Electronic Preprint for Journal of Information Processing Vol.30

Regular Paper

Automatic Optimize-time Validation for Binary Optimizers

Motohiro Kawahito[1,a)] Reid Copeland[2] Toshihiko Koju[1] David Siegwart[3]
Moriyoshi Ohara[1]

Received: June 17, 2021, Accepted: September 9, 2021

Abstract: We propose an approach called automatic optimize-time validation for binary optimizers. Our approach does not involve executing the whole program for validation but selecting a small part of code (1 to 100 instructions) for the target test code. It executes the target code and its optimized code with several input data during binary optimization. One benefit is that we can test a small part of an actual customer's code during binary optimization. Our approach can be used to test several input data not included in the target code, which is the most beneficial aspect of the approach. If the results are the same after execution, we will use the optimized code for the final output code. If the results differ, we can consider a couple of option, e.g., while developing a binary optimizer, we can abort the compilation with an error message to easily detect a bug. After a binary optimizer becomes generally available, we can use the input code for the final output code to maintain compatibility. Our goal is for the output binary code to be compatible, fast, and small. We focused on how to improve compatibility in this study. We implemented our approach in our binary optimizer and successfully detected one new bug. We used a very small binary program to observe the worst case of increased compilation time and output binary file size. Our implementation showed that our approach increases optimization time by only 0.02% and output binary file size by 8%.

Keywords: validation, binary optimizer, code generation

1. Introduction

A binary optimizer can exploit new hardware features [8] for old binary code without its source code. In fact, previous binary optimizers showed good performance improvements. However, optimizations (regardless of the existence of source code) always have a compatibility problem. Customers require the same behavior of optimized binary code as the input binary code. Therefore, they do not want to recompile their source code even on new hardware. Our target programming language is COBOL, which is often used in financial applications, so compatibility is very important. Compiler developers verify code by conducting many tests. However, the generated code highly depends on a combination of program sentences. This is because optimizations are performed for those sentences. Therefore, it is not easo to cover all possible combinations of the code through inhouse tests. We give such an example in Fig. 7. Because of this, we wanted to test an actual customer's code during binary optimization.

We propose an approach called automatic optimize-time validation for binary optimizers. A binary optimizer has an advantage in that the input code is also executable. Our approach involves selecting a small part of code (1 to 100 instructions) for the target code. It then executes the target code and its optimized code with several input data during binary optimization. If the results are the same, we will use the optimized code for the final output code. If the results differ, we can consider various options, e.g., aborting the compilation with an error message or using the input code for the final output code to maintain compatibility.

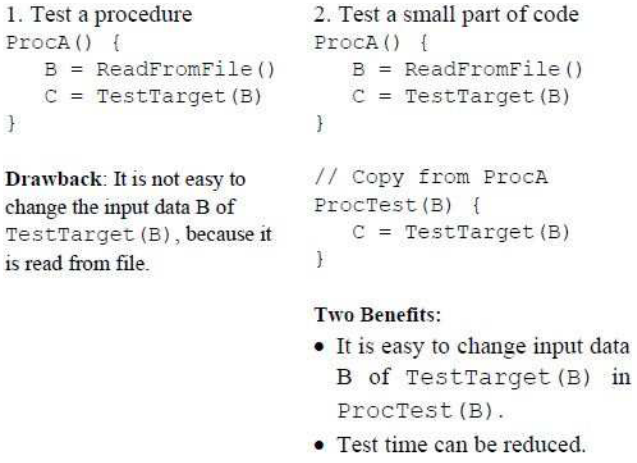


Fig. 1 Two benefits of testing small part of code.

A significant advantage of our approach is that we can test a small part of an actual customer's code during binary optimization. **Figure 1** shows two benefits of testing a small part of code. If we test procedure ProcA() in Fig. 1-1, it will not be easy to change input data B for TestTarget(B) because it is read from a file. In contrast, if we copy a test target of code to ProcTest(B) and test it as shown in Fig. 1-2, it will be easy to change input data B. At the customer site, we cannot do such a transformation at the customer-source-code level, so we do this at the binary-code level automatically. Therefore, we are able to test several input data not included in the original binary code. With this approach, the test time can also be reduced, so we can try more variants of input

[1] IBM Research - Tokyo, Chuo, Tokyo 103–8510, Japan
[2] IBM Canada, Toronto, Canada
[3] IBM United Kingdom Limited, Hursley, GB
[a)] jl25131@jp.ibm.com

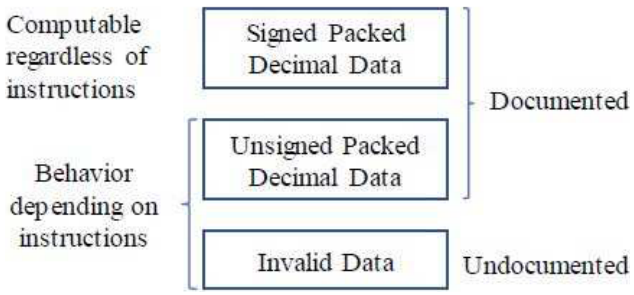


Fig. 2 Characteristics of BCD data on IBM Z architecture.

data.

We are currently focusing on decimal instructions for the target validation. There are two reasons for this. One reason is that several decimal instructions are very complex. The other reason is that there are three types of binary coded decimal (BCD) data: signed, unsigned, and invalid.

