good news that optimization during runtime is possible.

HDTrans [8] is a IA-32 to IA-32 dynamic instrumentation system. It lists many simple and effective optimization to translate branch instructions. To translate unconditional direct branches, it elides the branch instruction and keeps on translating at the destination of the branch instruction. To translate conditional branches, if the destination of the true/false branch is already translated before, it patches a jump to existing translated block.To handle return instructions, it draws on the return caching. To handle the rest of indirect branches, it uses the address mapping table.

mc2llvm [13] is a LLVM-based hybrid binary translation system. It is an variation of static binary translation. Basically, it performs just like the way of a static binary translator but when it cannot deal with the code location problems, it turns into a dynamic binary translator to solve the problem at runtime.

QEMU [21] is a open source and fast portable dynamic translator. It supports to emulate a wide range of different architectures such as x86, ARM, MIPS, SPARC. QEMU first translates guest instructions into its own TCG IR then turns TCG IR into host instructions. Two steps translation is a good technique to make it retargetable easily. Besides, QEMU can be a user-mode or system-mode emulator. PQEMU [6] is an variation of QEMU. PQEMU

modifies QEMU to utilize the underlying real multi-core machines when it does full-system emulation. In addition, PQEMU can configure threads to use shared or private cache. COREMU [2] is another variation of QEMU which also aims at full system emulation. Each thread is equipped with an individual QEMU instance and threads use private code cache. HQEMU [7] is also derived from QEMU. It picks traces and has dedicated threads to do trace generation. Furthermore, it merges frequently executed traces together.

2.2 Shared Code Cache

Having a shared code cache seems required when the translation system is doing emulation for multi-threaded applications. Some applications have threads to do similar tasks and having a shared code cache prevents memory explosion from duplication of translation blocks. [3] proposes to use the shared code cache. It scrutinizes the issues on synchronization of the shared code cache, and provides light-weight techniques to do with the synchronization.

2.3 Trace Selection

Building traces is an effective method to accelerate the emulation. Frequently consecutive emulated guest basic blocks are usually picked to become a trace.

The first thing to do with building a trace is trace selection. Concerning trace selection, some papers [20] [9] [7] focus on selecting the trace head while some [5] focus on the whole structure. In [20], the block whose address is the destination of a backward branch executed over a certain times is a trace head. Once the trace head is found, the rest of the trace is speculatively chosen based on the execution flow. In [7], it records the guest address of the visited blocks into a list and searches for a sequence which starts and ends at the same guest adress and that sequence is picked as a trace. Moreover, [9] improves it by filtering false loop. [5] also records visited blocks, it tries to reconstruct the program structure by sequitur algorithm and extracts the hot region in the structure. The above methods do trace selection at run time. All the methods need to be light-weight in order not to influence the speed of emulation. Next, the collected trace has to be optimized and generated to host instructions. This task can be time consuming. Some systems stop emulation to do it while some spawn another thread dedicated to it.

Chapter 3

Background

In this part, we first describe how mc2llvm works and then discuss the following issues on multi-threaded programs.

- How is a thread created and terminated?
- How to manipulate TLS (thread local stroage) base address?
- Atomic operations

3.1 Program flow of mc2llvm

In figure 3.1.

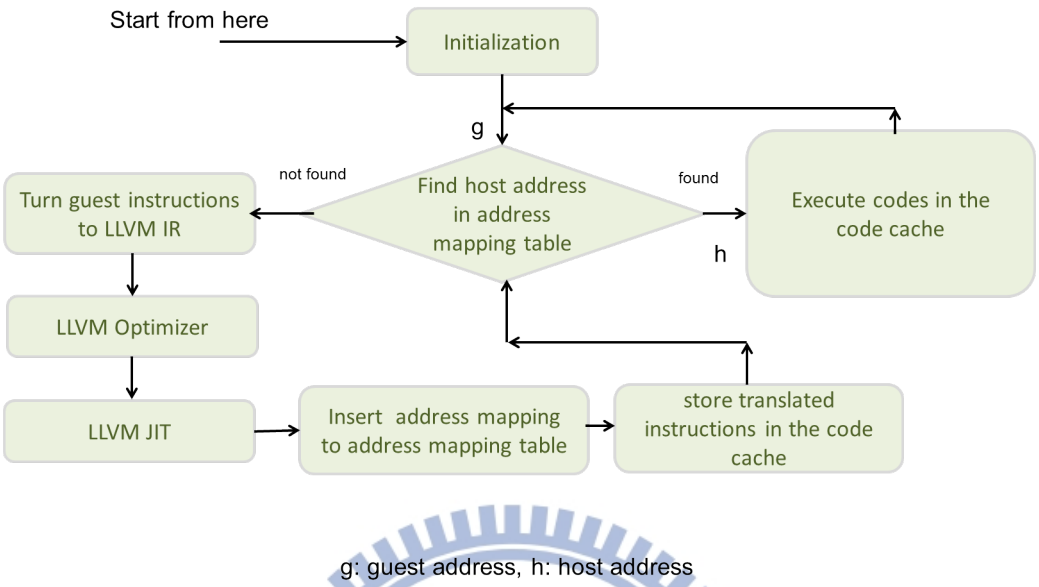

Figure 3.1: Program flow of mc2llvm

```
32ba0:    e3a07078    mov    r7, #120    ; 0x78
32ba4:    ef000000    svc    0x00000000
32ba8:    e3500000    cmp    r0, #0    ; 0x0
32bac:    0a000002    beq    32bbc <__clone+0x4c>
32bb0:    e8bd0090    pop    {r4, r7}
32bb4:    ba000919    blt    35020 <__syscall_error>
32bb8:    e12fff1e    bx     lr
```

Figure 3.2: A code snippet to do clone system call

3.2 Thread Creation and Termination

A thread is created by Linux **clone** system call. Figure 3.2 contains a code snippet to do clone system call dumped by GNU objdump. On the ARM architecture, system call is triggered by svc instruction with system call number

passed in r7.

Line 1: set r7 to 120, 120 is the clone system call.

Line 2: the current thread calls clone system call.

Line 3: Now there is one more thread. The value of r0 in the parent thread is the id of the child thread. The value of r0 in the child thread is 0.

Line 4: The child thread branches to $0x32bbc$ while the parent thread ignores the instruction.

Line 5: The parent thread executes the instruction following "beq 32bbc".

A thread is terminated by Linux **exit** or **exit_group** system call. exit kills the current thread while exit_group kills all the threads.

Because clone system call is more complex than exit and group_exit system calls, we discuss more on it by delving into its parameters passed in the clone system call. The following is the prototype of clone system call extracted from linux kernel.

```
asmlinkage int sys_clone(unsigned long clone_flags, unsigned long newsp,
int __user* parent_tidptr, int tls_val, int __user *child_tidptr,
 struct pt_regs *regs);
```

- clone_flags: specify what is inherited from the parent
- newsp: the address of the stack for the new thread
- parent_tidptr: a memory place to store the pointer to child id if CLONE_PARENT_SETTID is on
- tls_val: the base address of the thread local storage for the new thread

- child_tidptr: a memory place to store the pointer to child id if CLONE_CHILD_SETTID is on
- regs: the registers' state of the parent thread

Understanding the parameters passed in clone system call is important to emulating clone system call. This is because we can know what is already done before creating a new thread. For example, the stack of the new thread is already allocated before executing clone system call.

3.3 Manipulation of TLS Base

In a high level language like C, if a static or global variable is specified with __thread, it is a thread local variable and stored in tls (thread local storage). On the ARM architecture , the base address of thread local stroage is kept by a specific register set called cp15. On older ARM architecture (v5 and earlier), there is no such register set, Linux takes a software approach by calling a function at a specific address. The function is given as follows:

```
typedef void*(__kuser_get_tls_t)(void);
#define __kuser_get_tls(*(__kuser_get_tls_t*)0xffff0fe0)
```

This function **__kuser_get_tls** would return tls base address. To set tls base, we should call ARM-private system call named **set_tls** to set the tls base.

3.4 Atomic Operations

Atomic operations can be performed in two ways:

- atomic instructions. For example, ARM has ldrex, strex, swp instructions.

- Linux kernel support. For example, the function named **__kuser_cmpxchg**

  ```
  typedef int (__kuser_cmpxchg_t)(int oldval,int newval,volatile int* ptr);
  #define __kuser_cmpxchg(*(__kuser_cmpxchg_t*)0xffff0fc0);
  ```

  performs atomic compare-and-exchange operation and sets the conditional flag C if the operation is successful.

A typical program which requires mutex, conditional variable or semaphore would demand the library like pthread to support the operations. In other words, how the programs do synchronization depends on the implementation of the library function. Our benchmark uses openmp to support multithread. The lock operation is done by calling Linux system call futex. It is futex which uses atomic instructions to implement the atomic operations. Since mc2llvm is a process-level binary translator, it does not need to emulate atomic instructions except that the emulated program is embedded with inline assembly which uses atomic instructions.

Chapter 4

Design and Implementation

The whole translation system is graphically presented in figure 4.1. In this part, we spell out how the translation system does the emulation.

