Dynamically Translating Binary Code for Multi-Threaded Programs Using Shared Code Cache

Chia Lun Liu, Jiunn Yeu Chen, Wuu Yang, Wei Chung Hsu

Abstract—mc2llvm is a process-level ARM-to-x86 binary translator developed in our lab in the past several years. Currently, it is able to emulate single-threaded programs. We extend mc2llvm to emulate multi-threaded programs. Our main task is to reconstruct its architecture for multi-threaded programs. Register mapping, code cache management, and address mapping in mc2llvm have all been modified. In addition, to further speed up the emulation, we collect hot paths, aggressively optimize and generate code for them at run time. Additional threads are used to alleviate the overhead. Thus, when the same hot path is walked through again, the corresponding optimized native code will be executed instead. In our experiments, our system is 8.8X faster than QEMU on average when emulating the specified benchmarks with 8 guest threads.

Index Terms— binary translation, mc2llvm, multi-threaded program, hot path, ARM, QEMU.

1. Introduction

Binary translation emulates one instruction-set architecture (ISA) through another ISA. Its applications include legacy program migration, program instrumentation [2], fast simulation [3] and security investigation [4], etc. mc2llvm [1] is a process-level ARM-to-x86 hybrid binary translation system developed in our lab in the past few years, which is able to do static and dynamic binary translation. Currently, it is only able to emulate sequential programs. Our main task is to add support to mc2llvm so that it can emulate multi-threaded programs. In this paper, the emulated ISA is called the guest, and the emulating ISA is called the host.

DBT (dynamic binary translator) takes the guest executable file as data. At run time, guest instructions are translated to host instructions whenever needed. The translated host instructions are stored in the memory place called the code cache. Due to the code cache, subsequent emulations of the same instructions can be done by directly executing the generated native code in the code cache without repeated translation. In this research, we make the following contributions: (1) reconstruct mc2llvm to emulate multi-threaded ARM binaries using the shared code cache; (2) craft an efficient method to do lookup in the address mapping table; and (3) add trace compilation to mc2llvm. The remainder of this paper is structured as follows. Section II sketches the binary translator mc2llvm. Section III illustrates the implementation details. Section IV discusses the experimental results. Finally, this research is summarized in Section V.

2. Background

mc2llvm [1] is an ARM-to-x86 process-level LLVM-based [5] hybrid binary translation system. It is able to do static and dynamic binary translation and both guest and host OS should be Linux. Since our work focuses on dynamic binary translation, we would only describe the dynamic binary translation of mc2llvm in this paper. To emulate a program, mc2llvm first maps the text and data segments of the guest executable into the host memory directly without adding any offset. mc2llvm starts translation from the entry point specified in the guest ELF file. The translation unit of mc2llvm is the basic block. Three steps are involved in the translation. Guest instructions in the guest basic block are translated into the LLVM intermediate representation (IR) and then the LLVM optimizer attempts to optimize the LLVM IR. Because the optimization is performed at run time, only slight optimization is done at this stage in order to reduce run-time overhead. Finally, it uses the LLVM JIT compiler to emit x86 instructions. The emitted instructions are kept in the code cache. Thus, subsequent emulation can be done by directly executing the generated native code in the code cache without repeated translation. Before translating at a guest address, mc2llvm looks up the address mapping table to check if the corresponding translated host instructions
exist. If so, the table-lookup routine returns the host address where mc2llvm can execute code in the code cache. The next guest address is obtained when mc2llvm completes the emulation of a guest basic block by executing its translated code in the code cache. Some ARM instructions cannot be translated to LLVM IR directly such as the svc instruction. This instruction triggers a system call, mc2llvm translates this instruction into a call to a wrapper function that handles the corresponding guest system call. The execution flow is presented in figure 1.

![Fig. 1: The program flow of mc2llvm. g is a guest address. h is a host address.](image)

### 3. Design and Implementation

Our modified architecture of mc2llvm for multi-threaded programs is presented in figure 2. Two kinds of threads are involved in the emulation. One is called the *emulating threads*, which emulates the behavior of each guest thread. Each emulating thread has its private data structure that keeps the guest architecture state for the guest thread. The guest architecture state mainly includes the ARM guest registers, the base address of thread-local storage and the condition flags.

The other is called the *optimizing threads*. It periodically tries to take a trace from a FIFO queue of hot paths. If one is available, the optimizing thread would use the *trace generator* to generate the host instructions for the trace and then stores them in the shared trace code cache. The number of the emulating threads is determined by the programs being emulated and the number of the optimizing threads is set to 3 in our experiments.

#### A Thread Creation and Termination

A new child thread is created by the *clone* system call. Its stack and thread-local storage are prepared by its parent thread. When a guest *clone* system call is encountered, we first call the function pthread_create to produce a new emulating thread. This thread is responsible for emulating the new guest thread. mc2llvm copies the guest architecture state of the parent emulating thread to the child emulating thread and sets ARM guest registers r0 to 0 and r13 to the new stack address and sets the guest base thread-local storage to a value given by the guest *clone* system call. The guest register r0 of the parent thread is set to the child’s thread id. A parameter in the *clone* system call indicates the part of the parent’s data structure that should be copied to the child thread.