Figure 2 shows the characteristics of these three types. The behavior of each instruction is documented for signed and unsigned data. However, for unsigned and invalid data, the behavior depends on each instruction. Moreover, the behavior for invalid data is undocumented. Our COBOL compiler generates instructions for signed data even if a variable is declared as unsigned. Therefore, a compiler developer tends to consider only signed data. However, in a COBOL program, we can forcibly assign any value to a variable. Historically, we have experienced many problems related to unsigned and invalid data.

Our goal is for the output binary code to be compatible, fast, and small, and we specifically focused on how to improve compatibility in this study. Our goal of compatibility is for the execution results of input and optimized code to be the same. From this perspective, the binary optimizer has the benefit of compatibility compared to compiling from source code to binary, because the results of binary code compiled by different compilers differ in some cases, particularly for invalid data area.

Our approach is a type of specification-based testing [2], [4]. It is also known as black-box testing. This is because we do not know the detailed implementation of our target complex decimal instructions, particularly for behaviors for invalid data. One difficulty with our approach is identifying where the inputs and outputs (e.g., registers or memory areas) are in the final generated code. In particular, the optimized code is often a different algorithm from the original instruction and includes work areas, so we need to distinguish between the real output and work area. For selecting input data, we use the detailed specifications of the BCD data format.

We implemented our approach in our binary optimizer and successfully detected one new bug. We used a very small binary program to observe the worst case of the optimization time and the output binary file size increasing. Our implementation showed that our approach increases optimization time by only 0.02% and output binary file size by 8%.

1.1 Our Contributions

- **New optimize-time validation approach for binary optimizers:** Our validation approach is used to execute a part of a binary code with several input data during binary optimization then automatically select either the input code or optimized code based on the validation results in our binary optimizer. This mechanism allows us to test the optimized version of an actual customer's binary code in the binary optimization phase. To the best of our knowledge, this is the first approach of automatically selecting code during binary optimization based on the validation results.
- **Method of executing a part of a binary code as a function:** This method is necessary to execute a part of a binary code with several input data not included in the input code. When we want to execute a part of a binary code as a function, we need to collect input and output information of the target binary code. We discuss how to execute a part of binary code as a function in Section 2.

The rest of the paper is organized as follows. In Section 2, we describe our approach. In Section 3, we introduce actual examples of applying our approach. In Section 4, we present some of the results from the implementing our approach in our binary optimizer. We summarize previous studies in Section 5 and give concluding remarks in Section 6.

2. Overview of Our Approach

This section describes how we carry out automatic validation in our binary optimizer.

2.1 High-level Flow Diagrams

Koju et al. [11] proposed an optimization in our binary optimizer. We modified it to validate its optimization results. **Figure 3** shows two high-level flow diagrams of our approach. Note that the grey boxes are new or modified components for our approach compared to our original binary optimizer [11]. We modified both our compiler for high-level programming language and a binary optimizer.

Figure 3 (1) shows a diagram of our compiler for high-level programming language. We decide on the target validation code sequence based on each node of our intermediate language (IL). We chose a complex and decimal IL node for applying validation. The number of generated code of the chosen IL node is 1 to 100 instructions. After the code-generation phase, we create mapping from each IL to generated instructions. This mapping information is useful for converting instruction sequence to ILs in our binary optimizer. In addition, we create input and output information for each target validation code sequence. We use this information to execute the target code sequence as a function in the binary optimizer. Finally, we output the whole optimized binary code and additional information for some ILs, such as the mapping and I/O information mentioned above. We call this 'smart binary'.

Figure 3 (2) shows a diagram of a binary optimizer. Our automatic validation approach assumes 'smart binary' as the input binary. Otherwise, this feature is disabled. As we mentioned, we utilize the mapping information to convert instruction sequence to ILs. In the IL-generation phase, for each validation target, we create fast and slow paths and generate optimized and input codes for the fast and slow paths, respectively. We explain it in more details in Section 2.2. We select each for the final output code

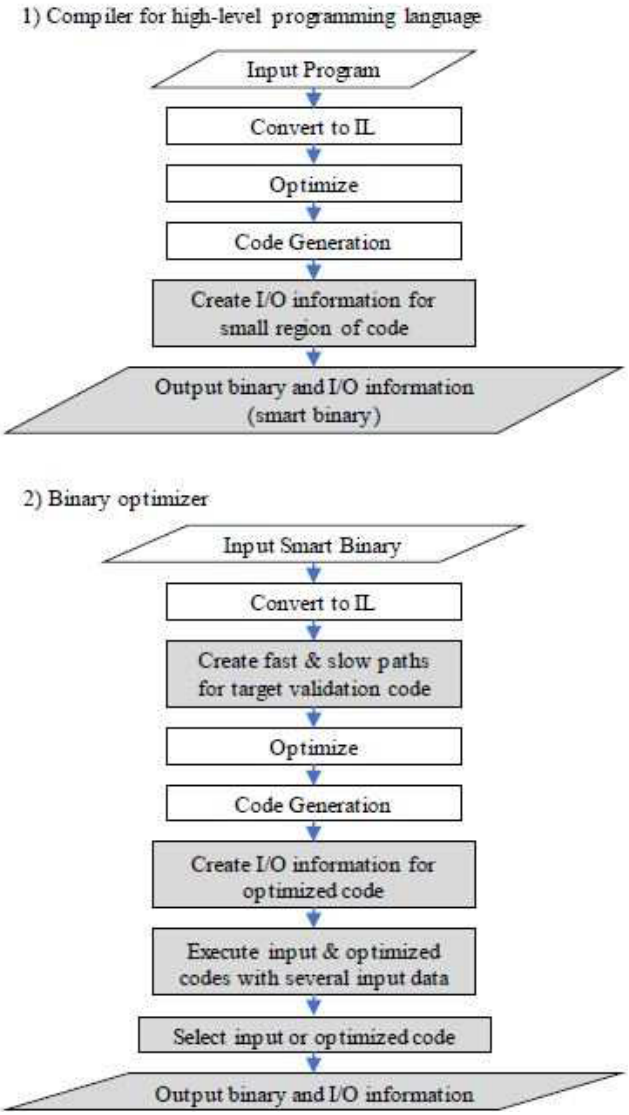


Fig. 3 High-level flow diagrams of our approach (Grey boxes are new or modified components).

