# Automated Generation of Dynamic Binary Translators for Instruction Set Simulation

Katsumi Okuda, Minoru Yoshida, Haruhiko Takeyama, Minoru Nakamura
Advanced Technology R&D Center, Mitsubishi Electric Corporation, Japan
Email: {Okuda.Katsumi@eb,Yoshida.Minoru@ea,Takeyama.Haruhiko@cw,Nakamura.Minoru@ea}.MitsubishiElectric.co.jp

**Abstract—Instruction set simulators (ISSs) are indispensable tools for developing new architectures and embedded software. Due to the increasing variety of architectures and time-to-market pressure, it is important to efficiently develop fast ISSs based on dynamic binary translation. However, the implementation of such ISSs needs more effort than interpretive ISSs. In this paper, we propose a novel framework that generates ISSs based on dynamic binary translation from descriptions of interpretive ISSs. Our results on SH, MIPS64, and ARM show that the generated ISSs are 1.4 to 13.4 times faster than their original interpretive ISSs.**

## I. Introduction

Instruction set simulators (ISSs), which are tools that functionally mimic the behavior of a target architecture on a host machine, are used to develop new architectures and embedded software. Due to the increasing variety of architectures and time-to-market pressure, fast ISSs must be efficiently developed.

Two kinds of ISSs are widely used: interpreters and dynamic binary translators (DBTs). An interpreter sequentially fetches, decodes, and executes target instructions to simulate the target program. On the other hand, a DBT converts target instructions into host instructions at runtime and executes them. When it executes the target instructions that have already been converted, it can skip the translation and just execute the host instructions.

In general, even though the implementation of an interpreter is easy, the simulation speed is slow. In contrast, a DBT's simulation speed is fast, but its difficult implementation reflects its complexity. A DBT has an instruction specific translation function that converts the target instruction into host instructions or intermediate representation. Hence, DBT developers need to deeply understand not only the target instructions but also the host instructions or the intermediate representation.

In this paper, we propose a novel framework that generates a DBT from the description of an interpreter. Our framework compiles instruction specific behavior functions for the interpreter into an object file using a host compiler and then generates translation functions for the DBT by extracting the translation template from the object file. As a result, DBT developers do not need to manually code translation functions.

The following are the contributions of this paper:

- We developed a novel framework that generates a DBT from the C/C++ behavior functions of an interpreter.
- We extended a decoding entry [1]–[4] for decoder generation, so that the decoding entries can also be used to generate a DBT.
- We experimentally confirmed that the DBTs for SH, MIPS64, and ARM generated from a reasonable amount of description by our framework are 1.4 to 13.4 times faster than their original interpreters.

The remainder of this paper is organized as follows. Section II discusses related work in the ISS field. Section III shows the DBT generated by our framework. Section IV describes the implementation difficulty of the binary translator and introduces the DBT's generation framework. Our experimental results are shown in Section V. Finally, our conclusion is in Section VI.

## II. Related Work

ISSs are classified into an interpreter and a binary translator based on how they execute target programs. SimpleScalar [5] and GDB include interpreters. Since an interpreter's implementation is easy, these interpreters have been retargeted to such multiple instruction sets as ARM and PowerPC. JIT-CCS [6], which is a kind of interpreter, caches the instruction decoding results. Since it can skip the instruction decoding that was previously executed, it simulates faster than conventional interpreters. However, since it does not translate target instructions into host instructions, the simulation cannot be faster than DBTs.

Binary translators are further classified into static binary translators (SBTs) and DBTs. SBTs, which translate the whole target program before its execution, have been previously described [7]–[10]. Even though SBTs can simulate quickly, they cannot support self-modifying code.

DBTs, which perform the same kind of translation as SBTs but at runtime instead of beforehand, can support self-modifying code by re-translating the modified code at runtime. Many DBTs have been developed, including Shade [11], Embra [12], and QEMU [13]. While Shade and Embra do not consider the portability of the host architectures, QEMU adopts an intermediate representation that improves the portability for multiple host architectures. However, since the translation procedure is described as handwritten C code, retargeting QEMU to a new architecture is more complex than implementing a new interpreter for it. In contrast, our framework generates DBTs from the descriptions of interpreters. Consequently, the cost of retargeting DBTs by our framework is equivalent to the cost of implementing interpreters.