First, we do some initialization such as mapping the code and data of executable file into the main memory. Then we would be given an entry point. We find in the address mapping table the address of translated codes. If we do not find it, we use turn guest instructions at the guest address to

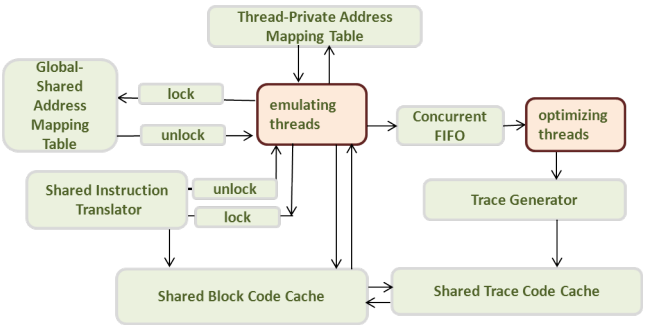

Figure 4.1: The architecture of the translation system

LLVM IR. Then we use LLVM optimizer and JIT to help us generate host instructions or called translated instructions of good quality. Next, we insert the mapping of guest and host address to the address mapping table and store the translated instructions into the code cache. We find in the address mapping table again. If we find the mapping, we execute the codes in the code cache. After executing code in the code cache, we would be given the guest address of the next instruction to execute.

4.1 Memory Initialization

DBT would need to read guest instructions to do instruction translation and access to the global and static variables of the guest program. Thus, it needs to map guest text, data, bss sections into host memory. On the other hand, it has to prepare the guest heap and guest stack for the main guest thread. The guest stack of the main thread thread and the guest heap are allocated from the host heap through mmap system call. By default, the size of the guest stack is set to 8MB and the guest heap is set to 200MB. Besides, we need to prepare argc, argv, envp, auxv for the guest stack. argc, argv, envp are borrowed from the stack of the host thread and auxv is handcrafted. On the ARM architecture, the starting address of text section is $0x8000$ by convention. mc2llvm maps those sections to the host memory at $0x8000$ directly. You can see the memory layout of the system in the figure 4.2. Figure 4.2 shows that the host text section starts at $0x50000000$. This is

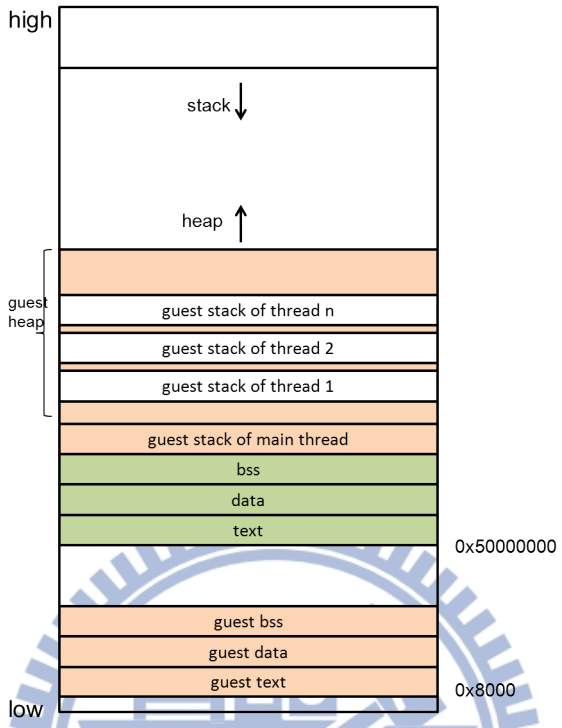

Figure 4.2: Memory layout

because the guest stack and heap are allocated from the host heap and they should be put within 32-bit boundary.

4.2 State Mapping

mc2llvm originally mapped each guest register and condition flag to a LLVM global variable, respectively. But it is impossible to use this mapping technique to have a shared code cache. Instead, a new method is devised to do state mapping. We have a data structure **GuestState** to keep the guest registers and the condition flag. Each thread has its own **GuestState**. Fig-

ure 4.3 shows the difference between the original and the new method.

```
int translationblock()
{
store i32 1, i32* @ARM_r0
…
}
```

original mc2llvm

```
int translationblock(struct GuestState* st)
{
  %1 = load st->r[0]
  store i32 1, %1
…
}
```

modified mc2llvm

We use pseudocode to describe the difference.

Figure 4.3: State mapping

4.3 Emulating Threads

Each guest thread is emulated by host thread, called an **emulating thread**. Each emulating thread has a structure called ThreadCtx in mc2llvm. It contains thread id, guest architecture state, guest stack, thread-local storage, child_tidptr and etc. child_tidptr is a parameter passed in the clone system call. This value is important when the thread emulates the exit system call. Except for the emulating thread which emulates the guest main thread, other emulating threads are created when mc2llvm is emulating the clone sytem call. Emulating threads are terminated when mc2llvm is emulating the exit or exit_group system call.

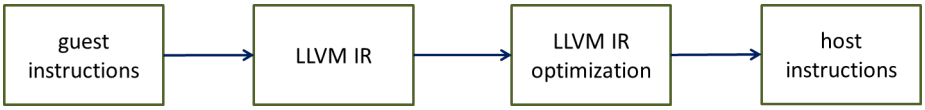

Figure 4.4: The procedure of instruction translation

4.4 Instruction Translation

In the system, instruction translation is divided into 3 parts: (1) turn guest instructions to LLVM IR (2) optimize LLVM IR (3) turn LLVM IR to host instructions. The step is graphically presented in figure 4.4. mc2llvm divides a guest instruction into 3 parts because most ARM instructions are predicated. They are the prolog, body, epilog.

- **prolog**: Test the related condition flags. This determines whether to execute the instruction.
- **body**: Execute the instruction and modify the necessary condition flags.
- **epilog**: Direct program flow to the next guest instruction.

An example of instruction translation is given in figure 4.5. We describe LLVM IR in figure 4.5 as follows:

Line 1,8,10: LLVM label, the number(b15c) in the label indicates the address of the guest instruction.

Line 2,3,4,5: the instructions load the condition flag Z from the guest architecture state.

the ARM instruction

```
1                beq b1a0
```

the LLVM IR used to emulate the instruction:

```
1  L_0000b15c_prolog
2    %49 = load { i32, [32 x i32], i64, i64, i64 }** %2, align 4
3    %50 = getelementptr inbounds { i32, [32 x i32], i64, i64, i64 }* %49, i32 0, i32 1
4    %51 = getelementptr inbounds [32 x i32]* %50, i32 0, i32 18
5    %52 = load i32* %51
6    %53 = icmp eq i32 %52, 1
7    br i1 %53, label %L_0000b15c_body, label %L_0000b15c_epilog
8  L_0000b15c_body
9    ret i32 45472
10 L_0000b15c_epilog
11   ret i32 45408
```

Figure 4.5: The translated LLVM IR of an guest instruction

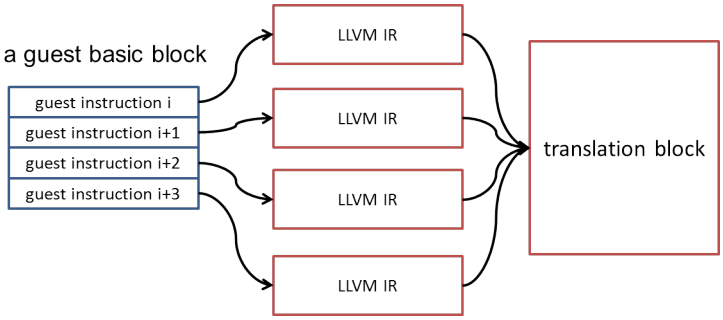

Figure 4.6: How the translation block is formed

Line 6: test if Z is 1.

Line 7: if Z is 1, branch to body; otherwise, branch to epilog.

Line 9: direct the program flow to the next guest instruction if the branch is taken. 45472 equals to b1a0 in hex.

Line 11: direct the program flow to the next guest instruction if the branch is not taken. 45408 equals to b160 in hex.

Instructions in a guest basic block are translated into LLVM IR and stored in a container called **translation block**. In mc2llvm, the translation block is the unit of translation. A graph of how the translation block is formed is presented in figure 4.6. A translation block is in fact a LLVM function. An example of a translation block is shown in figure 4.7. In figure 4.7, the LLVM function returns a 32 bit integer and it has an argument which is a structure called ThreadCtx. The structure contains 4 members - a 32-bit integer, an array of 32-bit integers, 3 64-bit integers. The structure is in fact a subset of the thread context mentioned in section **Emulating Threads**. The most important member among the 4 members is the second one. Let's

```
1  define i32 @L_000080d4_({ i32, [32 x i32], i64, i64, i64 }* %ThreadCtx) alwaysinline {
2  L_00000000_:
3    %0 = getelementptr inbounds { i32, [32 x i32], i64, i64, i64}* %ThreadCtx, i32 0, i32 3
4    store i64 1885165184, i64* %0
5    %1 = getelementptr inbounds { i32, [32 x i32], i64, i64, i64 }* %ThreadCtx, i32 0, i32 4
6    store i64 -1, i64* %1
7    %2 = load i64* inttoptr (i64 1885165192 to i64*)
8    %3 = icmp eq i64 %2, 1
9    ...
10   ret i32 %10
11 }
```

Figure 4.7: A snapshot of translation block in LLVM IR

call it A[32]. It corresponds to the guest architecture state. We map guest registers r0 - r15 to A[0] - A[15] and N,Z,C,V to A[17] to A[20]. The rest elements in the array either is not used or is used for debugging.