![Fig. 2: The architecture of the modified mc2llvm.](image)

A thread is terminated by the *exit* or *exit_group* system call. When an emulating thread encounters the *exit* system call, it does some cleanup then calls pthread_exit. But before it kills itself, the emulating thread has to test if child_tidptr1 has a non-zero value. If so, it does a *futex* wakeup at the child. To emulate *exit_group*, mc2llvm simply terminates the programs.

The emulation of the creation and termination of a thread is done in the wrapper function that emulates the guest system calls.

#### B Atomic Operations

In our experiments, we observe two ways to perform the atomic operations. One is to call the kernel supported __kuser_cmpxchg function, which lies at a specific address. The other is to call the *futex* system call, which is the building block for the mutex.

__kuser_cmpxchg is a function that does compare-and-swap atomically and sets the condition flag C to 1 if the operation is successful or 0 otherwise. We emulate __kuser_cmpxchg by doing the compare-and-swap and guarding the operation with a lock/unlock pair. For the guest *futex* system call, we emulate it by calling the host *futex* system call. Note that both the host and the guest are Linux systems in our experiments.

#### C Thread-Local Storage

In our environment, the base address of the guest thread-local storage is read by the kernel-supported function __kuser_get_tls and is set by the ARM-private system call set_tls. Each emulating thread has a member tls in its guest architecture state to hold this base address. set_tls sets tls to a given value while __kuser_get_tls only returns tls. Since the thread-local storage of every thread, including the main thread, is allocated at run time, mc2llvm does not need to specially pre-allocate the space for each guest thread.

---

1. child_tidptr is the name of a parameter of the clone system call. When the a guest clone system call is invoked, the new emulating thread keeps this value if necessary by testing a flag which is also
D Shared Code Cache

Because the operating system performs context-switch of the architecture states, multiple threads on the same CPU can execute the same code without interfering with one another. Under the emulation, the guest architecture state is maintained by the respective thread. Hence, the OS cannot perform context-switch of the guest architecture states automatically. It becomes the duty of the translator to make sure that emulating threads work with the correct guest architecture states. In mc2llvm, to load an ARM guest register, such as r1, we first load the guest architecture state of the current emulating thread and then load the member in the guest architecture state that corresponds to r1. In this way, when two emulating threads operate on the same guest register, each would load its own private register. Therefore, code cache can be shared because the registers used in an instruction refer to the private registers of the executing thread. The original mc2llvm, which is intended for single-threaded programs, maps each guest register to an individual LLVM global variable. The emulation of single-threaded programs can work correctly under this register mapping because only a single emulating thread manipulates the LLVM global variables. The alternative code cache is the private code cache, that is, each emulating thread keeps its own private code cache. More memory would be needed when the private code cache is used.

E Address Mapping Table

In figure 2, there are two kinds of address mapping tables: global-shared and thread-private address mapping tables. The global-shared table contains all the address-mapping entries and the thread-private table contains only the entries recently used by the owner thread. Both tables are hash tables.

Figure 3 presents how to access the address mapping tables. For a read access, the emulating thread looks up its own thread-private address-mapping table first. If not found there, the global-shared table will be searched next. For a write access, the emulating thread would store the address-mapping entry in both the global-shared and thread-private tables. The thread-private tables would speed up the search significantly because the access to the private table is free of synchronization while the global table has to be locked before the access. In our experiments, table lookup is much faster than synchronization. Although the thread-private tables store only recently used address mapping entries, the hit rate of read access in our experiments is very high (99%). As a result, the global-shared table is used only infrequently. The thread-private table is a simple array with no more than 10,000 entries because we discover that programs in the benchmark needs 1,000-7,000 address mapping entries.

F Synchronization

In our translation system, there are five major critical sections.

- the instruction translator
- the global-shared address-mapping table
- the concurrent FIFO
- LLVM functions used in the system
- the emulation of atomic operations

![Fig. 3: The write and read accesses to the address mapping tables.](image)

The instruction translator is invoked when an emulating thread fails to find the address mapping in the global-shared address-mapping table. Chances are that more than one emulating thread may fail to find the address mapping in the table all the emulating threads then call the instruction translator almost at the same time. Therefore, the instruction translator is guarded by a global lock.

The instruction translator is not used very frequently because instructions are translated only once. Consequently, having a global lock does not hurt the performance much. The global-shared address-mapping table is crafted based on [9]. The global-shared address mapping table is a hash table composed of an array of lists. Atomic CAS (compare-and-swap) instructions are used to lock the table before elements are inserted or deleted. If CAS fails, the whole operation repeats until it succeeds.

