Electronic Preprint for Journal of Information Processing Vol.27

Regular Paper

Design and Implementation of Superinstructions for JavaScript Virtual Machine Generation System for Embedded Systems eJSTK

Tomoya Nonaka[1,†1,a)] Tomoharu Ugawa[1,b)]

Received: February 18, 2019, Accepted: May 17, 2019

Abstract: Embedded systems generally have a small amount of memory and slow CPUs. Therefore it is desirable to increase the speed of JavaScript virtual machines (VMs) without increasing memory footprint. In this research, we introduce superinstructions, combinations of constant load instructions and arithmetic, logical, and relational (ALR) instructions to increase execution speed. Introducing superinstructions increases the size of a VM. Thus, we designed superinstructions so that they would share their implementation code with the ALR instructions, from which they are made. Furthermore, we simplified their type-based dispatching code through specialization to the datatypes of their constant operands. We developed a VM generator that creates VMs that have superinstructions in accordance with this approach and a compiler that compiles JavaScript programs using them.

Keywords: superinstruction, virtual machine, interpreter, JavaScript, embedded system

1. Introduction

Development of embedded systems has become popular in recent years. A variety of devices with poor processors and small memories for their sensor and communication facilities have come to be available for the Internet of Things (IoT). To support software development for these devices, some JavaScript VMs for embedded systems, such as eJS [1], [2], JerryScript [*1], and mJS [*2], have been developed.

Certain design decisions need to be made to keep virtual machines (VMs) small in embedded system. JavaScript VMs for embedded systems avoid just-in-time (JIT) compilation mechanisms, which require a large amount of memory. Rather, they are implemented as pure interpreters. This means they are slower executing than JavaScript VMs working in Web browsers [2]. Nevertheless, execution speed is important even for embedded systems. It is particularly important for battery-powered systems: having a higher execution speed allows their CPU return to sleep mode quicker and thereby save more energy.

In this research, we use superinstructions [3] to speed up the execution of a JavaScript VM while minimizing VM bloat. Our target is eJS. The eJS JavaScript system exploits the characteristic of embedded systems that each individual system executes a particular application to reduce the size of the VM. It generates a customized VM for each application that VM only has functionalities that the application actually uses. We introduce a new customization method to eJS, in which eJS generates VMs with superinstructions that are combinations of frequently executed VM instructions. In particular, we focus on combinations of constant loading instructions and arithmetic, logical, and relational operation instructions (ALR instructions).

The eJS JavaScript system comprises a eJS compiler and a VM generator eJSTK (embedded JavaScript Tool Kit). The compiler compiles a JavaScript program into intermediate code. The intermediate code is executed on eJSVM, the VM generated by eJSTK. eJSVM is designed as a reduced instruction set computer (RISC) style register machine. In the instruction set of eJSVM, ALR instructions take only registers as operands. This design simplifies the instruction interpreter and hence reduces the size of the VM. On the other hand, the VM has to load constants to registers by using constant loading instructions prior to carrying out operations on them. Thus, this instruction set has two disadvantages that affect execution time in comparison with an instruction set where constants are allowed to be operands. First, the VM executes more instructions, each of which requires an instruction dispatch. Second, memory accesses increase because constant loading instructions store constants to registers, which is implemented with an array, and the following ALR instructions loads them from the registers. If constants are allowed to be operands of ALR instructions, the instruction decoder would be able to store the constant to a local variable of the interpreter, which is likely to be allocated to a machine register, and the operation can use it. In this research, we combine frequently used combinations of constant loading instructions and ALR instructions to make superinstructions that take constants as operands.

We optimize the implementations of the superinstructions in two ways. First, we implemented each superinstruction so that it

[1] Kochi University of Technology, Kami, Kochi 782–0003, Japan
[†1] Presently with COLOPL, Inc., LTD., Shibuya, Tokyo 150–6011, Japan
[a)] nonaka@pl.info.kochi-tech.ac.jp
[b)] ugawa.tomoharu@kochi-tech.ac.jp

[*1] http://jerryscript.net
[*2] https://github.com/cesanta/mjs

shares its implementation with its original instruction to avoid VM bloat. Second, we simplified the type-based dispatching code of the superinstructions by leveraging the datatypes of constant operands. In JavaScript, operators are overloaded; that is, operators change their behavior depending on their operand datatypes. For example, a plus operator computes the sum of numeric operands, while it concatenates string operands. Therefore, the instruction interpreter has to perform a type-based dispatch, which selects an appropriate implementation based on the datatypes of operands. However, for superinstructions, the datatypes of constant operands can be decided statically. This allows us to simplify the type-based dispatching code. We utilize eJSTK's mechanism to generate interpreters that limit the operand datatypes of operators on the basis of user's specifications.