based on the validation result. For the input code, we read I/O information from the input smart binary. After code generation, we create I/O information for the optimized code. Next, we execute both input and optimized codes with several input data. To execute a code sequence as a function, we need I/O information. If the results are the same, we use the optimized code for the final output code. If the results differ, we use the input code for the final output code to maintain compatibility. As mentioned above, we can abort this optimization with an error message to quickly detect a bug.

2.1.1 Benefit of Binary Optimizer Compared to Compiling Source Code

For optimization effectiveness, compiling source code is usually better than using a binary optimizer. However, for compatibility, a binary optimizer is better than compiling source code because the optimized binary code must behave the same as that of the input binary code.

For example, assume that two binary codes `binA` and `binB` are compiled from the source code by different compilers. These two binaries behave the same for the documented specifications.

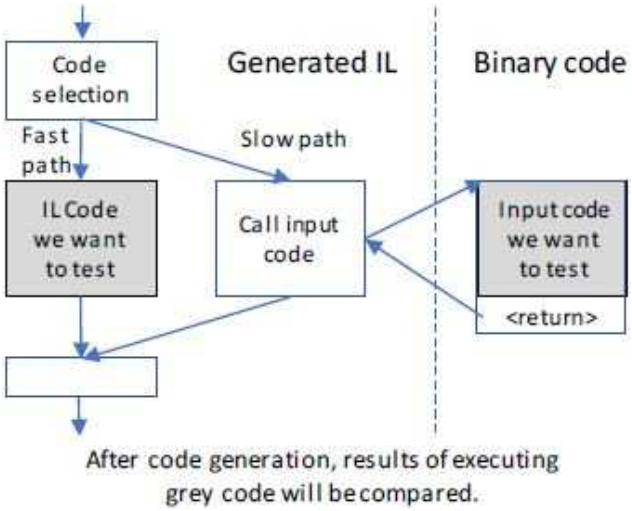


Fig. 4 How we generate IL code for our validation.

However, for invalid data, `binA` causes an exception but `binB` does not. We experienced such a problem in which the optimized code will need to behave the same as that of the input binary code. A previous study [6] also showed such an example. That is, for these invalid data, `optBinA` must cause an exception but not for `optBinB`. This cannot be achieved by compiling source code.

Therefore, a binary optimizer is beneficial in terms of compatibility. A use case scenario is as follows:

(1) A customer generates a smart binary B from the source code by using our compiler for high-level programming language for the computer architecture X.
(2) After several years, a new computer architecture Y becomes generally available. We also develop a binary optimizer tuned for Y.
(3) The customer reoptimizes B by using our binary optimizer in step 2. Our binary optimizer compares the results of executing B and its optimized code on Y. Finally, our optimizer generates compatible and optimized binary code by using a technique described later in this paper.

2.2 Creating Fast and Slow Paths in IL Level

Figure 4 shows how we generate IL code for our validation. Along the fast path, we generate IL code corresponding to the target code we want to test. Along the slow path, we generate a call IL node to call the input snippet code copied from the input binary (right-hand side of dashed line). Our binary optimizer optimizes and generates the code for the left-hand side of dashed-line. After IL-level optimizations and code generation, the results of executing the binary code of the grey blocks in Fig. 4 are compared.

We generate a No Operation (NOP) instruction for `Code selection` first, so the fast path will be executed. If the validation result for this target code is failure, we generate a jump instruction to the slow path for `Code selection`. Currently, the final code includes both fast and slow paths, but code execution always goes to either direction at runtime. For future work, we plan to generate a single path in the final code to reduce the code size.

For the input binary code, we support the case in which it is

not contiguous. Our smart binary information has map information from an IL node to instructions. Our approach first initializes the copy buffer by NOPs then copies each instruction while maintaining its offset. Information of input and output registers and memory areas are included in the input smart binary, so they are initialized and referred before and after calling the input code.

2.3 Creating I/O Information

To execute a part of a binary code as a function, it is important to create input and output information (e.g., registers and memory areas) of the target binary code.

Executing a part of a binary code seems like a trace-based compilation approach [1]. In this section, we explain a significant difference between that approach and ours. For trace-based compilation, the existing code writes input data before execution and reads the output data after execution. Therefore, we do not need to consider what the inputs and outputs are.

In contrast, to execute a part of a binary code as a function, we need to analyze what the inputs and outputs are. We devised the following seven sets by using a backward dataflow analysis [15] and information from the code generator.

- INR: Set of input parameters on registers at an entry point of the target code sequence
- INM: Set of input parameters on memory areas at an entry point of the target code sequence
- INM_CONST: Set of memory areas having constant data
- OUTR: Set of output registers of the target code sequence
- OUTM: Set of output memory areas of the target code sequence
- WORK: Set of the other memory areas accessed by the target code sequence
- WORK_REG: Set of registers used in the target code sequence except for INR and base registers

An element of a set of memory areas will have information of the effective address and its access size. Vector registers in INR or OUTR will have information of an access area in vector registers. Binary optimizers or simulators often map registers in the input code to virtual registers in memory. In this case, we treat the load or store for a virtual register in memory as a register access in the input code.