Such architecture description languages (ADLs) as nML [14], ISDL [15], LISA [16], EXPRESSION [17], ASIP Meister [18], and Harmless [19] can generate ISSs. Most can generate not only ISSs but also processors. However, developers need to learn an ADL to write the processor description. In contrast, our framework generates an ISS from the description written in general purpose language C/C++. Since C/C++ is one of the most popular general purpose languages, most developers do not need to learn a new language to implement a DBT.

All ISSs need to decode instructions to execute target programs regardless whether they are interpreters or binary translators. Instruction decoder generation algorithms have previously been proposed [1]–[4]. Our framework can be used in combination with these decoder generation algorithms.

978-1-5090-1558-0/17/$31.00 ©2017 IEEE

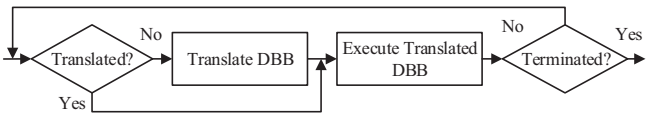


Fig. 1. Binary translation flow

## III. Generated DBTs

This section introduces the details of a generated DBT.

### A. Dynamic Binary Translation

The DBT executes target instructions by translating them into host instructions, which it executes. It also caches the host instructions to use them when executing the same target instructions more than once.

The values of the instruction fields are extracted at the translation time and embedded in the generated host instructions as immediate operands. Therefore, when the DBT executes target instructions already translated, it can skip decoding the instructions and extracting the instruction fields. As a result, it can simulate faster than the interpreters that need instruction decoding and field extraction to execute each target instruction.

The unit of binary translation is a dynamic basic block (DBB), which is an area that begins at the instruction that was executed immediately after a branch and ends with the next branch. The difference between a DBB and a conventional basic block has been discussed [20]. When the target architecture has a delay slot mechanism, a DBB contains instruction(s) in delay slot(s) after the branch instruction. The DBT translates each DBB into an execution function that contains the host instructions generated from the target instructions in the DBB.

The DBT's main control flow is its emulation loop (Fig. 1). At the head of the emulation loop, the DBT examines the translation cache to determine whether the DBB indicated by the program counter (PC) has been already translated. If it has not been translated yet, the DBT translates it into an execution function and stores the execution function in the translation cache; otherwise, the DBT skips the translation. Subsequently, the DBT calls the execution function. Finally, if the termination condition is not satisfied, the DBT processes the next DBB indicated by the PC.

### B. Translation Template

When translating a DBB to the execution function, the DBT converts the target instructions one by one. It uses specific instruction translation templates to convert each target instruction into host instructions. A translation template consists of parameterized host instructions. With translation templates, the DBT can generate host instructions by expanding a translation template in the translation cache and substituting values available at the translation time for template parameters.

A translation template contains the following parameters:

- *instruction field:* specifies a target instruction operand, such as register index, immediate value, or offset.
- *program counter:* indicates the target address at which the target instruction is executed.
- *symbol address:* holds the host address of a function or variable that must be determined at runtime, e.g., an array to which the target registers are mapped and a common function that is used to emulate the target instruction.

As an example of an instruction translation, Fig. 2 shows the translation of the DADD instruction in MIPS64. The translation template contains all of the host native x64 instructions needed to mimic the DADD behavior. The DBT extracts the fields of the DADD instruction and substitutes their values for the template parameters.

### C. Prologue and Epilogue

In our framework, some host registers are reserved so that the translation template can use them freely without save/restore operations. Hence, the prologue of an execution function saves the reserved registers and the epilogue restores them.

DBB basis translation does not need to update the PC for each execution of the target instruction since the PC value can be fixed at the translation time. It is sufficient that the execution function updates the PC only at the end of it. The epilogue updates the PC depending on the type of branch at the end of the DBB. After calling the execution function, the DBT refers to the updated PC to determine the next DBB to be executed.

## IV. DBT Generation

### A. Basic Idea

DBT implementation using translation templates needs a large effort from developers to create a translation template for each instruction, since they need to directly manipulate the host instructions. To address this problem, our framework uses behavior functions. A behavior function is a C/C++ function that implements the instruction's behavior instead of its translation. Our framework compiles behavior functions into an object file using a host compiler and extracts translation templates from the resulting object file. Since behavior functions are just modules of an interpreter, the effort required by developers to implement a DBT by our framework is equivalent to that of the interpreters.