In consideration of the above in Section 3, we design the superinstructions in the way that allows the optimizations above. Then, in Section 4, we show five implementation strategies with different optimizations of the superinstructions. Some of these strategies implement superinstructions such that they share their implementations with their original instructions, while others give independent implementations to the superinstructions so that they can be specialized to the datatypes of their constant operands. We compare these implementation strategies in Section 6.

2. eJS

2.1 Overview

eJS [1], [2] is a JavaScript system that has a mechanism to reduce the VM footprint by generating specialized VMs for individual applications. In particular, the VM generator of eJS, eJSTK, specializes VM instructions to their operand datatypes. eJS supports a subset of ECMA Script 5.1 [4], excluding facilities that requires complicated mechanisms such as `eval`.

Figure 1 shows the structure of the eJS system. Users enumerate the possible datatypes to be given to each VM instruction as operands in the executions of the target application. These datatypes are described in the *operand specification*. From the operand specification, eJSTK generates a VM, eJSVM, with an optimized instruction interpreter. For this eJSVM, the eJS compiler compiles the application into a sequence of VM instructions (intermediate code). These VM instructions are executed on the eJSVM in the embedded systems.

eJS is a successor to a server-side JavaScript VM, and it inherits some features that are not suitable for embedded systems. For example, eJS assumes a 64-bit environment. Nevertheless, the results of this work are also applicable to eJS if it is modified to fit embedded systems in a future.

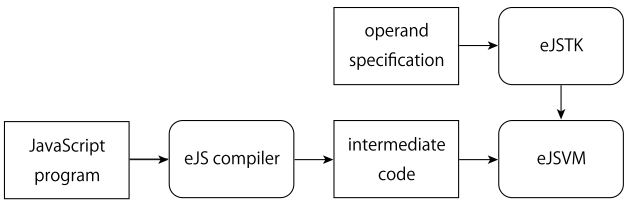

Fig. 1 Structure of eJS.

2.2 eJSVM

eJSVM is a JavaScript VM written in C. Its instruction interpreter is generated by the VM instruction generator, which is the heart of eJSTK. In the current implementation, fixed source code written in C is used for some parts of the eJSVM, including runtime systems, such as object manipulations and garbage collection, and built-in functions.

2.2.1 VM Datatypes

JavaScript is a dynamically typed language. Thus, each JavaScript value in eJSVM has a field to record its datatype. The internal datatypes of eJSVM, which we call *VM datatypes*, do not necessarily have a one-to-one correspondence to the datatypes defined in the JavaScript specification. The `Number` type of JavaScript, which is defined like a `double` of the C language, corresponds to two VM datatypes for efficiency: fixnum and flonum. fixnum is used for integers that fit within a fixed length of bits, and flonum is used for the others. The `Boolean` type, the `null` and `undefined` values of JavaScript belong to a single VM datatype: special.

eJSVM uses two kinds of tags to represent VM datatypes: *pointer tags* and *header tags*. Pointer tags are type tags recorded in spare bits in pointers. Header tags are type information stored in the headers of data in the heap. Because spare bits in pointers are limited, only limited VM datatypes are assigned unique pointer tags: datatypes of values embedded in words (fixnum and special) and frequently used datatypes (flonum and string) have unique pointer tags. Other datatypes share a common pointer tag, and are distinguished by their header tags. Note that eJSTK allows users to customize VMs by selecting datatypes that are distinguished by their pointer tags.

2.2.2 Instruction Set and Internal Instruction Format

eJSVM is a RISC-style register machine. It has distinct register sets for each function call. A function can use an unlimited number of registers. The register set is implemented as an array in C language, and the array is allocated to the execution stack. Registers are used for holding the intermediate values of computations of JavaScript expressions and for JavaScript local variables that are not referred to from closures. ALR instructions can only access registers. Local variables referred to from closures and object properties are accessed using dedicated instructions.

