Control-Flow Based Program Partition in Static Binary Translation

Jiunn-Yeu Chen*, Wuu Yang and Wei-Chung Hsu

aNational Chiao Tung University, Taiwan
bNational Taiwan University, Taiwan

*Corresponding Author: jiunnyeu@cs.nctu.edu.tw

ABSTRACT
Static binary translation has several advantages over dynamic binary translation. It can afford more time consuming global and whole program optimizations to generate faster and/or smaller code. However, since static binary translations often treat the complete executable as a single large function, this could suffer from excessive optimization time. Overly prolonged translation time could seriously burden the testing and validation cycles of the translated binary. Intuitively, dividing the whole executable into several smaller chunks might help since each chunk could be treated as a smaller function, thereby decreasing the global optimization time. However, improperly breaking up a segment of executable could render ineffective optimization, for example, local branches might be translated as expensive function calls, since the branch target is inadvertently separated into a different chunk. This could drastically increase the execution time of generated codes, especially when such local branches appeared in loops. In this paper, we proposed a control flow based executable partitioning method to effectively break a large executable into multiple smaller functions or segments. Such reduced translation units significantly improved translation time by 2.68x, on average, over the EEMBC suite. Our partitioning also successfully avoided breaking a segment at inadequate places so that the execution time suffered only 12% of degradation.

Keyword: Binary Translation; Control Flow Graph; Program Partition

1. Introduction
Binary translation is a widely used technique for several usage, including program migration, program simulation, runtime optimization, binary instrumentation. There are two main category of binary translator: dynamic and static. Dynamic binary translator translates the source program on the fly. The dynamic binary translators include Aries [11] which translate PA-RISC to IA-64 and IA32 execution layer [1] which translate IA-32 programs to IA-64 platform. QEMU [2] is a dynamic binary translator can support multiple different frontend ISAs and backend ISAs. HQEMU [7] is a variation of QEMU which can translate source binary to LLVM IR.
On the contrary, the static binary translator translates offline like the legacy compilers. The static binary translators include FX!32 [5] which instrument executed instructions
during execution time and translate it after execution. UQBT [6] translates source binary into self-defined high level intermediate representation. Chen et al. [4] implement an ARM based binary translator which translates ARM program to a MIPS like platform. LLBT [9, 3, 10] which translates ARM program to the LLVM IRs which can then support multiple different ISAs.

For the dynamic binary translators, there is an inevitable trade-off between the compilation time and the execution time. Since the compilation is working on-the-fly, any enabled optimization increases the execution time. The applied optimizations are limited to avoid long compilation time, which may delay the total execution time or the response time. Nevertheless, the translated instructions may located in a hot path. The code quality of the hot path is critical since it may executed repeatedly. The time saved by disabled optimizations may be consumed by the code with bad quality.

In contrast, the static binary translator can ignore the constraint of compilation time because the compilation is done offline. The static binary translator can do its best to optimize the translated instructions. Nevertheless, the concern of the translation time becomes more serious these days. It is more common to translate programs to intermediate representation instead of instructions of specific ISA. The intermediate representation may be self-defined, just like what UQBT or QEMU does, or already defined and used by some compiler infrastructures, just like LLVM for LLBT. These static binary translator can leverage the optimizations provided by the existing compilation framework. However, the rich set of optimizations consumes more compilation time.

The other problem is that the static binary translation raises the optimizations into whole program optimizations. In general functions or compilation units are not large enough so the optimizations can still accomplish in a reasonable amount of time even if it’s time complexity is quadratic. However, the static binary translator often translates the whole program into one function. The optimization time may drastically enlarged when the optimization is applied to the whole program.

In this paper, we demonstrate how to reduce the translation time by partitioning the translated LLVM IRs from one function which includes all the translated LLVM IR into several functions. The partition is based on the control flow analysis. The entry points of functions can be collected by either information from the ELF file or the destination addresses of the call instructions. With the collected entries the LLVM function which contains all translated IRs can be partitioned into several LLVM functions. Partitioning the function according to the control flow analysis can prevent unnecessary control transfer between functions. If the function is not partitioned according to the control flow, a basic block may be break into two functions. The fall through control flow becomes a function call to another function.
If the basic block locates in a loop, it may hurt the performance. Besides collection of the function entries, an additional combination is offered to prevent the fragmentation of the translated LLVM IRs. The collected function entries may include both function entries and basic block entries in the source program. Making a basic block entry as a function entry may hurt performance because local branch may translated into function call, especially when it is located in a loop. The combination help to avoid the performance issue brought by the fragmentation problem.

The rest of this paper is structured as follows. Section 2 introduces LLBT which is the static binary translator we implement the partition mechanism. Section 3 describes how the partition mechanism implemented, including the function entry collection and the combination of function entries. Section 4 mentions the necessary modifications to the translated LLVM IRs of LLBT, which makes the partitioned LLVM IRs can work correctly. Section 5 shows the experiment result including the translation time and the execution time of the translated program after the partition mechanism is used. Section 6 is the conclusion.