Fig. 3 shows the DBT generation flow. Our framework takes extended decoding entries and behavior functions as input and generates modules for the DBT.

An extended decoding entry (①) exists for each target instruction. The extended decoding entries are based on decoding entries that were previously introduced [4]. Our decoding entry is extended to generate both the instruction decoder and the translation functions. Each extended decoding entry is composed of the following items:

1) *instruction name:* unique instruction name
2) *pattern:* instruction's bit pattern
3) *exclusion conditions:* optional conditions that exclude an invalid instruction word that matches the instruction pattern
4) *instruction field information:* list of triplet (name, length, and position) for each field of instruction
5) *instruction type:* whether the instruction is a branch
6) *branch type:* whether the branch instruction is conditional
7) *delay slot count:* the number of delay slots

The first three items are needed to generate an instruction decoder. We added the last four items to generate DBTs. The last two items are required only if the instruction is a branch.

Behavior functions (②) are written in C/C++. Our framework defines the description guideline for behavior functions to use a behavior function as the source of a translation template.

The adaptor generator (③) creates adaptor functions (④) from the decoding entries. Each adaptor function exists for a

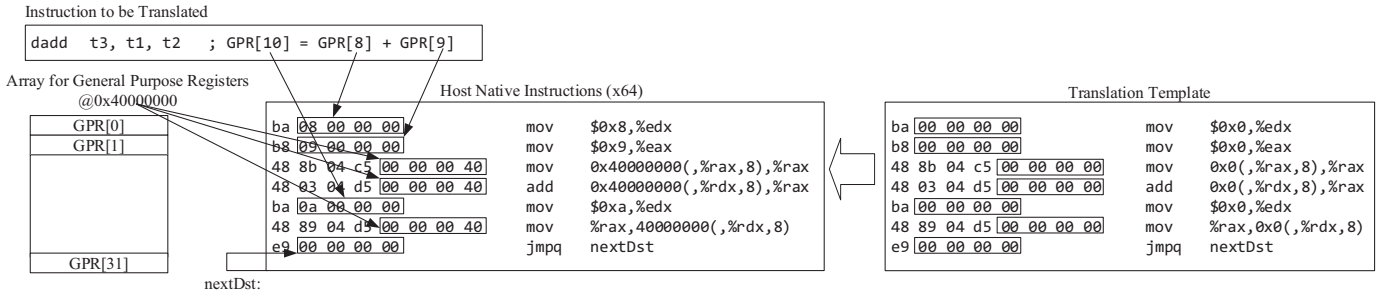


Fig. 2. Instruction translation by a translation template

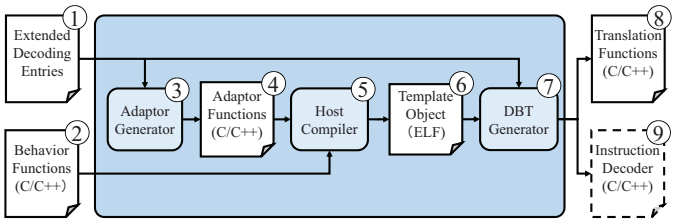


Fig. 3. DBT generation flow

```
1  void DADD(uint32_t pc, uint32_t rs, uint32_t rt,
       uint32_t rd)
2  {
3    GPR[rd] = GPR[rs] + GPR[rt];
4  }
5
6  void BEQ(uint32_t pc, uint32_t rs, uint32_t rt,
       uint32_t offset)
7  {
8    BRANCH_RESULT = GPR[rs] == GPR[rt];
9    if (BRANCH_RESULT) {
10     NEXT_PC = pc + (sext16(offset) << 2);
11   }
12 }
```

Fig. 4. Example of behavior functions

corresponding behavior function and contains the inline call of the behavior function.

The host compiler (⑤) builds the adaptor and behavior functions. The compilation result is the template object file (⑥) in a relocatable object file format such as ELF. The template object file contains translation templates.

The DBT generator (⑦) takes the decoding entries and the template object file as input and generates a DBT whose main part consists of instruction specific translation functions (⑧) and an instruction decoder (⑨). The DBT generator creates translation functions using the translation template stored in the template object and also creates an instruction decoder based on a previous algorithm [4].