Basically, these sets can be computed by conducting a liveness analysis [15]. However, it is not easy to analyze memory accesses from pure binary code. In particular, binary code sometimes includes temporary memory areas for computations. The use of such areas is often different between the input code and optimized code. To validate the execution results, we want to compare only the real outputs of the target code; thus, we need to distinguish which areas are temporary or real outputs. To distinguish them by using an analytical approach, we need to conduct a whole program analysis to find if a memory area will be read after the target code sequence. Instead, we obtain information from the code generator about real inputs and outputs because it determines which memory area, temporary or real, the output is in.

For input binary code, our compiler of high-level programming language embeds this I/O information in a smart binary, as illustrated in Fig. 3. Our binary optimizer reads it and creates I/O information for the optimized code. Our validation technique executes both input and optimized codes based on this I/O information during binary optimization.

2.4 Execute Input and Optimized Codes with Several Input Data

It is not difficult to execute a binary code with a single input datum included in the input code. However, our goal was to execute a binary code with several input data not included in the input code. The following are steps to execute both input and optimized codes with several input data:

(1) Compute the minimum and maximum offsets for each base register from the union of INM, INM_CONST, OUTM, and WORK.
(2) Compute the allocation size and assign an address for each base register from the minimum and maximum offsets computed in step 1.
(3) Prepare several input test data based on the binary code.
(4) Assign values into elements in INR, INM_CONST, INM, WORK, and WORK_REG into registers or allocated memory area in step 2. We assign constant values for INM_CONST and assign input test data for INR and INM. We initialize WORK and WORK_REG by a specific pattern (Example: 0xdeadbeef for 4-bytes).
(5) Execute the input and optimized codes with the data assigned in step 4.
(6) Compare OUTR and OUTM of the input and optimized codes.
(7) Iterate steps 4 to 6 for each input datum.

With these steps, we can execute a part of a binary code as a function. Step 4 handles input parameters for the target binary code, step 5 executes the target binary code, and step 6 handles outputs.

In addition to the above steps, we need to correctly handle relative instructions. Our target architecture (IBM Z) has relative instructions for computing the effective address by the current instruction address and offset. We copy the target instruction sequence to another allocated area to execute it as a function. Therefore, if there is a relative instruction in the target binary code, we need to copy the target memory area pointed by the effective address of that relative instruction while maintaining its offset.

3. Actual Example of Applying Our Approach

In the previous section, we gave an overview of our approach. In this section, we give an example of applying it. As we mentioned above, we focus on decimal instructions. First, we explain our target instruction for testing.

3.1 Background of Our Target Decimal Instruction

The IBM Z architecture [7] has ED (edit) and EDMK (edit and mark) instructions. They perform string formatting such as sprintf in C language. **Figure 5** shows an example using the ED instruction.

Figure 5 (a) shows COBOL code and its result. The declaration of the variable OUT specifies the format string. The MOVE state-

a) COBOL code and its result

```
IDENTIFICATION                          DIVISION.
PROGRAM-ID.                             TEST1.
ENVIRONMENT                             DIVISION.
DATA                                    DIVISION.
WORKING-STORAGE SECTION.
77 NUM PIC S9(4) COMP-3 VALUE 2000.
77 OUT   PIC **,**9.
PROCEDURE                               DIVISION.
MAIN.
    COMPUTE NUM = NUM * 2 + 321.
    MOVE NUM to OUT.    --- (1)
    DISPLAY OUT.
    STOP RUN.
```

Result: OUT has "*4,321".

b) Unoptimized assembly code of (1)

```
                        # [R13+288] is ***,**9.
ED    288(7,R13),236(R13) # Execute EDIT for NUM
```

Fig. 5 Example using ED instruction.

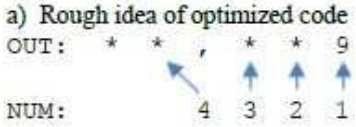
ment (1) copies NUM to OUT. As a result, OUT has "*4,321".

Figure 5 (b) shows the unoptimized version of assembly code corresponding to (a-1). Surprisingly, a single instruction performs such an operation. Before executing the ED instruction, the existing code copies the pattern string to R13+288, which points to the variable OUT. The ED instruction reads the pattern string in the first operand (R13+288) and overwrites it based on the second operand (R13+236), which points to the variable NUM. This means that the ED instruction interprets the pattern string at runtime. Therefore, this instruction is very complex and slow.

Koju et al. [11] proposed an optimization in our binary optimizer. Because the pattern string is known to be constant, this optimization performs a type of specialization. **Figure 6** (a) shows a rough idea of their optimized code. They converted the variable NUM into a string on the EBCDIC character code then copied each character to the appropriate position. If the number of digits of NUM is less than 4, the comma ',' will be replaced with an asterisk '*'. For example, if the variable NUM is 321, the variable OUT will be "***321". Figure 6 (b) shows the actual optimized code using vector instructions. Their experiments showed that this optimization improved performance for several benchmarks by 11% on average. This is because it can eliminate the overhead of interpreting the pattern string.