2. LLBT

This section introduces LLBT. LLBT is a static binary translator which can translate ARM binaries to LLVM IRs. It leverages the LLVM framework which supports multiple backends to translate the source binaries to binaries of different ISAs.

2.1 LLVM

LLVM [8] is a widely used compiler framework which was developed by Lattner er al. It provides a rich set of target independent optimizations which can operate on LLVM IRs. LLVM framework also supports lots of backends, including X86, ARM, MIPS, PowerPC...etc. LLBT takes this advantage to translate ARM binaries to different target ISAs more easily. Unlike the traditional binary translators which directly translate source binary to the target, LLBT saves the repeated work on writing code template for each target.

2.2 Layout

Each ARM instruction is translated to corresponded LLVM IRs by LLBT. Each translated instruction is composed by one or multiple basic blocks due to the support of conditional flags and shifter operands in ARM ISA. An ARM instruction can be translated into 3 basic blocks. The first basic block checks if the result of the condition code. If it is true, then the following basic blocks will be executed. Otherwise, it branches to the first basic block of the next ARM instruction. The second basic block is the instruction body which executes the bitwise and operation.
The last basic block updates the emulated conditional flags of the ARM processor. In the meantime, LLBT can translate both ARM and Thumb instruction set. All the translated LLVM IRs are put in one LLVM function. In other words, the whole program is wrapped into the function. When the translated program is started, the function will be invoked and the control flow will transfer to the LLVM IRs which belongs to the ARM instruction located at the entry point of the ARM program. An address mapping table is used to handle the indirect branches.

3. Partition

This section describes how to partition the whole program into several pieces. It includes two main issues. One is how to find the possible function entries. The other is how to prevent the fragmentation of the program because some found entries are actually block entries, and it is hard to distinguish them from the real function entries.

3.1 Function Entry Collection

We collect the possible function entries from two different ways. The first is according to the branch instructions of ARM/Thumb ISA. The destination addresses of the branches could be function entries. The other is according to the ELF file. An intuitive way to find the function entry is finding the function call instructions. In ARM/Thumb ISA, the instruction `bl` and `blx` is often used for function call. The destination addresses of these instructions are considered as function entries. Besides `bl` and `blx`, the instruction `b` may also be used to do function call. For example, a tail call may be generated as `b` instead of `bl`.

The instruction immediately follows unconditional branch `b` is also considered as a function entry because of tail call. If the unconditional branch `b` is a tail call, then it is the last instruction of the function, and the next instruction is also a function entry. ELF file also helps to find possible function entries. The starting addresses of executable sections such as `.text` can be fetched from the ELF file. They are function entries with no doubt. The sections `.init_array` and `.fini_array` contains function pointers which are invoked during begin and end of the program. The contents of the function pointers are also function entries. Besides the ELF information, we also check the word contents in the loadable sections to find probable function pointers. Functions only accessed by function pointers cannot be found through the direct branches or ELF information. Scanning all the word contents in the loadable sections can help find them.

3.2 Combination of Function Entries

In section 3.1 we introduce several method to partition the program into several
LLVM functions. Nevertheless, it is hard to precisely decide whether an address is a real function entry, so a basic block entry may be considered as a function entry. The fragmentation of the translated LLVM IRs does not affect the correctness of the execution, however, it may hurt the performance. One of the reason is because the number of function calls is increased, and the overhead brought by prologue and epilogue are also increased. The other reason is lots of optimizations are focus on function level, so the fragmentation problem limits the optimizations on smaller range. If basic blocks of a loop are generated into separate LLVM functions, and the loop is on the hot path, the overhead will cut down the overall performance.

To solve the fragmentation problem, an additional combination step is used after the function entry collection. It aims to avoid partitioning one function into smaller pieces and ensures that there is no basic block entry considered as function entry. The combination makes one function include several possible function entries. We call the first entry as function entry and the others as call entry. The following describes how to identify call entries.

The first case of call entry is the Thumb version _bl_ because it is sometimes used as long branch in large functions. An additional check is used to decide if the destination of _bl_ is a function entry. If the destination instruction is a push instruction, then it is a function entry. Otherwise, it is a call entry.

The second case is the destination address of instruction _b_. Although this instruction could be used for tail call, it is used for local branch in most case. The same reason as above, the address next to the instruction _b_ is also considered as a call entry instead of function entry.

The fourth case is the function entries collected by the word content in the load section. Although this method can help find function pointers, it may also collect addresses which are neither function entries, nor basic block entries.

The last case is switch statement. It can help decide if a function entry is a call entry. The switch statement does not jump cross functions. If a function entry is located in between a switch and its target, it is also considered as a call entry.

4. Modification to the Code Generation

In this section we are going to introduce the modifications to the generated LLVM IR. The basic modification is generating separate LLVM functions for ARM/Thumb instructions start from each function entries with the prologue and epilogue. The following sections mention additional considerations should to be noticed. Without these modifications, the partitioned program cannot work correctly.