### B. Behavior Function

Each behavior function written in C/C++ emulates a target instruction. A behavior function takes the instruction fields as input and performs such instruction specific operations as calculation between registers, memory access, PC manipulation, etc.

Our framework establishes a description guideline of behavior functions, so that their compilation results can be used as a translation template. Below, this section shows the details of the description guidelines.

*1) Function Signature:* The behavior function's signature must be written based on the following rules so that a generated adaptor function can correctly call the behavior function.

- *function name:* The function name must equal the instruction name described in the corresponding decoding entry.
- *function parameters:* The first parameter must be the PC value. The second and any subsequent parameters must be instruction fields whose order must correspond to the order of the instruction fields described in the decoding entry.

*2) Resource Access:* Target resources such as registers must be mapped to global variables. The host compiler translates the global variables used in behavior functions into relocatable symbols. The DBT generator can treat the relocatable symbols as template parameters, since the position of each parameter can be obtained from the relocation table in the template object file.

*3) Reserved Variable:* The guideline reserves the following variables for the implementation of a branch instruction:

- `NEXT_PC`: branch destination
- `BRANCH_RESULT`: whether the branch condition is satisfied

These variables are needed to update the PC values in the epilogue of the execution function. More specifically, the epilogue updates the PC with `NEXT_PC`, when the DBB's last instruction is an unconditional branch. When the last instruction is a conditional branch and `BRANCH_RESULT` is *true*, the DBT updates the PC with `NEXT_PC`; otherwise it updates the PC with the address of the instruction immediately following the DBB. The DBT generates an appropriate epilogue for each branch type that it can derive from the extended decoding entry as a result of decoding the instruction.

Note that `NEXT_PC` is a write-only variable. In a DBB basis binary translation, since updating the PC per instruction execution is not needed, `NEXT_PC` does not hold a valid PC value except the branch at the end of the DBB. The behavior function must get the PC value from the first parameter instead of `NEXT_PC`. The DBT always substitute the PC value for the first parameter at the translation time.

### C. Examples of Behavior Functions

As examples of behavior functions, Fig. 4 shows the definitions of the DADD and BEQ instructions in MIPS64. The DADD instruction adds two 64-bit source registers and stores the sum in a 64-bit destination register. Parameters `rs`, `rt`, and `rd` correspond to each instruction field. `GPR` is an array variable to which the general purpose registers are mapped.

Since the DADD instruction does not access the PC, the first parameter is not used.

The definition of the BEQ instruction stores the necessity of the branch in reserved variable `BRANCH_RESULT`. If the branch is needed, it updates variable `NEXT_PC`.

### D. Adaptor Function

The template generator creates an adaptor function for each behavior function. Adaptor functions allow the DBT generator to use the compilation results of the behavior functions as translation templates.

In the template object file compiled from behavior functions, access to the instruction fields and the PC must be parameterized so the DBT can use host instructions in the template object file as a translation template. The parameter positions in the template object file must also be available to the DBT generator, which uses the positions of the template parameters to generate parameter substitution code.

Adaptor functions address this problem. An adaptor function calls its behavior function which is expanded inline. In the generated adaptor function, the PC and instruction fields are declared to be external variables. The address references of these variables are given to the instruction behavior function as its arguments. As a result, the references of the PC and the instruction fields are converted to relocatable symbols in the template object file. The positions of the relocatable symbols are stored in the relocation table with which the DBT generator can manipulate the template parameters.

The adaptor function also inserts a marker and jump code around the inline call of the behavior function. Markers and jump codes help the DBT generator extract translation templates from the template object file.

Fig. 5 shows an example of an adaptor function. Instruction fields are declared to be external variables `pc, i_rs, i_rt, i_rd`. These external variables are converted to relocatable symbols in the template object file.

Macro `MARK` is an alias of inline assembler code that only places a marker, which is a fixed magic number that indicates that the instruction next to it is the beginning of the translation template. Hence, the DBT generator can remove the prologue of the adaptor function by searching for the magic number.

Compared to the adaptor function's prologue, it is difficult for the DBT generator to remove the adaptor function's epilogue. If the adaptor function is compiled into the code that has multiple exits, then its epilogue may be duplicated more than once.