While such an optimization greatly improved performance, we need to consider the compatibility problem. Our goal is to compare the execution results of Fig. 5 (b) and Fig. 6 (b) with several input data. There are three difficulties in testing such decimal instructions and these optimized codes: (1) many variations in the pattern string for the `ED` instruction, (2) optimization effects between statements, and (3) three possible types of BCD data (signed, unsigned, and invalid). For (1), one can imagine that testing all the coverages of `sprintf` in C is very difficult. For (2), we actually observed a problem caused by optimizing computations between statements. In this case, because a problem occurred only with a specific code pattern, it is very difficult to find

a) Rough idea of optimized code



b) Optimized code

```
  VSTRL   VRF16,544(GPR10),0x8
  UNPK    536(5,GPR10),550(3,GPR10)
  LARL    GPR2,*+432                --- (1)
  VGBM    VRF20,0x0
  CDPT    FPR8,550(3,GPR10),0x8     --- (2)
  ESDTR   GPR4,FPR8
  VLEIB   VRF20,0xf0,15
  VL      VRF17,56(GPR2)
  VL      VRF19,72(GPR2)
  VLRL    VRF18,536(GPR10),0x4
  VN      VRF21,VRF18,VRF20
  VO      VRF18,VRF18,VRF20
  VPERM   VRF16,VRF17,VRF18,VRF19
  VREPIB  VRF19,0x0
  VL      VRF18,88(GPR2)
  VLEIB   VRF19,0x50,7
  VSLB    VRF18,VRF18,VRF19
  VLVGB   VRF19,GPR4,7
  VLEIB   VRF20,0x10,15
  VPERM   VRF17,VRF18,VRF18,VRF19
  VREPIB  VRF19,0xff
  VREPIB  VRF18,0x5c
  VTM     VRF21,VRF20
  VSLB    VRF19,VRF19,VRF17
  VSLB    VRF18,VRF18,VRF17
  BRC     MASK8(0x8), Label L0115
  VLEIB   VRF20,0xf0,15
  VTM     VRF21,VRF20
  BRC     MASK4(0x4), Label L0114
Label L0115:
  VLEIB   VRF18,0x5c,15
  VLEIB   VRF19,0xff,15
Label L0114:
  VSEL    VRF16,VRF18,VRF16,VRF19
  VSTRL   VRF16,288(GPR3),0x7
```

Fig. 6 Optimized code of Fig. 5 (b).

a problem through in-house tests. Our approach addresses these two difficulties by validating actual customer binary code during binary optimization. For (3), we prepared input test data for three data types. In the next section, we explain how we prepared the input test data.

Figure 7 shows an actual example related to (2) and (3) we experienced. It is not an example of the `ED` instruction but uses decimal instructions. In this COBOL program snippet (a), there are two `MOVE` statements. The first one internally converts from packed decimal data to zoned decimal data. The second one converts from zoned decimal data to binary data. The problem occurs by an incorrect IL-level optimization that replaces the source operand of the second `MOVE` from a zoned one `ZN` to a packed one `PK`.

Figure 7 (b) and (c) show instruction sequences of these two

a) COBOL program snippet

```
77 A1 PIC X(2).
77 PK REDEFINES A1 PIC 9(2) COMP-3.
77 ZN PIC 9(2).
77 BIN PIC 9(4) COMP.
      :
MOVE PK TO ZN.
      :
MOVE ZN TO BIN. => MOVE PK TO BIN.
      (incorrect optimization)
```

b) Instruction sequence for unoptimized version

```
UNPK 248(2,GPR9),224(2,GPR9)
OI   248+1(GPR9),0xf0
PKA  480(GPR13),248(2,GPR9)
CVB  GPR2,480+8(GPR13)
```

c) Instruction sequence for optimized version

```
UNPK 248(2,GPR9),224(2,GPR9)
OI   248+1(GPR9),0xf0
ZAP  480+1(8,GPR13),224(2,GPR9)
CVB  GPR4,480+1(GPR13)
```

Fig. 7 COBOL example in which optimizing computations between statements caused problem.

`MOVE` statements for unoptimized and optimized versions, respectively. The third instruction is different. The problem here is that the fourth instruction `CVB` supports only signed packed decimal data. For the unoptimized version (b), the output of the third instruction `PKA` is always a signed packed decimal. However, for the optimized version (c), the output of the third instruction `ZAP` is an unsigned packed decimal if the input is an unsigned one.

The optimized version correctly works only for signed data. However, this causes an exception if the variable `PK` has unsigned packed decimal data. Therefore, the program behavior differs between the input and optimized code. Note that the variable `PK` usually has signed packed decimal data in the common COBOL program context, even if it was declared as "unsigned". This significantly increases the difficulty of detecting this problem. To reproduce it, we can forcibly assign unsigned packed decimal data by using "`MOVE x'0000' TO A1`". This uses the fact that two variables `A1` and `PK` point to the same memory area by using the keyword `REDEFINES`.

If the second `MOVE` of the original COBOL program is "`MOVE PK TO BIN`", it will cause an exception for unsigned packed decimal data even with the unoptimized version. Again, let us emphasize that unsigned packed decimal data are uncommon in COBOL programs, but a programmer can forcibly assign unsigned data, as mentioned above. In this case, both the input and optimized codes behave the same, causing an exception, so there is no problem in a binary optimizer.

This problem is caused by incorrectly optimizing computations between statements in a binary optimizer. In other words, this problem was revealed by the existence of these two statements; thus, it is difficult to find such a problem through normal test approaches.

3.2 How We Prepared Input Test Data

Since our target decimal instructions perform on packed decimal data, we focused on the following three types of packed decimal data. For such data, a 4-bit value means one digit or sign.