Every instruction occupies 64 bits in eJSVM, of which the first 16-bit field holds its opcode. However, eJSVM employs the threaded code technique for instruction dispatch, and so the opcode field is not used during executions.

Figure 2 shows the format of the ALR and constant loading instructions. An ALR instruction takes three register operands. The first one is the destination register, and the subsequent ones

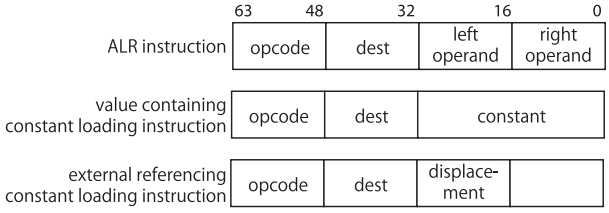

Fig. 2 Format of eJS instructions.

are source registers, holding values to be computed. In the rest of this paper, we call these two source operands *left* and *right operands*, respectively. Each operand occupies a 16-bit field.

eJSVM has constant loading instructions for fixnum, flonum, special, string and regexp datatypes. The regexp datatype is for regular expressions. These constant loading instructions fall into two categories: *value containing instructions*, which contain the constant values in the instructions, and *external referencing instructions*, which refer to external constants. The instructions loading fixnum and special are value containing instructions. The `fixnum` instruction, which loads a fixnum constant, has a 16-bit field for the destination register and a 32-bit field for the constant value.

The flonum, string, and regexp datatypes each require more than 32 bits. Thus, the constant loading instructions for them are external referencing instructions. The constant values to be loaded using these instructions are created when the VM loads the intermediate code. An external referencing instruction has a 16-bit field containing a displacement to the location of the constant value.

2.2.3 Type-based Dispatch

In JavaScript, operators are overloaded on operand datatypes. Thus, an ALR instruction performs type-based dispatch to select an appropriate implementation based on the operand datatypes. The VM instruction generator, which is the heart of eJSTK, generates a specialized type-based dispatching code according to the given operand specification.

Type-based dispatch is realized in the form of nested `switch` statements, each of which tests the pointer or header tag of an operand and selects the lower level `switch` statement or an appropriate implementation. The generated type-based dispatching code basically tests the pointer tags of the operands one by one, and then the header tags. It is optimized as follows. First, the generated `switch` statements do not have branch targets that cannot be selected according to the given operand specifications. Second, the branch targets of a `switch` statement are merged into a single branch target to avoid code duplication. Third, if a `switch` has only a single branch target as a result of the optimizations above, the `switch` statement is removed in order to skip the test of the tag.

For example, **Fig. 3** (a) shows the body of the unspecialized `add` instruction, where `v1` and `v2` are variables holding operands, `dst` is a macro to be expanded to the destination register, and `PTAG` and `HTAG` are macros to obtain the pointer and header tags, respectively. If the pointer tag of an operand is `GENERIC`, its header tag has to be tested to distinguish its datatype, because `GENERIC` is shared among multiple VM datatypes. Figure 3 (b) shows the specilalised code for the body of the `add` instruction that only works when both operands are fixnums or both operands are strings. This optimized instruction selects the implementation solely by testing `v1`. Furthermore, the test for its header tag is omitted because fixnum and string have unique pointer tags. As a result, type-based dispatch is realized with a single `switch` statement.

2.2.4 Instruction Interpreter

The instruction interpreter of eJSVM employs threaded code [5] for instruction dispatch. Each instruction comprises the following steps.

(1) Decode the instruction where the operand values are stored into interpreter's local variables.
(2) Execute the body of the instruction.
(3) Jump to the next instruction.

For example, the `add` instruction is shown below `I_ADD` in **Fig. 4**. First, it decodes the instruction; it reads the destination register number from the instruction and the values of the source registers; then it stores them in the interpreter's local variables `r0`, `v1`, `v2`. Here, `Register` and `JSValue` are the types of variables that have register numbers and JavaScript values, respectively. Next, it executes the body of the instruction that uses those local variables. The body of the instruction is generated as a separate file by the VM instruction generator. For `I_ADD` in Fig. 4, code like Fig. 3 is generated in `insn/add.c`.