Macro `JUMP_TO_NEXT_BLOCK` solves the difficulty of removing the epilogue by eliminates the need to do so. Macro `JUMP_TO_NEXT_BLOCK` is another inline assembler code that holds the jump instruction. The jump's destination is given as a reserved symbol. A symbol value must be assigned to the address of the next instruction's emulation code at the binary translation time in a way that resembles other template parameters. Hence, the DBT generator can use the host instructions (including the epilogue) as a translation template.

### E. Generation Method

In this section, we show the structure of the translation function and introduce the generation algorithm's details.

*1) Translation Function:* A DBT calls an instruction specific translation function after decoding the instruction. The translation function takes the PC and the extracted instruction fields as input and generates host instructions that emulate the target instruction. The generation of host instructions is done by expanding a translation template code and substituting all of the template parameters.

```
1 void AdaptorDADD(void)
2 {
3     extern int pc, i_rs, i_rt, i_rd;
4
5     MARK;
6     DADD((uint32_t)&pc, (uint32_t)&i_rs, (uint32_t)&
          i_rt, (uint32_t)&i_rd);
7     JUMP_TO_NEXT_BLOCK;
8 }
```

Fig. 5. Adaptor function

```
1  /* Start: Function Header */
2  void TranslateDADD(uint32_t pc, uint32_t rs, uint32_t
       rt, uint32_t rd)
3  {
4  /* End: Function Header */
5
6    /* Start: Template Expansion */
7    static const uint8_t template[] = {
8        0xBA, 0x00, 0x00, 0x00, 0x00, ...
9        ...
10       ..., 0x00, 0x00, 0x00, 0xC3 };
11   memcpy(dst, template, sizeof(template));
12   uint64_t nextDst = (uint64_t)dst + sizeof(template);
13   /* End: Template Expansion */
14
15   /* Start: Parameter Substitutions
16   memcpy(&dst[1], &rs, sizeof(rs));
17   memcpy(&dst[6], &rt, sizeof(rt));
18   uint32_t tmp1 = (uint32_t)&GPR;
19   memcpy(&dst[14], &tmp1, sizeof(tmp1));
20   ...
21   uint32_t tmp2 = uint32_t(nextDst + -4 - (uint64_t)&
         dst[51]);
22   memcpy(&dst[51], &tmp2, sizeof(tmp2));
23   /* End: Parameter Substitutions */
24
25 /* Start: Function Footer */
26   dst = nextDst;
27 }
28 /* End: Function Footer */
```

Fig. 6. Translation function

A translation function is composed of the following generated parts:

- function header
- template expansion
- parameter substitutions
- function footer

As an example, Fig. 6 shows the translation function for MIPS64 DADD instructions. The comments inside Fig. 6 indicate the corresponding generation parts.

In the function header part, Parameter `pc` corresponds to the PC, and Parameters `rs`, `rt`, and `rd` correspond to each instruction field.

In the template expansion part, the translation template is a constant byte array (lines 7-10). The subsequent code expands the translation template into the translation cache indicated by the variable `dst` (lines 11-12).

The parameter substitution parts in lines 16–23 substitute the values available at the translation time for the template parameters. In this case, Parameters `rs`, `rt`, and `rd` and the address of array `GPR` to which the target registers are mapped are substituted for corresponding template parameters. In the last line of the parameter substitution parts, the start address of the code that emulates the subsequent instruction is assigned to the reserved symbol.

The function footer part updates variable `dst` with the address where the next template will be expanded.

*2) Algorithm:* To generate the translation function, the DBT generator takes the decoding entries and the template object file derived from the behavior functions as input and applies Algorithm 1 to each decoding entry.

In line 1, Procedure $CreateFunctionHeader$ outputs the function header part. It uses instruction name $entry.name$ and field information $entry.fields$ to generate the signature of the translation function.

In line 2, Procedure $FindSymbol$ finds the symbol table entry for the instruction's adaptor function by its name. The adaptor function's name is created by concatenating "Adaptor" with instruction name $entry.name$. The symbol table entry gives the position and the size of the adaptor function in the template object file.

In line 3, Procedure $CreateTemplateExpansion$ outputs the template expansion part using the translation template contained by the adaptor function. To extract the translation template from the adaptor function, the generator skips the function prologue by searching for the magic number from which it extracts the area to the end of the adaptor function.