The return value of the LLVM function is the address of the next guest instruction after the translation block is executed. In the translation block, instruction accessing to the guest architecture would do operation on the second member of the ThreadCtx. When a translation block is formed, a lot of LLVM IR seems redundant. We use LLVM optimizer to eliminate it. Finally, we utilize LLVM JIT to emit host instructions. The procedure is presented in figure 4.8.

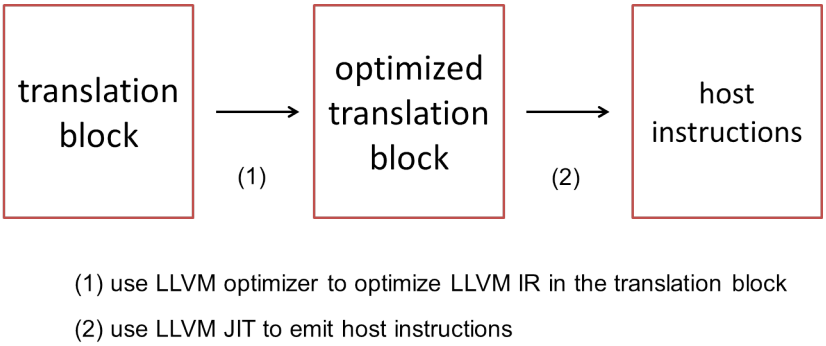

Figure 4.8: Turning LLVM IR to host instructions

4.5 Address Mapping Table and Shared Code Cache

A translation block keeps two addresses. One is the guest address of the original guest basic block and the other is the address of its host instructions. The system keeps the emitted code in a code cache to avoid repeated translation and stores the mapping of the guest address and the host address in the address mapping table. Thus, before translating at a guest address, the system looks up the table to find if its corresponding host instructions exist. The relationship between address mapping table and code cache is presented in figure 4.9. The original data structure of the address mapping table in mc2llvm is a C++ STL map. We modify the data structure to a hash table and it is not just a single address mapping table anymore. In the current implementation, there are two types of tables: a **global-shared table** and a **thread-private table** per emulating thread. The global-shared table

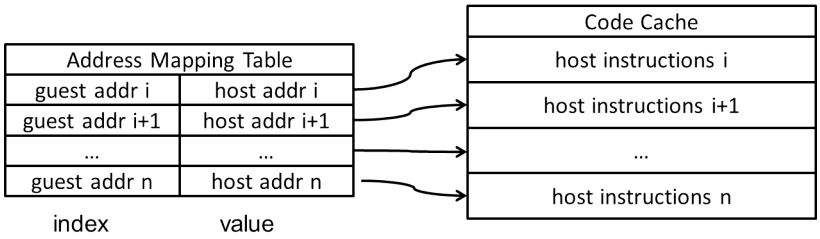

Figure 4.9: Address mapping table and code cache

contains the complete address mappings. The thread-private table contains the mappings recently used by the emulating thread. The global table [4] contains 2^{14} entries. Each entry points to a list. The list contains host addresses that are hashed to the same value. The thread table also contains 2^{14} entries. Each entry contains a host address. The global-shared table and the thread-private table all use the same hash function. The hash function is $hash(x) = (x >> 2)\&((1 << 14) - 1)$. Both types of tables are graphically presented in figure 4.10.

The global-shared and the thread-private table collaborate to complete the access operation and it is presented in figure 4.11.

4.6 System Call Handler

mc2llvm is a process-level binary translator. It has to specially handle the system call instructions (svc in ARM). By convention in ARM processors, the system-call number is in r7 and the parameters are in r0 - r5 when the svc instruction is executed. To emulate system calls, mc2llvm generates

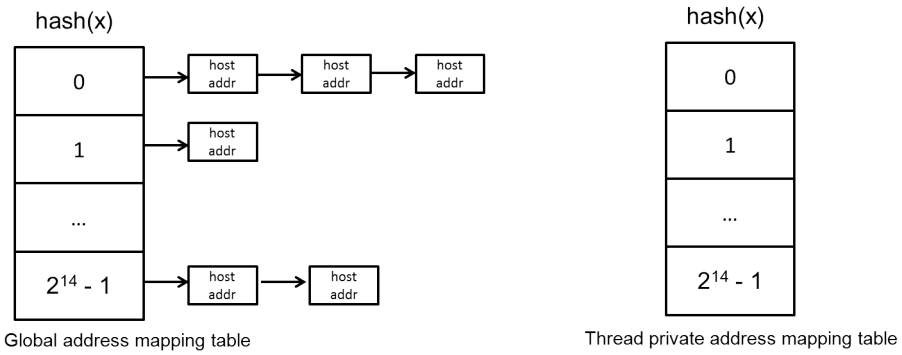

Figure 4.10: Data structure of the global-shared and thread-private address mapping table

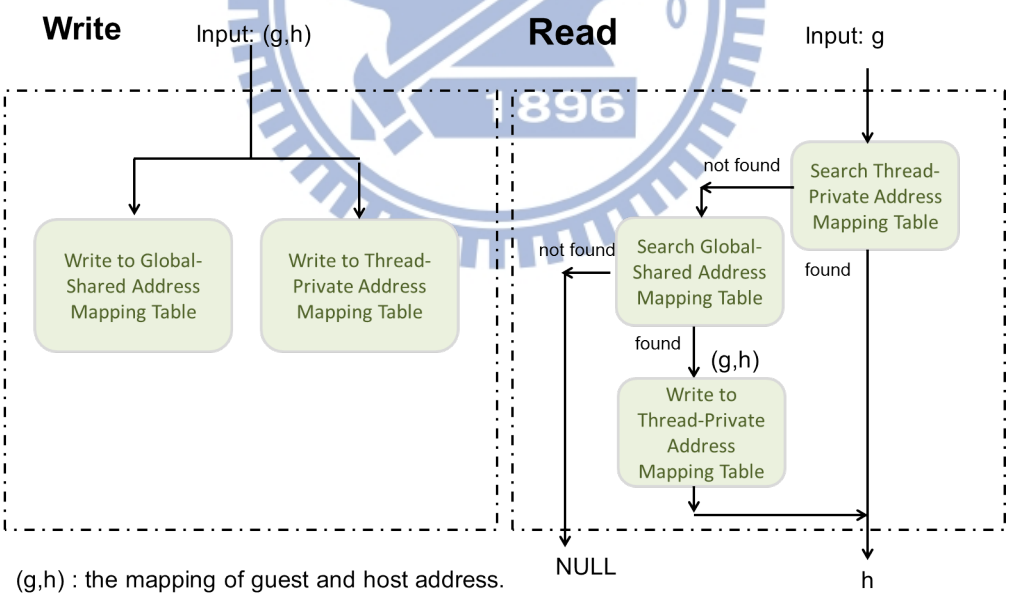

Figure 4.11: Access to address mapping tables

```
1   %11 = getelementptr inbounds [32 x i32]* %8, i32 0, i32 7
2   %12 = load i32* %11
3   %13 = getelementptr inbounds [32 x i32]* %8, i32 0, i32 0
4   %14 = load i32* %13
5   %15 = getelementptr inbounds [32 x i32]* %8, i32 0, i32 1
6   %16 = load i32* %15
7   %17 = getelementptr inbounds [32 x i32]* %8, i32 0, i32 2
8   %18 = load i32* %17
9   %19 = getelementptr inbounds [32 x i32]* %8, i32 0, i32 3
10  %20 = load i32* %19
11  %21 = getelementptr inbounds [32 x i32]* %8, i32 0, i32 4
12  %22 = load i32* %21
13  %23 = getelementptr inbounds [32 x i32]* %8, i32 0, i32 5
14  %24 = load i32* %23
15  %25 = call i32 (i32, ...)* @mc2llvmSyscall(i32 %12, i32 %14,
16          i32 %16, i32 %18, i32 %20, i32 %22, i32 %24)
17  store i32 %25, i32* %13
```

Figure 4.12: How svc instruction is translated into LLVM IR

LLVM instructions to load guest r0 - r5 and r7 registers into LLVM variables and translates svc to an instruction that calls the wrapper function mc2llvmSyscall. Finally, it generates an instruction that saves the return value from the wrapper function in r0. The procedure is presented in figure 4.12. The system translates svc instruction into the following steps:

1. Line 1 - 14, load r0 - r5, r7 from the guest architecture state
2. Line 15, 16, call a helper function to handle the system call with parameters r0 - r5 and r7
3. Line 17, store the return value of the wrapper function to r0.