The `fixnum` instruction, which loads a fixnum value, is shown below `I_FIXNUM` in Fig. 4. It first decodes the instruction as `add` does. Here, `get_small_immediate` is a macro to obtain the constant in the 32-bit field of the instruction. Next, it executes the body of the instruction, where it calls `cint_to_fixnum` to add a fixnum tag to the obtained integer and stores the tagged value to the register, the element of the register array, `regbase`, indexed by `r0`. The bodies of the constant loading instructions are not generated by the VM instruction generator. Rather, fixed code fragments are used.

```
ADD_HEAD:
switch(PTAG(v1)){
case FIXNUM: {
  switch(PTAG(v2)){
  case FIXNUM: {
    // addition of fixnum operands
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    dst = cint_to_number(s);
  } break;
  caes GENERIC: {
    switch(HTAG(v2)){
    case OBJECT: {
      // addition of fixnum operand and object-type operand
      v2 = to_number(context, v2);
      goto ADD_HEAD;
    } break;
...
}
```

(a) No specialization.

```
ADD_HEAD:
switch(PTAG(v1)){
case FIXNUM: {
    // addition of fixnum operands
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    dst = cint_to_number(s);
  } break;
case STRING: {
    // concatenation of string operands
    dst = ejs_string_concat(context, v1, v2);
  } break;
}
```

(b) Specialized add accepting only a pair of fixnums and a pair of strings.

Fig. 3 Body of add instruction.

```
I_ADD:
{
  // destination register
  Register r0 = get_first_operand_reg(insn);
  // left operand
  JSValue v1 = get_second_operand_value(insn);
  // right operand
  JSValue v2 = get_third_operand_value(insn);
#include "insns/add.inc"
}
NEXT_INSN_INCPC();
...
I_FIXNUM:
{
  // destination register
  Register r0 = get_first_operand(insn);
  // constant (fixnum)
  int64_t  i1 = get_small_immediate(insn);
  regbase[r0] = cint_to_fixnum((cint) i1);
}
NEXT_INSN_INCPC();
```

Fig. 4 Implementation of VM instructions.

2.3 VM Instruction Generator

The VM instruction generator receives an *instruction definition* as well as the operand specification. An instruction definition describes the specifications of an instruction in a domain specific language (DSL) for defining instructions. In particular, it describes, for each combination of datatypes of operands, a fragment of a C program to be executed for the datatype combination. The VM instruction generator generates the body of the VM instructions, such as the one in Fig. 3, by merging the type-based dispatching code, which consists of nested `switch` statements generated from the operand specification, and the C code fragments in the instruction definition.

3. Design of Superinstructions

In this research, we combine an ALR and a constant loading instructions into a superinstruction, which is an ALR instruction taking a constant operand. For example, a superinstruction `addfixreg` is formed by combining the `fixnum` instruction, which loads a fixnum value, and the `add` instruction that receives the loaded constant as its left operand. We use the word *the original ALR instructions* to refer to the ALR instruction that is a source of a superinstruction. We also call the operand of a superinstruction to which a constant value is to be given a *constant operand*. We give the superinstructions the names of the original ALR instructions followed by the abbreviations of the datatypes of the constant for constant operands and `reg` for the register operands.

3.1 Format of Superinstruction

Each operand field of an ALR instruction has 16 bits (see Section 2.2.2). For superinstructions, we store a bit sequence representing a constant value rather than the register number. All constant loading instructions specify constants in 16 bits, except for the `fixnum` instruction. `fixnum` loads a 32-bit fixnum value. Thus, we combine a `fixnum` instruction only when the value can be represented in 16 bits. We believe this restriction does not degrade the usefulness by much because integer constants are usually small. We also believe that it will be still useful if a future change to eJSVM reduces the number of bits for an operand field.

3.2 Limitations on Constant Operands

We impose the following limitations on constant operands of superinstructions to create room for optimizations of type-based dispatch of superinstructions.

First, we design the superinstructions so that the datatype of the constant operand is unique for each superinstruction. In the instruction set of eJSVM, all constant loading instructions are in one-to-one correspondence with the VM datatypes, except for the `number` instruction. The `number` instruction loads a fixnum (integer) value that exceeds 32 bits or a flonum (floating point number) value. However, we decided to disallow superinstructions from taking integer operands exceeding 16 bits, in Section 3.1. Thus, we combine a `number` instruction only when it loads a flonum.