In lines 4-6, the DBT generator outputs the parameter substitution parts with the relocation table, where all of the parameter's positions are stored. The generator iterates over each relocation table entry and checks whether the entry corresponds to one of the parameters in the translation template. Function $IsParameter$ checks it by examining whether the relocation position is in the translation template area. Procedure $CreateParameterSubstitution$ creates each parameter substitution. It uses the relocation position to get a parameter's position. The values to be substituted for parameters are accessible symbols in the translation function. The names of these symbols is also obtained from the relocation entry.

In line 7, Procedure $CreateFunctionFooter$ outputs the function footer part. It has no parameters, since the footer part always has a fixed form.

**Algorithm 1** Generation algorithm

**Input:** template object file *objFile*, extended decoding entry *entry*
**Output:** translation function

```
1: CreateFunctionHeader(entry.name, entry.fields)
2: symbol ← FindSymbol(objFile.symbolTable, entry.name)
3: CreateTemplateExpansion(symbol)
4: for all relocationEntry in objFile.relocationTable do
5:     if IsParameter(symbol, relocationEntry) then
6:         CreateParameterSubstituion(symbol, relocationEntry)
7: CreateFunctionFooter()
```

## V. Experimental Results

We implemented ISSs for SH, MIPS64, and ARM with our framework to measure the productivity. We also experimented with them, and compared the speedup of the generated DBTs to their original interpreters without decoded instruction caches implemented in our previous work [21]. The generated DBTs are equipped with translation chaining introduced in [11]. We performed all of the measurements presented in this paper on a 64-bit Core-i7 2.8 GHz, Linux-based desktop machine with 16-GB main memory. We used gcc 4.9.2 with the -O2 flag to build the ISSs and the translation templates. We also used gcc 4.9.3 with the -O2 flag to cross-compile programs for SH, MIPS64, and ARM.

### A. Productivity

We measured the code metrics of the behavior description for the DBTs. Table I shows their code metrics and the interpreters included in GDB. The GDB results come from armemu.c, which includes the implementation of ARMv7 (excluding FPU instructions).

Lines of code (LOC) do not include comments or blank lines. The LOC per instruction is LOC divided by the total number of implemented instructions. When the target is ARM, the LOC per instruction of our framework is 12, which is 40% smaller than GDB. The reason reflects how the instruction decoders are implemented. In GDB, since the behavior and decoder descriptions are mixed, they cannot be easily separated. On the other hand, in our framework, the behavior description does not contain an instruction decoder that is generated from the decoding entries. As a result, the behavior description for our framework is simpler than GDB.

When comparing different instruction sets within the ISSs of our framework, the largest LOC per instruction is 12 for ARM and the smallest is 5 for MIPS64. This result is explained by the different complexities of the instruction sets.

Most ARM instructions have predicates that determine the instruction's execution. Furthermore, many ARM instructions have shifted operands. Hence, the ARM description is more complex than the other two instruction sets without these features.

In terms of instruction behaviors, some instructions such as division in SH are more complex than the equivalent instruction of MIPS64. Hence, LOC per instruction in SH is slightly larger than MIPS64.

TABLE I
Code metrics

| | Our framework | | | GDB |
|---|---|---|---|---|
| | SH | MIPS64 | ARM | ARM |
| Implemented instructions | 152 | 234 | 175 | 170 |
| LOC | 870 | 1,257 | 2,111 | 3,461 |
| LOC per instruction | 6 | 5 | 12 | 20 |

### B. Simulation Speed

We used the Dhrystone and CHStone [22] benchmarks to confirm the speedup of the generated DBTs. CPIs are shown in Table II. In this paper, CPI denotes the host cycles per target instruction. The lowest CPI was 88.87, which was measured when executing dfadd on the ARM interpreter. The best CPI was 4.11, which was measured when executing sha on the SH DBT.

The speedup of DBTs compared to their original interpreters is shown in Fig. 7. The minimum speedup was 1.4, which was measured when executing adpcm on the ARM DBT. The maximum speedup was 13.4, which was measured when executing sha on the SH DBT. Regardless of the target instruction set or the program, we confirmed that speedup was achieved. However, its degree was different among the target instruction sets or programs.

When comparing the differences among instruction sets, SH speedup exceeds the speedup of the other two instruction sets. Since DBTs can skip instruction decoding, the simpler the behavior function is, the more speedup is achieved. In other words, if the computation of the behavior function is heavier than that of the instruction decoding part that can be skipped in DBTs, less speedup is achieved.