The internals of the wrapper function (mc2llvmSyscall) are shown in figure 4.13. The original mc2llvm does not support system calls used in multi-

```
int mc2llvmSysCall(int syscall_num,int r0,int r1,int r2,int r3,int r4,int r5){
    int ret;
    switch( syscall_num)  {
            case Linux::getrlimit:{
                struct rlimit t1;
                ret = err(getrlimit(r0,&t1));
                host_to_guest_rlimit(&t1,(struct guest_rlimit*)r1);
            break;
            case Linux::write:
                ret = err( write( r0,r1,r2 );
            break;
            case Linux::brk:
                    ret = mc2llvm_brk( .. );
             break;
              ...
    }
    return  ret;
}
```

Figure 4.13: Internals of system call handler

threaded programs such as clone, set_tid_address, futex and etc. In multi-threaded programs, new threads are created with the clone system call. Mutex, semaphores are implemented with the futex system call. On the x64 host machine, we treat variables of the long or pointer type used in system calls as 32 bits. For example, the prototype of getrlimit in figure 4.13 is **int getrlimit(int resource, struct rlimit *rlim);**, its second argument is a pointer type. In our case, we create a temp variable t1 and pass t1 to getrlimit then we pass the values in t1 to r1.

4.7 Emulate multi-threaded programs

A binary translator able to emulate multi-threaded programs needs to consider the following things:

- How to emulate thread creation and termination?
- How to emulate atomic operations?
- How to access TLS (thread local storage)?
- What are shared in the translation system?

4.7.1 How to emulate thread creation and termination?

A thread is created by Linux clone system call. We do the following things to emulate clone system call.

1. Create a new thread called emulating thread by pthread_create
2. Copy the guest architecture state of the parent thread to the child thread.
3. Set guest stack registers to newsp. (refer to the parameter newsp passed in clone system call in Chapter 3)
4. Set guest thread-local storage of the child thread.

5. Set r0 in the child and parent threads. The return value of the parent thread is the id of the child thread while the return value of the child thread is 0. On the ARM architecture, the return value is saved in r0.

6. Set child id to parent tidptr and child_tidptr if the CLONE_PARENT_SETTID and CLONE_CHILD_SETTID is on.

7. Keep a copy of child_tidptr for the new thread if the CLONE_CHILD_CLEARTID is on.

8. Both threads do emulation from the instruction after svc instruction.

A thread is terminated when mc2llvm encounters an exit or exit_group system call. The exit system call terminates the current thread. But before terminating itself, it should clear the location child_tidptr and do a futex wakeup if the thread is created with CLONE_CHILD_CLEARTID on. exit_group system call terminates all the emulating threads.

4.7.2 How to emulate the atomic operations?

We observe that our benchmarks do not use atomic instructions in user programs except one case. There are two reasons. First, regular programs would not contain atomic instructions directly. Instead, those atomic instructions are embedded in the system call such as futex. Second, mc2llvm only support ARMv5 instructions. Such architecture does not have atomic instructions like ldrex and strex. We found that there is only one atomic operation used

in all benchmarks. This is kernel supported function: **__kuser_cmpxchg**. We emulate what the __kuser_cmpxchg does and guard the actions with a global lock. The function __kuser_cmpxchg does compare-and-swap and sets the condition flag C if the operation is successful.

4.7.3 How to access TLS base address?

On the ARM architecture, TLS base address is read by **__kuser_get_tls** and is set by ARM-private system call **set_tls**. The declaration of __kuser_get_tls is shown as follows:

```
typedef void*(__kuser_get_tls_t)(void);
#define __kuser_get_tls(*(__kuser_get_tls_t*)0xffff0fe0)
```

To emulate the TLS base, mc2llvm uses a thread-local variable to keep the TLS base since each thread has its only TLS base.

4.7.4 What are shared in the translation system?

We design mc2llvm to have

- **a shared instruction translator**
- **two types of address mapping tables**(one is global-shared and the other is thread-private)
- **a shared code cache**

A single instruction translator is enough to handle translation request from multiple emulating threads because each guest basic block is translated only once. Host address lookup is done by first searching in the thread-private table followed by searching in the global-shared table if necessary. The thread-private table can alleviate the synchronization overhead incurred by accessing to the global-shared table because accessing to global-shared table is protected by a lock. A shared code cache can be shared among all the emulating threads so no duplicate translation is needed.

4.8 32-bit ARM on 64-bit x64 machine

Originally, our supported host machine is x86_32 machine. If we want to support x64 one, some tuning of the translation system is needed. In the following, we describe what we have done to make mc2llvm able to run on x64 machine.

4.8.1 Memory Address Space

we need to make sure all the memory used by the guest binary code is allocated within 32-bit memory address space. To complete this task, guest stack and heap are allocated by host mmap system call with flag MAP_32BIT. Besides, runtime memory allocation such as guest mmap system call is also allocated with mmap system call with flag MAP_32BIT. argc, argv, envp ,auxv put onto the guest stack should be also reachable by 32-bit pointer.

4.8.2 ABI Size of System Call Parameter

On the 32-bit ARM architecture, long and pointer type is 32-bit address. When we executes system call handler, we need to notice those guest system call parameters. For example,

```
struct timespec {
                time_t   tv_sec;        /* seconds */
                long     tv_nsec;       /* nanoseconds */
            };
```

timespec is a structure used by clock_gettime system call. For guest architecture, the memory size of the structure is 8 bytes while for host architecture like x64, it is 16 bytes. We need to treat those parameters using long or pointer type as 32 bits.

Chapter 5

Optimization

In mc2llvm [13], it already has optimization of block chaining and switch table reconstruction. We describe new optimizaton in this section. The performance enhancing mechanism focuses on enlarging the translation unit and having better code quality for the emulating instructions. To utilize the multi-core architecture, the system has extra threads called optimizing threads. They cooperate with emulating threads to do trace compilation. Emulating threads do trace selection and push the collected traces into a queue called trace queue. Optimizing threads pop a trace from the queue and emit instructions for the trace.

5.1 LLVM IR optimizations

LLVM provides a fleet of LLVM IR optimizations. The system uses four LLVM IR optimizations to remove redundant instructions on the LLVM IR.

LLVM function name	effect
createPromoteMemoryToRegisterPass()	promote memory to registers
createEarlyCSEPass()	eliminate trivial redundant instructions
createCFGSimplificationPass()	remove redundant basic blocks and merge basic blocks together if possible
createGVNPass()	remove redundant instructions

Figure 5.1: The chosen LLVM IR optimizations

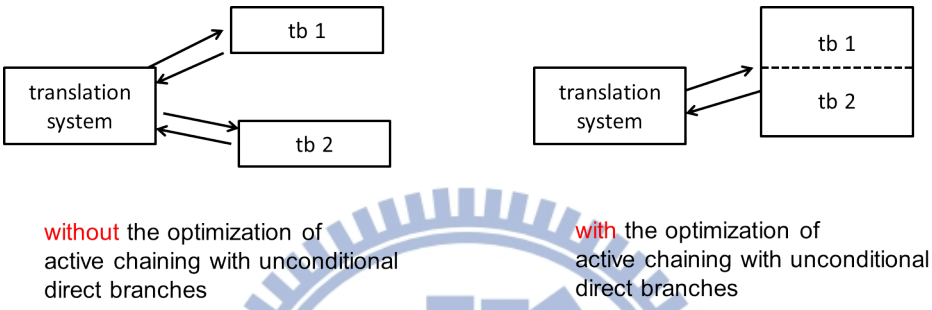

Figure 5.2: Chaining with unconditional direct branches

The chosen optimizations are presented in figure 5.1.

5.2 Active Chaining with Unconditional Direct Branches

The emulating thread keeps translation when it meets an unconditional direct branch [8] by eliding the unconditional direct branch instruction and translating at the destination of the unconditional branch instruction. The effect of this optimization is presented in figure 5.2. Instead of translating the guest direct branch instruction to LLVM IR return instruction, the translate skips the instruction, and continues to do translation at the target of the un-

conditional direct branch instruction [8]. This technique increases the scope of the translation block. As a result, more optimization opportunity is made. This method elimiates all guest unconditional direct branch instructions with an side effect on increasing a little memory usage.

5.3 Trace Compilation

Trace compilation is a dynamic optimization technique that selects the hot path and emits the codes for it at run time. We think there are 2 main issues. One is trace selection and the other is trace generation.

5.3.1 Trace Selection

Trace selection has been discussed by many papers [9], [20], [23], [24]. Our trace selection is mainly based on them. For trace selection, we insert the profiling codes at the beginning of each translation block. Figure 5.3 shows the profiling codes which are LLVM call instructions to helper function (trace_helper1 and trace_helper2).

The functions trace_helper1 and trace_helper2 cooperate to do trace selection. How trace is selected is shown in figure 5.4.