- Signed packed decimal data: The rightmost 4-bit value means the sign (0xA to 0xF). The other 4-bit values mean a digit (0x0 to 0x9). For example, 0x12345B means -12345.
- Unsigned packed decimal data: All the 4-bit values mean a digit (0x0 to 0x9). For example, 0x123456 means 123456.
- Invalid data: Otherwise

As we explained in Section 3.1, the optimization result is affected by the number of digits. For example, there is no difference between "4321" and "1234" for testing this example. On the other hand, it is valuable to test "1234" and "0123". Thus, we prepared a program to generate the following input data:

- Signed data for each number of digits (plus and minus)
- +0, −0
- Unsigned data for each number of digits (Conditional)
- Invalid data for the largest number of digits

Preparing unsigned data is conditional. At first, we assume that all tipes of data will be correctly passed for Fig. 6 (b). Therefore, we prepared unsigned data. However, we found that this optimized code assumes the input code as signed data. This was a bug, and we fixed it. We disabled unsigned data preparation after fixing this bug. We will explain it in Section 3.2.1.

The sign value is rotated. For the plus sign, each (0xE, 0xA, 0xF, 0xC) is used. For the minus sign, each (0xB, 0xD) is used.

In the example in Fig. 5 (a), the number of digits of `OUT` is 5. As we mentioned, before fixing a bug, we generated all the types of BCD data for this validation. In this case, our input data generator prepares the following 19 types of input data.

- 0x12345E (plus data from here)
- 0x01234A
- 0x00123F
- 0x00012C
- 0x00001E
- 0x12345B (minus data from here)
- 0x01234D
- 0x00123B
- 0x00012D
- 0x00001B
- 0x00000E (+0)
- 0x00000B (−0)
- 0x123456 (unsigned data from here)
- 0x012345
- 0x001234
- 0x000123
- 0x000012
- 0x000001
- 0xABCDEF (invalid data)

Currently, a compiler developer needs to implement such an input data generator. For the complete test, we need to test from 0x000000 to 0xFFFFFF for this example. We limit the number of test data based on the knowledge of the instruction behavior; the number of digits is important rather than data value. For future work, we plan to explore automatic generation of input data based

a) I/O information for input code (Figure 5(b))

```
INM_CONST: {R13, 288, 7}
INM: { R13, 236, 3 }
OUTM: { R13, 288, 7 }
```

b) I/O information for optimized code (Figure 6(b))

```
INR { VR16, 3 }
OUTM: { R3, 288, 7 }
WORK: { R10, 544, 9 },{ R10, 536, 5 },
      { R10, 550, 9 }
```

Fig. 8 I/O information for input and optimized code.

on the target instruction sequence by using a variant of symbolic execution or concolic testing.

3.2.1 Changes of Input Test Data after Fixing a Bug

Previously, we always used the optimized code in Fig. 6 (b). We found that this code correctly works if the input data are signed packed decimal data. However, this causes an exception if the input data are unsigned packed decimal data. In the latest optimized code, if the input variable is not known to be always a signed one, we generate a test IL to detect if the input is a signed packed decimal.

In general, when our binary optimizer detects that the source operand always has the valid sign, it will generate the code assuming no unsigned data. In this case, our automatic validation should also suppress the generation of unsigned data because it may cause a false positive. The example in Fig. 6 (b) is generated by assuming no unsigned data. In this example, `CDPT` instruction at (2) supports only signed packed decimal data. This will cause an exception if unsigned packed decimal data arrive. After fixing this bug, our binary optimizer determines that the source operand always has the valid sign, so this code works. Therefore, our approach does not generate unsigned packed decimal data for this example after fixing this bug.

3.3 Applying Steps in Section 2.3 to Examples

Figure 8 shows I/O information for the input and optimized codes. In this figure, the elements of a register mean "{Register, size (from LSB)}". The elements of a memory mean "{Base register, offset, size}". We can obtain these sets by using the approach discussed in Section 2.3. For the optimized code in Fig. 6 (b), the base register GPR2 is computed by the `LARL` instruction at (1), which points to a literal pool address. Since we do not need to prepare data in this case, we do not create I/O information for GPR2. Because the code generator explicitly generates the `LARL` instruction at (1), we can easily find it. As a result, the I/O information is now very clear. For inputs, we assign test data to [R13+236] and VR16 for the input and optimized code, respectively. For outputs, we compare the results from [r13+288] and [r2+288] for the input and optimized code, respectively.

3.4 Applying Steps in Section 2.4 to Examples

Below are the results from applying each step in Section 2.4 to the different I/O information in Fig. 8.

(1) Compute the minimum and maximum offsets for each base register.
  (a) For the input code
    - R13: min=236 max=295
  (b) For the optimized code
    - R10: min=536 max=559
    - R3: min=288 max=295

(2) Compute the allocation size and assign an address for each base register
  (a) For the input code
    - R13: size=60 offset= -236
  (b) For the optimized code
    - R10: size=24 offset= -536
    - R3: size=8 offset= -256

(3) Prepare several input test data (as discussed in Section 3.2)

(4) Assign input test data in elements of sets
  (a) For the input code
    - Assign each test datum into [R13+236]
  (b) For the optimized code
    - Assign each test datum into VR16

(5) Execute the input and optimized codes. We execute the code snippets of Fig. 5 (b) and Fig. 6 (b).

(6) Compare OUTR and OUTM of input and optimized code. We compare [R13+288] and [R3+288] with length 7.

(7) Iterate steps 4 to 6 for each input datum.