The concurrent FIFO is a bridge to pass traces selected by the emulating threads to the optimizing threads. A well-known method in [10] is employed. The technique is similar to [9]. As for LLVM functions, LLVM has an internal mechanism to prevent the race condition. Finally, the atomic operations are emulated with a lock/unlock pair.

G Trace Compilation

The emulating threads perform profiling and select traces. The selected traces are kept in the concurrent FIFO. The optimizing threads compile and optimize the traces in

---

*a parameter of the clone system call.*
the FIFO one by one. At the beginning of a translated block, we patch code to perform trace selection. Trace selection is presented in figure 4. The trace is aggressively optimized with LLVM -O3 option. We insert several instructions in each translated block to test if it is the head of the trace. If so, the emulating thread would jump to the shared trace code cache to execute the instructions of the trace. Otherwise, the emulating thread executes the host instructions in the shared block code cache.

![Trace selection diagram](image)

**Fig 4: Trace selection.**

4. Experimental Results

ARMv5te statically linked binaries are fed into the binary translator. We run the translator on a 48-core x86_64 host machine with Debian Linux operating system. Each core is 2.1 GHz and the memory is 48GB. The benchmark we use is SPEC omp 2001. All the benchmarks are compiled with GNU gcc or gfortran 4.3.2 with flags -O2 -static -fopenmp. Among the 11 benchmarks in the SPEC omp, our translator fails to compile galgel, and fails to emulate ammp. To build mc2llvm, we have to install LLVM-3.2 library beforehand. Four experiments are conducted and described as follows.

A Parallel Emulation

We experiment with various numbers of threads for each benchmark. The resulting emulation time is shown in figure 5. The experiments show that the emulation time decreases when more threads are used, except gafort.

B Comparison of QEMU and mc2llvm

Figure 6 shows the emulation time of QEMU and mc2llvm. Each benchmark makes use of 8 guest threads. QEMU is slower than mc2llvm because, in QEMU, the access to the common address-mapping table is protected by a lock and the access frequency of the table is really high while mc2llvm provides a thread-private address-mapping table. In average, mc2llvm is 8.8x faster than QEMU.

C Comparison with ARM native machine - Origen

Origen is an ARM board. It is equipped with Samsung Exynos 4 Quad Cortex-A9 cores at 1.4 GHz and 1GB memory. Figure 7 shows the emulation time of Origen and mc2llvm. Each benchmark makes use of 4 guest threads. On average, Origen is 5.45x faster than mc2llvm.

D Trace compilation

Figure 8 shows the emulation time of mc2llvm with and without trace compilation. Each benchmark makes use of 8 guest threads in this experiment. We take the SPEC omp 2001 benchmark with the ref data set as input because trace compilation normally takes longer time to show its advantage. On average, we obtain 33% performance speedup with trace compilation.
5. Conclusion

We extend the binary translator mc2llvm [1] from sequential-program emulation into multi-threaded-program emulation. The code cache is designed to be shared among the threads in order to reduce the memory usage. The synchronization overhead is mostly reduced by having thread-private address-mapping tables in addition to the global-shared address-mapping table. Runtime optimization is done by trace compilation with 3 extra threads to generate code for the traces. According to our experiments, mc2llvm is 8.8x faster than QEMU when emulating programs with 8 guest threads. mc2llvm is 5.45x slower than Origen [11] when emulating programs with 4 guest threads. When mc2llvm performs trace compilation, it gains 33% performance speedup on average.

References

(Periodical style)


Chia Lun Liu was born in Tainan, Taiwan, in 1989. He received the B.S. degree from National Sun Yat-sen University, Kaohsiung, in 2011 and the M.S. degree from National Chiao Tung University (NCTU), Hsinchu, in 2013, both in computer science. He is currently working as a software engineer at Gemtek. His research interests include operating systems, compilers.
Jiunn Yeu Chen was born in Taiwan, in 1984. He received both the B.S. degree and M.S. degree from the National Chiao Tung University, Hsinchu, in 2006 and 2008. He is currently pursuing the Ph.D. degree with the Department of Computer Science in the National Chiao Tung University. His research interests include binary translation, system software and parallel programming.

Wuu Yang was born in Taiwan, Republic of China, in 1960. He received the B.S. degree from the National Taiwan University, Taipei, Taiwan, in 1982 and the Ph.D. degree from the University of Wisconsin at Madison, in 1989, both in computer science. He is currently a professor with the Department of Computer Science, National Chiao-Tung University. His research interests include embedded systems, systems software, programming language, and garbage collection.

Wei Chung Hsu received his PhD degree in computer science from the University of Wisconsin, Madison, in 1987. Currently, he is a professor in the department of computer science and information engineering, at the National Taiwan University, Taiwan. From 1999 to 2009, he was a professor at the computer science and engineering department, at University of Minnesota, twin cities, USA. From 1993 to 1999, he was a runtime optimization architect in the California Language Lab, at Hewlett Packard Company. Prior to joining HP, he was a senior architect at Cray Research, in Chippewa Falls, Wisconsin. His current research interests include virtualization, dynamic binary translation and optimization techniques.