5.3.2 Trace Generation

In the system, trace generation is done by optimizing threads and the system has 3 optimizing threads by default. The trace selected by emulating

```
1  define i32 @L_000097a4_({ i32, [32 x i32], i64, i64, i64 }* %ThreadCtx) alwaysinline {
2  L_00000000_:
3    %0 = getelementptr inbounds { i32, [32 x i32], i64, i64, i64 }* %ThreadCtx, i32 0, i32 3
4    store i64 1910052688, i64* %0
5    %1 = getelementptr inbounds { i32, [32 x i32], i64, i64, i64 }* %ThreadCtx, i32 0, i32 4
6    store i64 -1, i64* %1
7    %2 = load i64* inttoptr (i64 1910052696 to i64*)
8    %3 = icmp eq i64 %2, 1
9    br i1 %3, label %L_00000000_1, label %L_00000000_2
10
11 L_00000000_1:                                        ; preds = %L_00000000_
12   call void @trace_helper2()
13   %4 = load i32 ({ i32, [32 x i32], i64, i64, i64}*)** @traceGV6
14   %5 = call i32 %4({ i32, [32 x i32], i64, i64, i64}* %ThreadCtx)
15   ret i32 %5
16
17 L_00000000_2:                                        ; preds = %L_00000000_
18   call void @trace_helper1()
19   %6 = getelementptr inbounds { i32, [32 x i32], i64, i64, i64 }* %ThreadCtx, i32 0, i32 1
20   %7 = getelementptr inbounds [32 x i32]* %6, i32 0, i32 16
21   store i32 38820, i32* %7
22   ...
23 }
```

Figure 5.3: Insert profiling codes for selecting traces

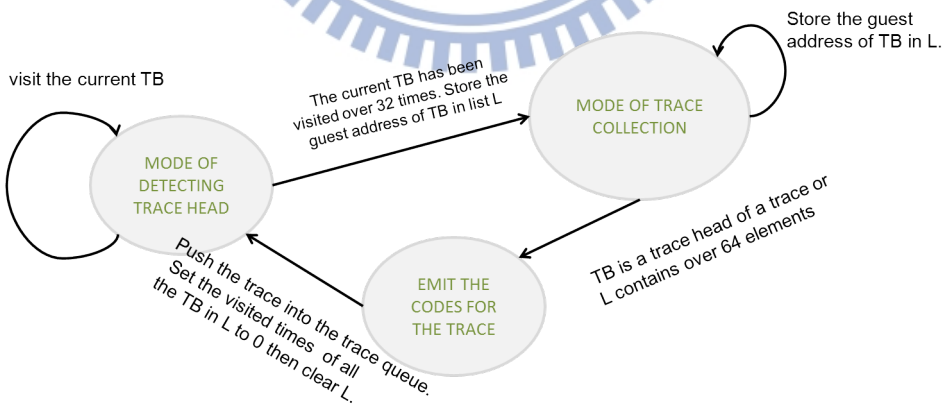

Figure 5.4: The procedure of trace selection

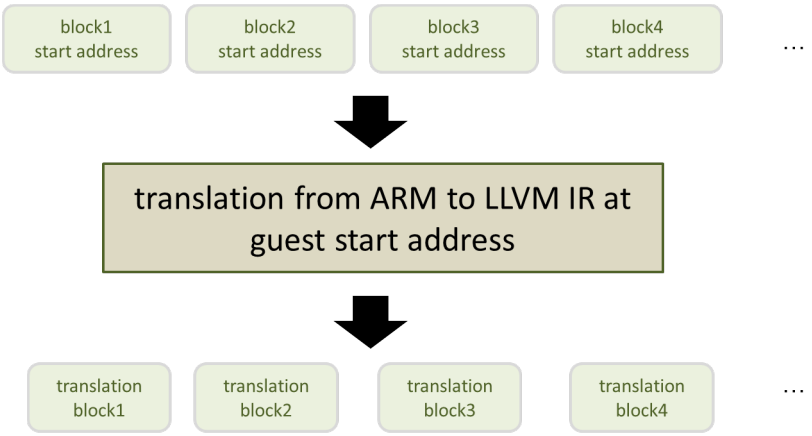

Figure 5.5: Generate each translation block at the guest address

threads would be pushed into a concurrent queue [26]. Each optimizing thread periodically tries to pull the trace from the queue. The trace in the queue is actually a vector of integer which stores the guest addresses of the collected translation blocks, we can see that in figure 5.5. Once the optimizing thread successfully pulls a trace from the queue, the optimizing thread does instruction translation based on each guest address in the vector one after another. Finishing translation, the optimizing thread links neighboring translation blocks together by adding guard instructions. This is shown in figure 5.6. This includes testing if the next block is exactly the block of the next executing block. If the test fails, the program would execute the exit stub which returns the address of the next guest instruction.

Before JIT the trace, we modifies those LLVM ret instructions which have return value that equals the guest address of the current trace head to branch instructions that branch to the trace head. This is shown in fig-

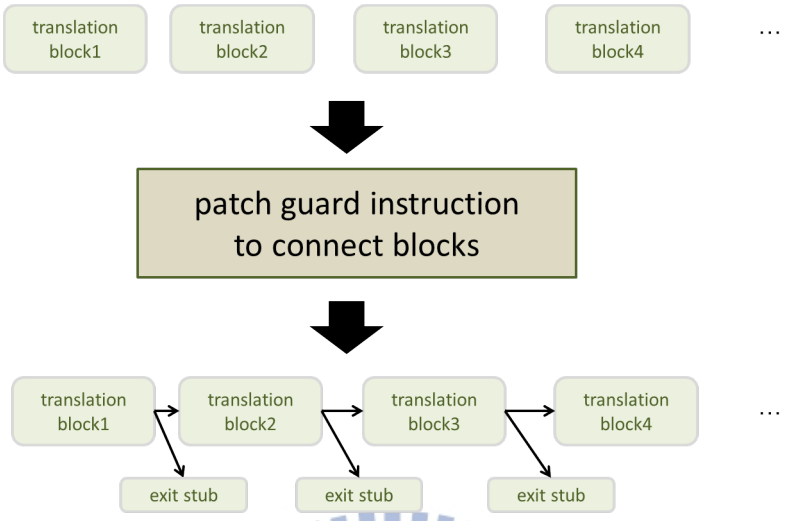

Figure 5.6: Link translation block together

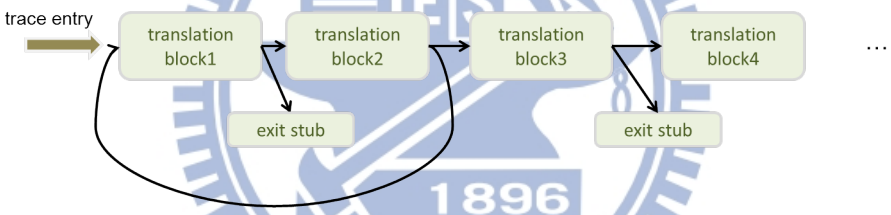

Figure 5.7: Optimize the trace

ure 5.7. Because the system has optimizing threads to do trace generation, the emulating thread can keep on emulation while the optimizing threads are generating the trace. Thus, our runtime overhead of trace compilation is only executing the trace profiling code and inserting the selected trace into the queue.

When the optimizing thread emits the host codes for the trace. it modifies a variable in the translation block from false to true. This variable indicates

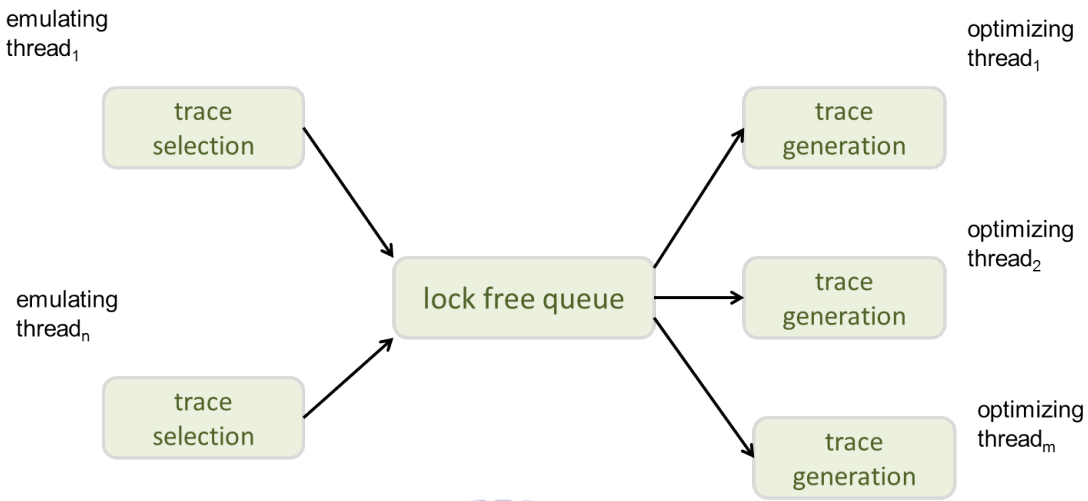

Figure 5.8: The communication between emulating and optimizing threads

whether the current translation block is a trace head. That variable is set to false in the beginning and each translation block keeps its own variable. In figure 5.3, testing the variable is shown in line 8 and 9. Line 13 loads the starting address of the instructions of the trace. Line 14 executes the trace. Line 15 returns the address of the next guest instruction.