Note that Fig. 6 (b) includes the `LARL` instruction at (1), which is a relative instruction. Therefore, we need to copy the target memory area of this `LARL` instruction.

We can validate the input and optimized codes with the test data presented in Section 3.2, which are not included in the original input code. These test data include the data that rarely arrive, and our approach can test this. This is the most beneficial aspect of our approach.

4. Implementation and Results

4.1 Bug Detected with Our Approach

Previously, we always generated the optimized code in Fig. 6 (b). As we mentioned in Section 3.2.1, this code correctly works if the input data are signed packed decimal data. However, this causes an exception, as shown in Fig. 6 (b)(2), if the input data are unsigned packed decimal data. In our implementation, our approach successfully detected this new bug. Our approach finds the difference of raising an exception between the original and optimized code for unsigned data. The original code does not cause an exception, but the optimized one does. Unsigned packed decimal data are uncommon in COBOL programs even if a variable is declared as unsigned, so detecting such a bug is difficult through a normal test. In the latest optimized code, if the input variable is not known to be always a signed one, we generate a test instruction to detect if the input is a signed packed decimal.

4.2 Optimization-time and File-size Increase with Our Approach

In this section, we show how much optimization time and binary file size increase with our approach. We used a very small binary program compiled from the COBOL program in Fig. 5 (a)

to observe the worst case. The results for optimization time are as follows:

- Whole optimization time without our approach: 802.0 ms
- Whole optimization time with our approach: 802.2 ms

We mesured the whole optimization time without and with our validation approach. Our approach increased optimization time by only 0.02%. With such a small optimization-time increase, we can execute two versions of target code sequences (input and optimized) with several input data.

The results for the binary file-size increase are as follows:

- Disable our validation: 49,152 bytes
- Enable our validation: 53,248 bytes

Our approach increased the output binary file size by 8%. We generated both the input and optimized codes shown in Fig. 4 in the final binary codes, even though only either path is executed. For future work, we plan to reoptimize the input binary code based on the validation results to generate a single path, though this would further increase the optimization time compared to that of the proposed approach.

5. Previous Approaches

There have been several binary optimizers, such as Dynamo [1], DynamoRIO [3], ADORE [14], COBRA [9], LLVM [12], and Automatic Binary Optimizer [11]. However, none used a validation technique to ensure the equivalence of the input and optimized codes. Our approach uses a validation technique during binary optimization then selects either the input code or optimized code based on the validation results.

A trace-based compilation approach is similar to our approach in the context of executing a part of the generated code. However, there are two significant differences. First, our approach automatically selects the input or optimized code for the final output code based on validation result. Second, for a trace-based compilation approach, given the target trace, existing code prepares its input data and uses its output data. Therefore, it can validate the target trace only with the single input datum existing in the input code. In contrast, our approach executes the target instruction sequence as a function with several input data that are not included in the input code. Therefore, we need to identify the places (in registers and memory areas) for inputs and outputs, as explained in Section 2.4.

As mentioned above, our approach is a type of specification-based testing [2], [4]. It is also known as black-box testing. We do not know the detailed implementation of our target complex decimal instructions but know the inputs and outputs for them. One difficulty with our approach is to identify where the inputs and outputs are in the final generated code. In particular, the optimized code often includes work areas, so we needed to distinguish between the real output and work area. For selecting input data, we used the detailed specifications of the packed-decimal-data format.

A formal verification technique proves that a program satisfies a formal specification of its behavior. For example, LLVM's Alive2 [13] successfully detects many bugs. The quality of our approach depends on input data. If a formal verification technique can be used, we can solve this problem. However, our target instruction is very complex and like a black box. As illustrated in Fig. 5 (b), the `ED` instruction performs string formatting, such as `sprintf` in C. It is a very complex instruction and interprets the given pattern string then overwrites it using the given parameter. Therefore, it is difficult to define a formal specification for this instruction. In addition, our optimization significantly changes its algorithm based on a specialization technique. We tried using the HOL theorem prover [5] to detect the equivalence of two different algorithms, but our conclusion is that it is very difficult to detect the equivalence of two different algorithms by using a formal verification technique.

Symbolic execution [10] is a technique of analyzing a program to determine what inputs cause each part of a program to be executed. If the target code consists of simple instructions, this technique will be effective in finding which values should be tested. It usually analyzes the conditions of conditional branches. However, for our examples, the input code is a single complex instruction. The optimized code includes two conditional branches, but the bug we found is not related to them. As mentioned above, our target architecture (IBM Z) has many complex instructions including hidden conditional branches. We observed problems caused by such hidden conditional branches. If we can break down these instructions into combinations of simple instructions, a symbolic execution technique may be effective.

Concolic testing [16] is a combination of a symbolic execution and concrete execution. It executes a target program with input data then records a path of execution. It attempts to execute another path. The next input value will be computed by the constraints of the new path. It iterates until all paths are executed. To apply this technique to our approach, there is the same problem of symbolic execution. Since our target architecture has many complex instructions including hidden conditional branches. This technique may be effective if we can break them down.

We have two concerns for simply applying symbolic execution or concolic testing for our purpose. First, an optimized code often includes a temporary work area. In Fig. 6 (b), the set `WORK` shows such work areas. In contrast, the input code in Fig. 6 (a) does not include such a work area. We do not want to treat it as the output of the target instruction sequence. Even when using these techniques, we would need to analyze what the real outputs are. Second, the target code is still very complex. The `ED` instruction should include a large loop to interpret the pattern string. Assuming that we can represent this loop by using simple instructions, we next need to automatically specialize this loop with the given constant pattern string. Finally, we want to find the equivalence between the specialized loop and optimized code. We are concerned if a better set of input data can be found with these approaches than with of our approach.