The difference between SH and MIPS64 is the bit width of the instruction set. While SH is a 32-bit instruction set, MIPS64 is a 64-bit instruction set. As a result, the behavior functions of MIPS64 are heavier than SH.

The behavior functions of ARM are heavier than SH and MIPS64, since most ARM behavior functions have if-

TABLE II
CYCLES PER INSTRUCTION

| Benchmark | Interpreter | | | DBT | | |
|---|---|---|---|---|---|---|
| | SH | MIPS64 | ARM | SH | MIPS64 | ARM |
| dhrystone | 61.95 | 49.86 | 76.16 | 9.09 | 10.00 | 27.74 |
| adpcm | 53.95 | 52.76 | 69.95 | 8.25 | 8.37 | 49.56 |
| aes | 52.54 | 46.24 | 72.80 | 8.62 | 11.29 | 37.30 |
| blowfish | 52.46 | 43.23 | 72.45 | 4.74 | 5.75 | 40.60 |
| dfadd | 69.66 | 47.93 | 88.87 | 12.22 | 13.10 | 55.22 |
| dfdiv | 66.41 | 53.54 | 79.45 | 10.21 | 12.12 | 47.78 |
| dfmul | 62.78 | 53.64 | 88.57 | 9.99 | 14.01 | 49.12 |
| dfsin | 67.09 | 51.41 | 76.67 | 8.78 | 7.16 | 53.51 |
| gsm | 61.95 | 49.86 | 70.23 | 8.93 | 7.41 | 35.38 |
| jpeg | 54.35 | 47.82 | 71.77 | 7.07 | 8.46 | 29.77 |
| mips | 47.21 | 40.14 | 63.94 | 6.40 | 8.61 | 24.89 |
| sha | 54.86 | 41.69 | 71.55 | 4.11 | 4.87 | 24.63 |

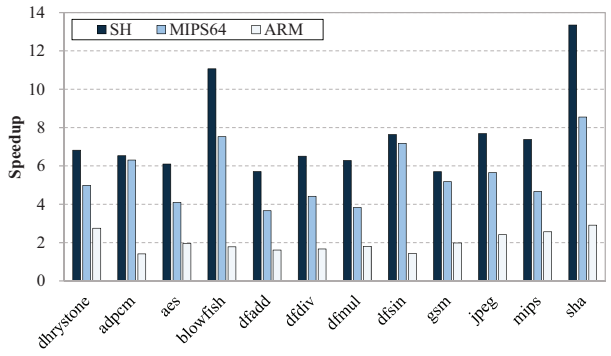


Fig. 7. Speedup

statements to check whether the execution condition is satisfied. Therefore, the speedup of ARM is smaller than SH and MIPS64.

The factor of the differences among the benchmark programs is the DBB length. When the target is SH, the relation between the DBT's speedup and instructions per block chain is shown in Fig. 8. Similar tendencies were also measured when ARM and MIPS64 are the targets. In Fig. 8, the max speedup was 13.4 with 302 instructions per block chain, which is also the max value, and the second largest speedup was 11.1 with 196 instructions per chain, which is also the second largest value. The results show that the longer the DBBs of the target program, the more speedups tend to be achieved.

## VI. CONCLUSION

This paper addresses the difficulty of implementing a dynamic binary translator (DBT). Our framework compiles instruction specific behavior functions into an object file and generates translation functions by extracting translation templates from the object file. As a result, there is no need to manually code the translation functions for a DBT. We implemented DBTs for SH, MIPS64, and ARM with our framework and confirmed that the resulting DBTs generated from a reasonable amount of description for interpreters are 1.4 to 13.4 times faster than their original interpreters. The speedup compared to the interpreters depends on the target instruction set and the lengths of the dynamic basic blocks (DBBs). The simpler the target instructions are or the longer the DBBs of the target program, the more speedups tend to be achieved.

## REFERENCES

[1] H. Theiling, "Generating decision trees for decoding binaries," in LCTES '01, pp. 112–120.