The overall trace compilation is shown in figure 5.8. In our design, all generated traces are also shared among emulating threads.

5.4 Filling empty thread-private table

This optimization comes with the design of two-level address mapping tables (one is global-shared and the other is thread-private address mapping table). When an emulating thread is created, its thread-private table is empty. The

emulating thread that creates the new thread copies its own thread-private table to the thread-private table of the new thread. This method benefits when emulating threads are performing similar tasks. The thread-private table is not large so copying the table just takes a little time.

Chapter 6

Synchronization

In this section, we point out the portions in our binary translator that require synchronization and discuss how we handle them.

6.1 Instruction Translator

The instruction translator is responsible for translating guest binary code to LLVM IR and turning LLVM IR to host binary code. There is only one instruction translator in our system and it is shared among emulating threads. In terms of implementation, the translator is protected by pthread_mutex_lock. This part is not a performance bottleneck due to each block only being translated once.

6.2 Access to Global Mapping Table

One thread may write to address mapping table while some threads may read from the global mapping table. To deal with the simultaneous read and write operations, the mapping table is crafted to be concurrent [4].

6.3 Trace Queue

Emulating threads and optimizing threads cooperate with trace compilation, the trace selected by emulating threads is pushed into the trace queue and poped by the optimizing threads to do code generation. The optimizing threads periodically try to pop values from the trace. In our observation, the queue is frequently accessed. To build a high performance queue that can be accessed by many threads at the same time, we adopt the algorithm in [26].

6.4 Repeated Trace Detection

In the system, all emulating threads are able to do trace selection. Chances are that they may select the same trace but emulating threads would not detect that because they just pass the collected trace to the trace queue. The detection of the repeated trace is done by the optimizing thread. Once the optimizing thread obtains a trace from the queue, it searches a C++ set data structure protected by lock to find if the trace has been generated. If it does, the optimizing thread discards the trace. Otherwise, it does trace

generation.

6.5 Translation of __kuser_cmpxchg

This is a function fixed at specific address by Linux kernel. Its main task is to do atomic compare and swap. We emulate this function with a global lock.

6.6 LLVM Library

LLVM has its own mechanism to protect access to its own data structure from multiple threads but that mechanism should be turned on by calling llvm_start_multithreaded().

Chapter 7

Experiment Result

7.1 Experimental Setup

ARM static linked binaries are fed as the input data for the translator. In figure 7.1, the information on experiemental setup is presented. We run the translator on the 48-core x86_64 host machine. Each core is 2.1 GHz and the memory is 48GB. The benchmark we use is SPEC omp 2001 [25] and we use the provided test data set as the input for the benchmark. All the benchmark is compiled by GNU gcc or gfortran 4.3.2 with flag -O2 -static -fopenmp. Among 11 benchmark in the SPEC omp, we fail to compile galgel, and fail to emulate ammp. Others are successfully cross compiled and emulated correctly. We exclude fma3d in the experiment because it has short emulation time (< 10s). To build mc2llvm, we have to install LLVM 3.2 library beforehand. Before emulating any program, we set vm.mmap_map_addr="4096"

	Description
benchmark	SPEC ompm 2001 with test data set
host machine	48 core x86_64 machine
host OS	Linux Debian 3.2.35-2
compilation flags for mc2llvm	-O2
compiler for mc2llvm	clang-3.2
Library	LLVM-3.2
CPU	AMD Opteron™ Processor 6172 , 2.1 GHz
memory size	48GB
ARM cross compiler	gcc, gfortran 4.3.2
compilation flag for ARM cross compiler	-O2 -static -fopenmp
ARM board	Board name: Origen CPU : Samsung Exynos 4 Quad Cortex-A9 core 1.4 GHz DRAM: 1GB
QEMU	Version: 0.12.3

Figure 7.1: Information on Experimental Setup

on the command line.

7.2 Parallel Emulation

In this part, we set each benchmark to use different number of threads by specifying an environment variable OMP_NUM_THREADS and run it on the translator. The result is shown in two ways. One is in figure 7.2 which shows the exact emulation time and the other is in figure 7.3 which shows the ratio. In the emulation, all the benchmarks seem to have decreasing running time when more threads are used. Two benchmarks gafort and apsi do not meet the expectation, but this condition is almost the same when the programs are run in the native machine.

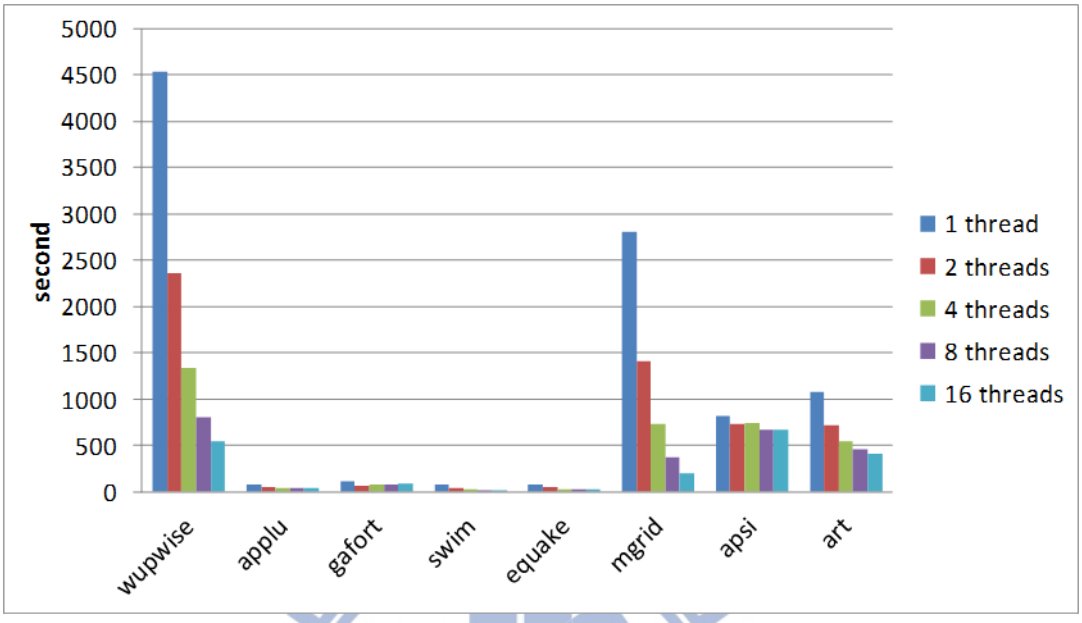

Figure 7.2: Emulation with different number of guest threads

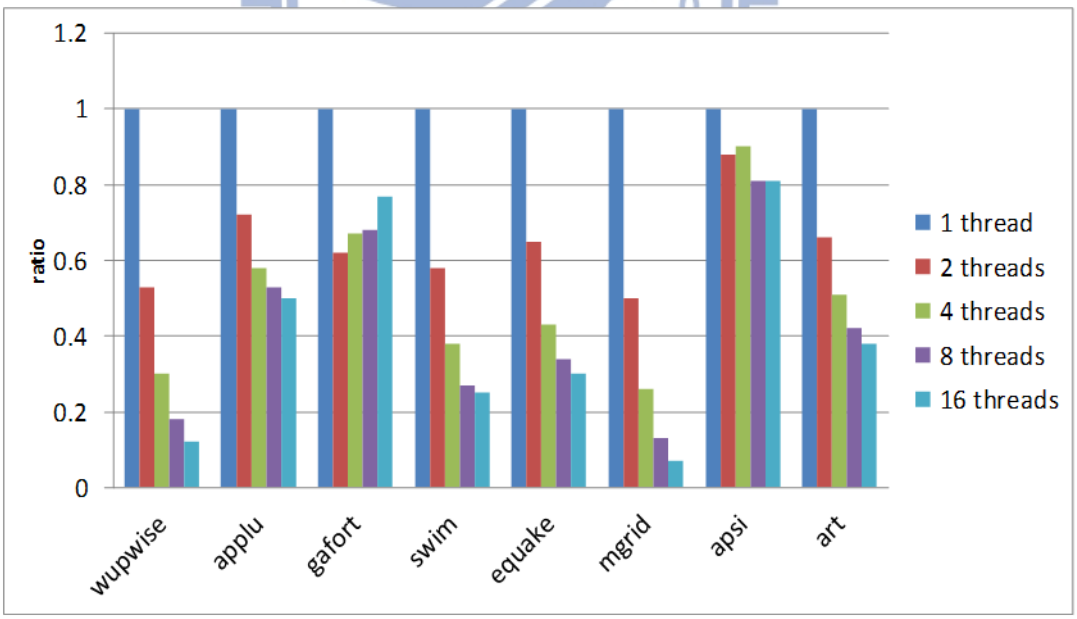

Figure 7.3: The emulation time in ratio when the benchmarks are using different number of threads

benchmark	JIT time	translation time	total emulation time
wupwise	0.6%	1.1%	550s
applu	11.3%	22.2%	55.7s
gafort	4.2%	7.6%	113s
swim	14.9%	25.8%	24.8s
equake	10.1%	17.9%	34.5s
mgrid	2%	3.4%	215s
apsi	1.9%	3.7%	650s
art	0.6%	1%	361s