Next, we restrict constant operands to those values whose datatypes can be determined solely by their pointer tags. This allows superinstructions and their original ALR instruction to share their implementation by arranging the tests in type-based dispatch so that the pointer tags of the operands are tested in an appropriate order (see Section 4.3 for details). In the default configuration, all constants of eJSVM other than regexps can be determined from their datatypes by using pointer tags.

A future change to eJSVM may reduce the number of bits for pointer tags; in that case, pointer tags would not be able to determine some constants. Nevertheless, eJS allows users to customize the VM in such a way that they can select datatypes that can be determined by pointer tags. Thus, the users can select the datatypes of the constants that they want to allow to be operands of superinstructions.

4. Implementation of Superinstruction

As we mentioned in Section 2, the code to process an ALR instruction of eJSVM comprises

- instruction decoding code,
- type-based dispatching code, and
- code for operations to be selected by type-based dispatch (we call this *operation code*).

Among these, the type-based dispatching code can be specialized if the operand datatype is fixed.

In this research, we use superinstructions to reduce execution time, while minimizing VM bloat. Our policy has two factors.

F1 To minimize VM bloat, the superinstructions share type-dispatching and operation code with their original ALR instruction.

F2 To shorten execution time, type-based dispatching code is specialized to the datatype of the constant operand.

These two factors have some contradictions. We consider various implementation strategies, each of which takes the factors in a different way.

If we follow only factor F1, the superinstructions and their original ALR instruction can share the entire of the type-based dispatching and operation code. We name this implementation

Table 1 Implementation strategies of superinstructions. IND, IND+SP, and SH represent "independent", "independent and specialized", and "shared", respectively.

implementation strategy	type-based dispatch	operation
S1 naive (Section 4.1)	IND	IND
S2 dispatch code sharing (Section 4.2)	SH	SH
S3 improved dispatch code sharing (Section 4.3)	partly SH	SH
S4 specialized dispatch (Section 4.4)	IND+SP	IND
S5 improved specialized dispatch (Section 4.5)	IND+SP	SH

strategy *dispatch code sharing*. In this strategy, a superinstruction jumps to the entry point of the type-based dispatch of its original ALR instruction.

The dispatch code sharing implementation strategy can be optimized by leveraging the datatypes of constant operands. This optimization is possible when the operand that the first `switch` statement of the type-based dispatch tests is a constant in a superinstruction. In this case, the superinstruction skips the first `switch` statement. In particular, it jumps to the appropriate branch of the `switch` statement rather than the entry point of the type-based dispatch. We call this implementation strategy *improved dispatch code sharing*.

If we follow only factor F2, we implement superinstructions separately from the original ALR instruction. In this implementation strategy, we can specialize the type-based dispatch of each superinstruction to the datatype of its constant. We call this implementation strategy *specialized dispatch*.

Even if we specialize type-based dispatch of superinstructions, superinstructions can share the operation code with their original ALR instruction. We call this implementation strategy *improved specialized dispatch*. In this implementation, superinstructions are type-based dispatched to an appropriate operation code of the original ALR instruction.

In addition to the four implementation strategies above, we will explain a *naive* implementation strategy, in which instructions do not share code and type-based dispatch is not specialized. **Table 1** summarizes the implementation strategies, and the following subsections describe them in details.

4.1 Naive Implementation Strategy

In the naive implementation strategy, a superinstruction has its own code. The code is the same as its original ALR instruction except for instruction decoding, which loads a constant.

For example, **Fig. 5** shows `addfixreg`, which is the superinstruction of `add` taking a `fixnum` as its left operand. `addfixreg` reads an integer value from the left operand field of the instruction, calls the `cint_to_fixnum` macro to add the `fixnum` type tag to the integer, and stores it in the local variable `v1`. The subsequent process is the same as `add`. Thus, the included file `insn/addfixreg.inc`, the body of the instruction, is almost the same as `insn/add.inc`, but has different label names that are used locally to avoid name collisions. The body of an instruction uses labels described explicitly in the instruction definition and labels generated by the VM instruction generator. The `DEFLABEL` and `USELABEL` macros in Fig. 5 add a prefix to the former labels. Instruction definitions have following form:

```
DEFLABEL(HEAD):
```

```
I_ADDFIXREG:
{
  // destination register
  Register r0 = get_first_operand_reg(insn);
  // left operand (integer constant)
  int16_t i1 = get_second_operand_int(insn);
  JSValue v1 = cint_to_fixnum(i1);
  // right operand
  JSValue v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADDFIXREG ## l
#define USELABEL(l) ADDFIXREG ## l
// addfixreg.inc and add.inc are the same except
// for label names
#include "insns/addfixreg.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

Fig. 5 Superinstruction in naive implementation strategy.