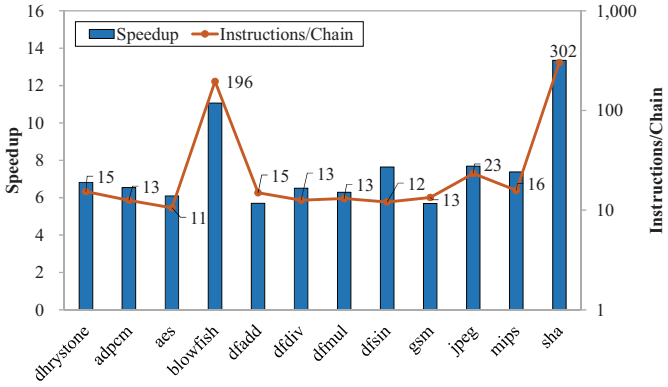


Fig. 8. Relation between speedup and instructions per block chain (SH)

[2] W. Qin and S. Malik, "Automated synthesis of efficient binary decoders for retargetable software toolkits," in DAC '03, pp. 764–769.
[3] N. Fournel, L. Michel, and F. Pétrot, "Automated generation of efficient instruction decoders for instruction set simulators," in ICCAD '13, pp. 739–746.
[4] K. Okuda and H. Takeyama, "Decision tree generation for decoding irregular instructions," in DATE '16, pp. 1592–1597.
[5] T. Austin, E. Larson, and D. Ernst, "Simplescalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb 2002.
[6] G. Braun, A. Nohl, A. Hoffmann, O. Schliebusch, R. Leupers, and H. Meyr, "A universal technique for fast and flexible instruction-set architecture simulation," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 23, no. 12, pp. 1625–1639, Nov. 2006.
[7] S. Pees, A. Hoffmann, and H. Meyr, "Retargetable compiled simulation of embedded processors using a machine description language," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 4, pp. 815–834, Oct. 2000.
[8] J. Zhu and D. D. Gajski, "A retargetable, ultra-fast instruction set simulator," in DATE '99.
[9] F. Engel, J. Nuhrenberg, and G. Fettweis, "A generic tool set for application specific processor architectures," in CODES '00, pp. 126–130.
[10] M. M. Hamayun, F. Petrot, and N. Fournel, "Native simulation of complex vliw instruction sets using static binary translation and hardware-assisted virtualization," in ASP-DAC '13, pp. 576–581.
[11] B. Cmelik and D. Keppel, "Shade: A fast instruction-set simulator for execution profiling," in SIGMETRICS '94, pp. 128–137.
[12] E. Witchel and M. Rosenblum, "Embra: Fast and flexible machine simulation," in SIGMETRICS '96, pp. 68–79.
[13] F. Bellard, "Qemu, a fast and portable dynamic translator," in ATEC '05, pp. 41–46.
[14] M. R. Hartoog, J. A. Rowson, P. D. Reddy, S. Desai, D. D. Dunlop, E. A. Harcourt, and N. Khullar, "Generation of software tools from processor descriptions for hardware/software codesign," in DAC '97, pp. 303–306.
[15] G. Hadjiyiannis, P. Russo, and S. Devadas, "A methodology for accurate performance evaluation in architecture exploration," in DAC '99, pp. 927–932.
[16] S. Pees, A. Hoffmann, V. Zivojnovic, and H. Meyr, "Lisa machine description language for cycle-accurate models of programmable dsp architectures," in DAC '99, pp. 933–938.
[17] M. Reshadi, N. Dutt, and P. Mishra, "A retargetable framework for instruction-set architecture simulation," *ACM Trans. Embed. Comput. Syst.*, vol. 5, no. 2, pp. 431–452, May 2006.
[18] K. Okuda, S. Kobayashi, Y. Takeuchi, , and M. Imai, "A simulator generator based on configurable vliw model considering synthesizable hw description and sw tools generation," *Proc. Workshop on Synthesis and System Integration of Mixed information Technologies, Apr. 2003*, pp. 152–159, 2003.
[19] R. Kassem, M. Briday, J.-L. BéChennec, G. Savaton, and Y. Trinquet, "Harmless, a hardware architecture description language dedicated to real-time embedded system simulation," *J. Syst. Archit.*, vol. 58, no. 8, pp. 318–337, Sep. 2012.
[20] J. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes (The Morgan Kaufmann Series in Computer Architecture and Design)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.
[21] K. Okuda and H. Takeyama, "Design and implementation of generation framework for instruction set simulators," *IPSJ Journal*, vol. 57, no. 8, pp. 1718–1736, Aug. 2016.
[22] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii, "Chstone: A benchmark program suite for practical c-based high-level synthesis," in ISCAS '08, pp. 1192–1195.