4.1 Changes on Branch Instructions
Due to the whole program is generated as multiple functions instead of one function, all the branch instructions should be taken care to make the control flow work correctly. For a direct branch, if the destination is not in the same function, it will be translated to an LLVM call instruction. Otherwise, it will be translated to an LLVM branch instruction. Even if the direct branch is `bl`, it still be translated into a branch instruction instead of a call instruction to reduce the function calls.

For an indirect branch, if the destination is in the same function, the address mapping table will handle the control transfer. If the destination is outside of the function, it will be handled in different way according to the instruction. If the instruction is a return, it will invoke the LLVM return instruction. Otherwise, a function pointer dispatcher is invoked to call the function pointer.

**4.2 Multiple Function Entries in One Function**

As mentioned in section 3.2, an LLVM function could contain several entries due to combination. Some of the entries could still be function entries and may be called from the outside of current function. To make the control flow work correctly, an LLVM switch instruction is used to redirect the control flow to the corresponding LLVM IR according to the destination address in the run time. The LLVM switch instruction records the ARM address of each entry and pairs the address with the corresponding LLVM IR.

**4.3 Function Pointer Dispatcher**

The function pointer dispatcher is used to handle the indirect branches cross different functions. It is a function which is only used to call the corresponding LLVM function by the destination address in the run time. It is implemented by LLVM switch instruction. Each entry is composed by an ARM address and a corresponding LLVM label. The only instruction belongs to the label is a LLVM call instruction which calls to the function contains the ARM address. All the function entries and call entries are included in the switch table to ensure the correctness.

**5. Experiment**

In this section, we present several experiment results to show the effectiveness of the partition mechanism. We compare both the translation time and the execution time.

**5.1 Experiment Environment**

The benchmarks used in the experiments is the EEMBC benchmark suite. All the benchmarks are compiled with ARM GCC 4.9.1 with flag “-O2 -mthumb -march=armv5te” and linked with µClibc version 0.9.33.2. LLVM version 3.5 is used
to translate the LLVM IRs into x86 machine code.
All the benchmarks are tested on a machine with an Intel Xeon 5506 processor and
12GB RAM, running Linux version 3.16.

5.2 Translation Time Comparison

![Total Translation Time Ratio (without partition/with partition)](image)

Figure 1: The performance of whole the binary translation after partitioning

This section present the total translation time from the translation of LLBT to the
accomplishment of the target binary. Figure 1 shows the speedup of whole the binary
translation when the partitioning method is used. The result shows that the
performance of whole the binary translation can be improved to 2.68x on average. The
speedup is at least 2.39x for iirflt01 and can exceed to 4.26x for cjpeg. It means that
the overhead brought by the partitioning method is much lower than the improvement
gained in the optimizer and the code generator.

5.3 Execution Time Comparison

![Exection Time Ratio (without partition/with partition)](image)

Figure 2: The performance of the translated program after partitioning

This section shows the execution time ratio of the translated program when the
partition is applied. Comparing the execution time can tell us if the partitioning
method can reduce the translation time and keep the performance at the same time.
Figure 2 shows the performance after partitioning. There is a great variety between
the benchmarks. Some programs are slow downed as predicated. For programs like
rspeed01 and bezierfloat, the degradation of the performance can be 44% and 47%.
For some other programs, the partition version performs even better. For example, the
partitioned canrd01 and pntrch01 has 1.35x and 1.37x speedup. By the result, it
means that the partitioning method does not necessarily increase the execution time. The degradation of the performance is 12% on average, which is affordable especially when the translation time has 2.68x speedup.

6. Conclusion
Static binary translation is able to perform optimizations much more effective than dynamic binary translation. First, since static translation time is not part of runtime, it can tolerate some slower but more powerful global optimizations, such as graph coloring register allocation (which runs in quadratic time) instead of linear scan approach (which runs in linear time). Second, static binary translation can view the entire executable as one giant function, it could conduct optimizations at a scope much greater than the scope of a dynamic binary translator. For example, the static translator LLBT can map the guest architecture states effectively to the registers of the host machine since it can assign a guest register to one or more physical registers in the host over the entire executable. However, when the enlarged scope coupled with aggressive global optimizations, the result is a possible excessive long translation.

This paper presents a mechanism to partition the translated LLVM IRs of LLBT into smaller translation units. This would significantly reduce the optimization time since several very powerful analyses and optimizations in LLVM run in quadratic time. The proposed mechanism collects both function entries and basic block entries to guide the partitioning process. Naive partitioning could sometimes introduce undesirable limitations to optimizations and degrade the generated code. For example, if a branch and its target get separated into two translation units, then a local branch may be translated as an external call. Our mechanism avoids such unwise partitioning. We have also considered combining functions and blocks into a larger unit to enable more effective optimizations.

The EEMBC benchmark suite is used in our experiment to evaluate the effectiveness of our executable partitioning approach. Overall, the translation becomes 2.68 times faster, while the runtime suffers 12% of degradation. Follow up tuning has been conducted and we expect to see greater improvement to both compile time and code execution performance in the near future.

REFERENCES