Figure 7.4: Time analysis

7.3 Time Usage

Figure 7.4 shows the time usage when we emulate programs with 16 guest threads. Translation time represents time spent to decode guest instructions and translate them into LLVM IR and the JIT time represents time spent in optimization of LLVM IR and turning LLVM IR into host instructions. Total time is the time to emulate the benchmark from program startup to program termination. Translation and JIT time account for only a little proportion of the total time when the total time is large. This is because all the guest basic blocks are translated once. The total time is measured by Linux **time** command and the translation and JIT time are measured by inserting the function **clock_gettime** in our system. All the measurement is done separately. In average, JIT time accounts for 5.7% and translation time accounts for 10.3%. The rest of the time is mostly occupied by executing codes in the code cache.

benchmark	number of block	number of trace	hit rate of thread table lookup	size of block code cache	size of trace cache	number of unconditional direct branch	number of conditional branch	total translated instruction
wupwise	3429	412	99.9%	1.8MB	0.76MB	2285	2618	31929
applu	6108	863	98.9%	5.41MB	2.58MB	6643	5220	131571
gafort	4861	1180	99.8%	2.92MB	3.18MB	4049	4044	52969
swim	4076	414	99.9%	2.53MB	0.66MB	3400	3431	43313
equake	2869	248	99.9%	1.65MB	2.24MB	2012	2317	28188
mgrid	3239	1335	99.9%	1.57MB	4.19MB	1737	2573	27656
apsi	7569	1477	99.9%	5.2MB	16.91MB	7558	6404	98164
art	1987	376	99.9%	0.95MB	2.01MB	1111	1508	16629

Figure 7.5: Various statistics

7.4 Various Statistics

Rigure 7.5 shows various statisics. We will explain each piece of statistics in the following:

1. **number of block**: It means the number of translated guest basic block.

2. **number of trace**: It means the number of generated trace. A trace usually takes less than 0.1s to generate and we have 3 threads to do trace generation. That means the trace queue is usually empty.

3. **hit rate of thread table lookup**: In our design, the thread-private table has 2^{14} entries. Since the number of the block is smaller than the table entries, the emulating thread almost hit in the thread-private table all the time.

4. **size of block code cache**: To our surprise, the size of the shared code cache is quite small. If our code cache is private then the size of the

cache would almost N times larger than the shared code cache where N is the number of the emulating threads.

5. **number of unconditional direct branch**: This includes unconditional direct branch and call instructions. The number implies how much the emulation speed can benefit from the optimization of active chaining with unconditinal direct branch. It accounts for 7% in average among the total translated instructions.

6. **number of conditional branch**: It means the number of translated conditional branch instruction. The number implies how much the emulation speed can benefit from the optimization of block chaining. It accounts for 7% in average among the total translated instructions.

7. **total translated instruction**: It means the number of translated guest instruction.

7.5 The Impact of Trace Compilation

Trace compilation is not essential in binary translation but it is a technique that may have a chance to speed up emulation. In figure 7.6, we can see that some benchmarks benefit from trace compilation while some do not. Only the case when the speedup gained from executing traces outweight the slowdown of profiling traces can the emulation become faster. In our work, profiling traces can be the burden to the emulation. Profiling codes for traces

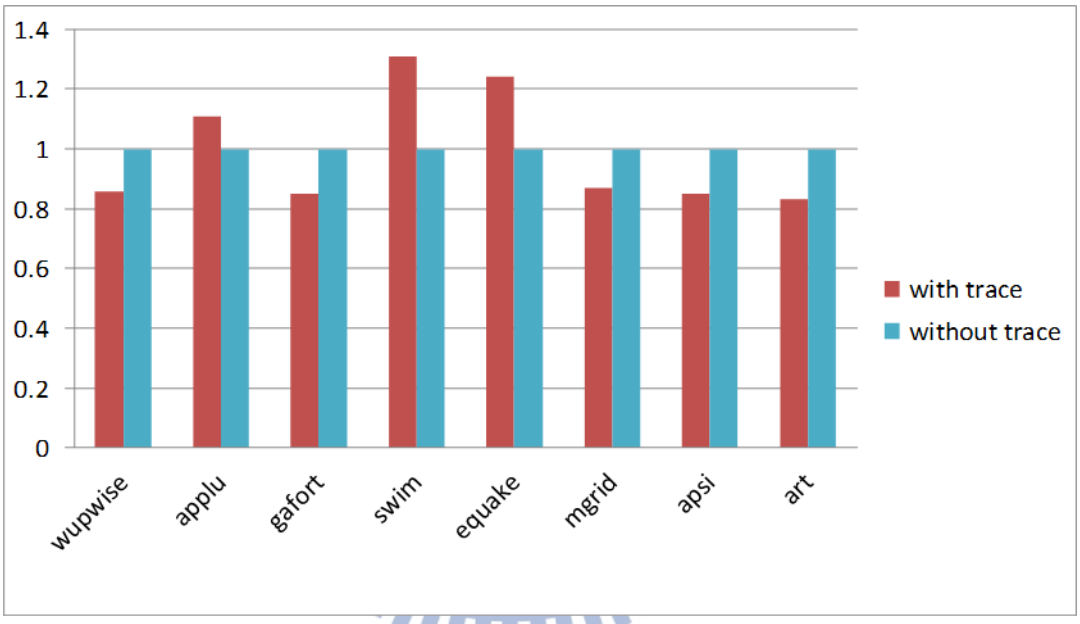

Figure 7.6: Emulation time in ratio with/without trace compilation when programs are using 16 guest threads

are inserted into each translation block but not the trace. Thus, emulating threads have to execute profiling codes when they are executing the codes in the translation block but they do not have to execute profiling codes when they are executing the codes in the trace. If a benchmark has shorter emulation time when we use trace compilation. This may indicate that most of the time emulating threads are executing the traces. But if trace compilation does not favor the benchmark, with high probability that the formed traces have problems like early exit and trace separation so emulating threads spend their time executing host instructions in translation blocks but not traces. With trace compilation, in the best case we gain 17% performance speedup from the benchmark art and in the worst case we increase 31% overhead from the benchmark swim. The execution time we collect is measured by Linux **time** command.

7.6 Comparison with QEMU

QEMU is a well-known dynamic binary translator. In figure 7.7, we compare mc2llvm with QEMU. QEMU fails to run applu, gafort. QEMU seems to encounter the deadlock when it emulates applu and it encounters segmentation fault immediately when it emulates gafort. mc2llvm is faster than QEMU when emulating programs with 8 guest threads. We profile by perf that QEMU performs a lot of locks when emulating the benchmarks. In average, mc2llvm is 8.8X faster than QEMU.

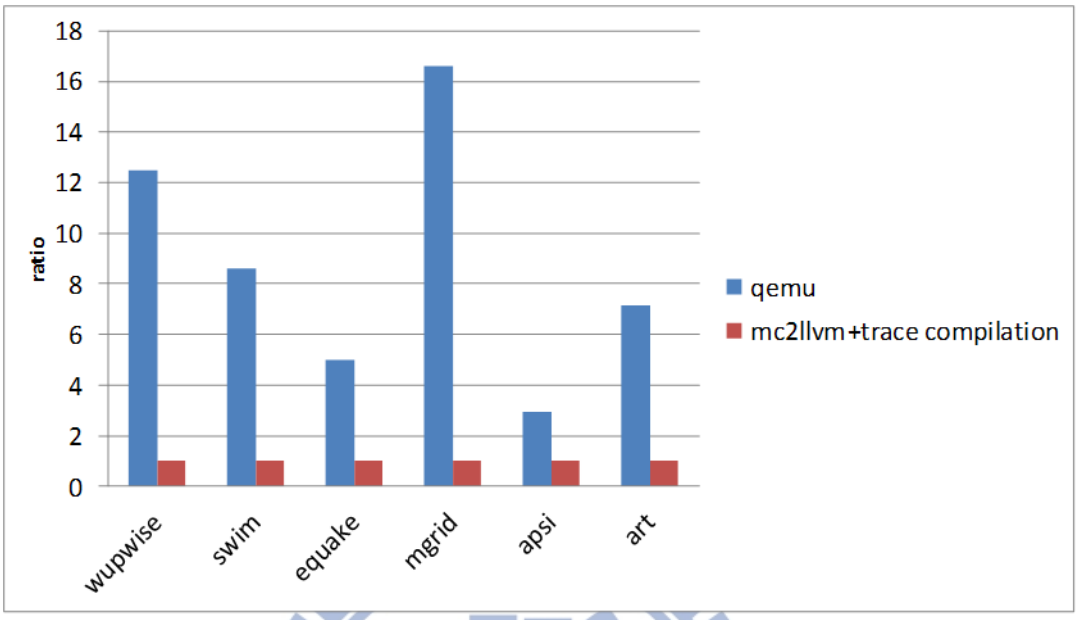

Figure 7.7: Comparison with QEMU

7.7 Comparison with Native Machine

We compare our result with an ARM machine named **origen** [27]. Its basic description is in figure 7.1. origen is shipped with Linaro Android OS and is a 4-core ARM machine. Thus, we run each benchmark with 4 threads. The result is shown in figure 7.8. origen encounters segmentation fault when it runs wupwise and gafort. The experiment aims at realizing the performance gap between programs run in native machine and emulated by the dynamic binary translator. In average, mc2llvm is $5.45X$ slower than origen.