6. Conclusion

We proposed an automatic optimize-time validation approach for binary optimizers. This approach executes the target code and its optimized code with several input data during binary optimization. One benefit is that we can test a small part of an actual customer's code during binary optimization. We can also test input data not included in the target code. This is the most beneficial

aspect of our approach. If the results are the same, we will use the optimized code for the final output code. If the results differ, we can abort the optimization with an error message or use the input code for the final output code to maintain compatibility. We implemented our approach in our binary optimizer and successfully detected one bug. We used a very small binary program to observe the worst case of increased optimization time and output binary file size. Our implementation showed that our approach increased optimization time by only 0.02% and output binary file size by 8%.

For future work, we plan to reoptimize the input binary code based on the validation results to generate a single path, though this would further increase the optimization time compared to that of the proposed approach. We also plan to explore automatic generation of input data based on the target instruction sequence by using a variant of symbolic execution or concolic testing.

References

[1] Bala, V., Duesterwald, E. and Banerjia, S.: Dynamo: A transparent dynamic optimization system, *PLDI'00: Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (2000).

[2] Boyapati, C., Khurshid, S. and Marinov, D.: Korat: Automated testing based on Java predicates, *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA '02*) (2002).

[3] Bruening, D., Garnett, T. and Amarasinghe, S.: An infrastructure for adaptive dynamic optimization, *International Symposium on Code Generation and Optimization, CGO 2003* (2003).

[4] Goodenough, J.B. and Gerhart, S.L.: Toward a theory of test data selection, *IEEE Trans. Software Engineering* (1975).

[5] HOL, C.: HOL Interactive Theorem Prover.

[6] IBM: Migrating to Enterprise COBOL V6.

[7] IBM: *z/Architecture Principles of Operation*, IBM Corp. (2017).

[8] Kawahito, M., Komatsu, H., Moriyama, T., Inoue, H. and Nakatani, T.: A new idiom recognition framework for exploiting hardware-assist instructions, *Proc. 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp.382–393, ACM (2006).

[9] Kim, J., Hsu, W. and Yew, P.: COBRA: An Adaptive Runtime Binary Optimization Framework for Multithreaded Applications, *2007 International Conference on Parallel Processing* (*ICPP 2007*), *September 10-14, 2007, Xi-An, China*, p.25, IEEE Computer Society (online), DOI: 10.1109/ICPP.2007.23 (2007).

[10] King, J.C.: Symbolic execution and program testing, *Comm. ACM* (1976).

[11] Koju, T., Copeland, R., Kawahito, M. and Ohara, M.: Re-constructing high-level information for language-specific binary re-optimization, *Proc. 2016 International Symposium on Code Generation and Optimization, CGO 2016, Barcelona, Spain, March 12-18, 2016*, Franke, B., Wu, Y. and Rastello, F. (Eds.), pp.273–283, ACM (online), DOI: 10.1145/2854038.2854057 (2016).

[12] LLVM, F.: The LLVM Compiler Infrastructure.

[13] Lopes, N., Lee, J., Hur, C.-K., Liu, Z. and Regehr, J.: Alive2: Bounded Translation Validation for LLVM (2021).

[14] Lu, J., Chen, H., Yew, P. and Hsu, W.: Design and Implementation of a Lightweight Dynamic Optimization System, *J. Instr. Level Parallelism*, Vol.6 (2004).

[15] Muchnick, S.S.: *Advanced compiler design and implementation*, Morgan Kaufmann Publishers, Inc. (1997).

[16] Sen, K., Marinov, D. and Agha, G.: CUTE: A concolic unit testing engine for C, *Proc. 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, Wermelinger, M. and Gall, H.C. (Eds.), pp.263–272, ACM (online), DOI: 10.1145/1081706.1081750 (2005).

Motohiro Kawahito received a Ph.D. degree in Information and Computer Science from Waseda university in 2004. He joined IBM Japan in 1991. He is a research staff member of Commercial Systems group at IBM Research - Tokyo. His experties are compiler optimizations particularly for using dataflow analysis.

Reid Copeland is Senior Technical Staff Member at IBM focusing on COBOL optimization and performance. He has 18 years experience developing a wide range of static and dynamic compilers at IBM for numerous languages, targeting many computer architectures. For the last decade he has been deeply involved in modernizing COBOL performance through new compiler optimizations, co-designing new Z hardware facilities and leading the development of a binary to binary optimizer, the IBM Automatic Binary Optimizer, for COBOL modules.

Toshihiko Koju is a research staff member of Commercial Systems group at IBM Research - Tokyo.

David Siegwart is a Senior Software Developer at IBM in Compiler Technology. He is currently working on the IBM Automatic Binary Optimizer for z/OS and has worked on various compilers since 2006, including IBM Enterprise COBOL for z/OS and the IBM Java Just-In-Time (JIT) compiler. He received a Ph.D. degree in Theoretical Physics from the University of Durham, United Kingdom in 1990 and is a Chartered Physicist and Member of the Institute of Physics.

Moriyoshi Ohara his research interests include workload optimized systems with a special focus on high performance commercial systems. He and his team are exploring various techniques on hardware accelerators, compilers, and performance characterization to accelerate the performance of commercial systems. He received a Ph.D. degree in Electrical Engineering from Stanford University in 1996 and is a Senior Member of Association for Computing Machinery (ACM).