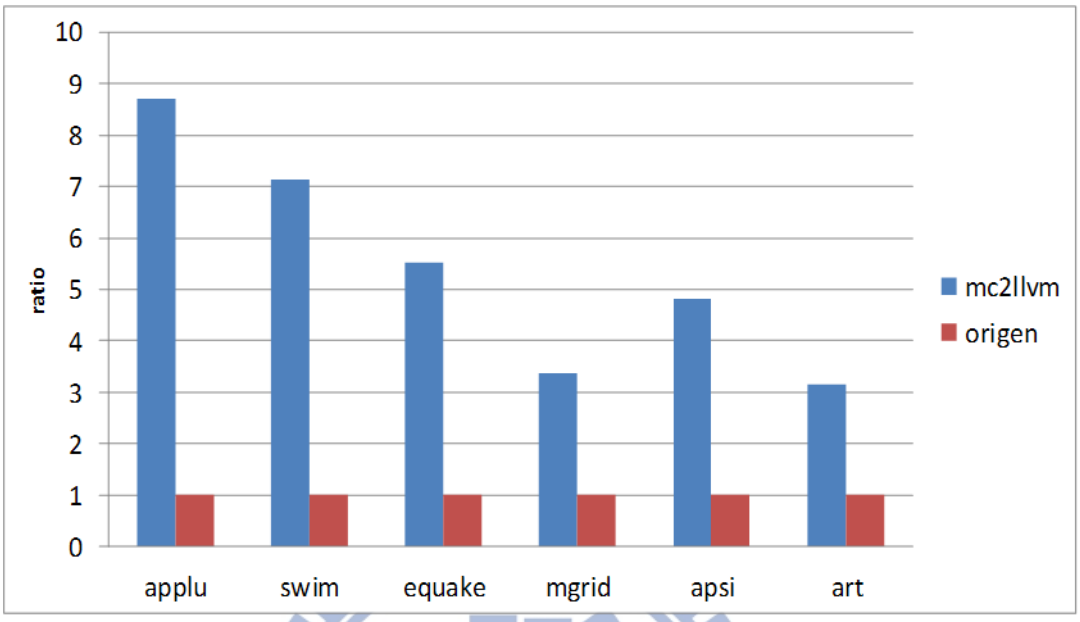

Figure 7.8: Comparison with ARM machine

7.8 Future Work

1. Currently, LLVM JIT is not able to map LLVM global variable to specific host registers. Thus, we cannot do a better register mapping by mapping guest registers to host registers. This could hurt the performance a lot due to extra load and store instructions.

2. Our trace profiling incurs a lot of overhead so that the performance would go down when the emulating threads are not executing the traces. A better trace profiling technique is required.

Chapter 8

Conclusion

We turn a binary translator mc2llvm[13] which is only able to emulate sequential programs into the one able to emulate multi-threaded programs. We reduce the memory usage by using the shared code cache. The synchronization overhead is reduced as much as possible by shortening the lock section as we can and using the concurrent data structure. Trace compilation chooses hot paths which are selected at runtime by emulating threads and generated by optimizing thread. In the work, we extend mc2llvm to run on the x64 host machine and fully utilize the underlying multicore architecture to speed up emulation. In average, mc2llvm is 8.8X faster than QEMU when emulating programs with 8 guest threads, mc2llvm is 5.45X slower than origen [27] when emulating programs with 4 guest threads and mc2llvm gains 17% performance with trace compilation in the best case.

Bibliography

[1] Kim Hazelwood, Greg Lueck, Robert Cohn, *Scalable Support for Multithreaded Applications on Dynamic Binary Instrumentation Systems*, ISMM'09 June 19-20, 2009, Dublin, Ireland.

[2] Zhaoguo Wang, Ran Liu, Yufei Chen, Xi Wu, Haibo Chen, Weihua Zhang, Binyu Zang, *COREMU: A Scalable and Portable Parallel Full-system Emulator*, in PPoPP'11, February 12-16, 2011, San Antonio, Texas, USA.

[3] Derek Bruening, Vladimir Kiriansky, Timothy Garnett, and Sanjeev Banerji, *Thread-Shared Software Code Caches*, in CGO '06 Proceedings of the International Symposium on Code Generation and Optimization Pages 28-38

[4] Maged M. Michael, *High performance dynamic lock-free hash tables and list-based sets* , in SPAA '02 Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures Pages 73 - 82

[5] Jason Mars, Mary Lou Soffa, *MATS: Multicore Adaptive Trace Selection*, in STMCS'08 Third Workshop on Software Tools for MultiCore Systems

[6] Jiun-Hung Ding, Po-Chun Chang, Wei-Chung Hsu, Yeh-Ching Chung, *PQEMU: A Parallel System Emulator Based on QEMU*, in ICPADS '11 Proceedings of the 2011 IEEE 17th International Conference on Parallel and Distributed Systems Pages 276-283

[7] Ding-Yong Hong , Chun-Chen Hsu, Pen-Chung Yew, Jan-Jan Wu, Wei-Chung Hsu, Pangfeng Liu, Chien-Min Wang, Yeh-Ching Chung, *HQEMU: a multi-threaded and retargetable dynamic binary translator on multicores*, in CGO '12 Proceedings of the Tenth International Symposium on Code Generation and Optimization Pages 104-113

[8] Swaroop Sridhar, Jonathan S. Shapiro, Eric Northup, Prashanth P. Bungale, *HDTrans: an open source, low-level dynamic instrumentation system*, in VEE '06 Proceedings of the 2nd international conference on Virtual execution environments Pages 175-185

[9] Hiroshige Hayashizaki, Peng Wu, Hiroshi Inoue, Mauricio J. Serrano, Toshio Nakatani, *Improving the performance of trace-based systems by false loop filtering*, in ASPLOS XVI: Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems

[10] Jason D. Hiser, Daniel W. Williams, Wei Hu, Jack W. Davidson, Jason Mars, Bruce R. Childers , *Evaluating indirect branch handling mecha-*

nisms in software dynamic translation systems, in Transactions on Architecture and Code Optimization (TACO) , Volume 8 Issue 2

[11] A. Jeffery, *Using the LLVM compiler infrastructure for optimised, asynchronous dynamic translation in QEMU*, Master's thesis, University of Adelaide, Australia, 2009.

[12] Chun-Chen Hsu, Pangfeng Liu, Chien-Min Wang, Jan-Jan Wu, Ding-Yong Hong, Pen-Chung Yew, Wei-Chung Hsu, *LnQ: Building High Performance Dynamic Binary Translators with Existing Compiler Backends*, in ICPP '11: Proceedings of the 2011 International Conference on Parallel Processing

[13] Bor-Yeh Shen, Jyun-Yan You, Wuu Yang, Wei-Chung Hsu, *An LLVM-based Hybrid Binary Translation System*, 7th IEEE International Symposium on Industrial Embedded Systems (SIES'12), Karlsruhe, Germany, June 20-22, 2012

[14] Bor-Yeh Shen, Jiunn-Yeu Chen, Wei-Chung Hsu, Wuu Yang, *LLBT: an LLVM-based static binary translator*, in CASES '12 Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems Pages 51-60

[15] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, 2005

[16] Chris Lattner and Vikram Adve, *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*, in Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04), Palo Alto, California, Mar. 2004

[17] Emmett Witchel, Mendel Rosenblum, *Embra: fast and flexible machine simulation,*, in Proceedings of the 1996 ACM SIGMETRICS international conference on Measurement and modeling of computer systems

[18] Nicholas Nethercote, Julian Seward, *Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation*, PLDI'07 June 11-13, 2007, San Diego, California, USA

[19] Mathias Payer, Thomas R. Gross, *Fine-Grained User-Space Security Through Virtualization*, VEE'11, March 9-11, 2011, Newport Beach, California, USA

[20] Vasanth Bala, Evelyn Duesterwald, Sanjeev Banerjia, *Dynamo: a transparent dynamic optimization system*, in Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation

[21] QEMU, `http://qemu.org/`

[22] LLVM, `http://llvm.org/`

[23] Evelyn Duesterwald, Vasanth Bala, *Software profiling for hot path prediction: less is more*, in Proceedings of the ninth international conference

on Architectural support for programming languages and operating systems

[24] David Hiniker, Kim Hazelwood, Michael D. Smith, *Improving Region Selection in Dynamic Optimization Systems*, in Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture

[25] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, Bodo Parady *SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance*

[26] Maged M. Michael, Michael L. Scott, *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*

[27] An ARM board - origen, http://www.origenboard.org