```
{
  Register r0;
  JSValue  v1, v2;
I_ADDFIXREG:
  r0 = get_first_operand_reg(insn);
  {
    int16_t i1 = get_second_operand_int(insn);
    v1 = cint_to_fixnum(i1);
  }
  v2 = get_third_operand_value(insn);
  goto I_ADD_BODY;
I_ADD:
  r0 = get_first_operand_reg(insn);
  v1 = get_second_operand_value(insn);
  v2 = get_third_operand_value(insn);
I_ADD_BODY:
#define DEFLABEL(l) ADD ## l
#define USELABEL(l) ADD ## l
#include "insns/add.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

Fig. 6 Dispatch code sharing implementation strategy.

for a label definition and

```
goto USELABEL(HEAD);
```

for jumping to a label. For the latter labels, the VM instruction generator adds prefixes.

4.2 Dispatch Code Sharing

In dispatch code sharing, a superinstruction uses `goto` to jump to the entry point of the type-based dispatch of its original ALR instruction. In this way, the superinstruction and its original ALR instruction share code. These two instructions are generated in the same scope, because they share the interpreter's local variables to store their operands.

Figure 6 shows the `addfixreg` instruction in the dispatch code sharing implementation strategy. `I_ADDREGFIX` is its entry point. It decodes the instruction, stores the operands to the interpreter's local variables `r0`, `v1`, and `v2`, and then jumps to the

label I_ADD_BODY placed before the body of the add instruction, insn/add.inc.

4.3 Improved Dispatch Code Sharing

In improved dispatch code sharing, a superinstruction jumps into the middle of the type-based dispatch of its original ALR instruction, if possible, to skip a switch statement.

Figure 7 (a) shows the addfixreg instruction in the improved dispatch code sharing implementation strategy. This is similar to the unimproved version, but the labels of the goto statements differ. In this improved strategy, it jumps to a label placed in the middle of the type-based dispatch in add.inc, as shown in Fig. 7 (b), the body of the add instruction. The left operand (the value of v1) of addfixreg can statically be determined to be a fixnum. Thus, it skips dispatch based on the datatype of its left operand and jumps to an appropriate entry point of dispatching on the basis of the right operand (the value of v2).

To make this possible, the VM instruction generator computes the paths of the tree of the nested switch statements that can be statically followed when parts of the operand datatypes are known, for all combinations of operand datatypes, and it adds a label to the end of each path. For the add instruction, it adds the label TLadd_fixnum_any where the left operand is known to be fixnum and the right is unknown, where any corresponds to an operand whose datatype is not known.

To create more chances to skip tests in type-based dispatch, we arrange the order of the tests of the operand datatypes. If the type-based dispatch tests the left and right operands in this order like, add.inc in Fig. 7 (b), a superinstruction whose right operand is a constant, but the left (*right-constant superinstruction*), cannot skip the type-based dispatch. Thus, the VM instruction generator creates a type-based dispatch testing the right operand before the left when such a superinstruction is added. When both left-constant and right-constant superinstructions are added, the user decides the priority. eJSVM tests the pointer tags of all operands; then, it tests the header tags, if necessary. This is because testing a header tag is costly, as it involves memory accesses. Because our design in Section 3.2 restricts the datatypes of constant operands to those that can be determined by pointer tags, rearranging the order of the tests of the pointer tags suffices for this purpose, and no extra header tag tests are needed.

```
{
  Register r0;
  JSValue  v1, v2;
I_ADDFIXREG:
  r0 = get_first_operand_reg(insn);
  {
    int16_t i1 = get_second_operand_int(insn);
    v1 = cint_to_fixnum(i1);
  }
  v2 = get_third_operand_value(insn);
  goto TL_add_fixnum_any;
I_ADD:
  r0 = get_first_operand_reg(insn);
  v1 = get_second_operand_value(insn);
  v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADD ## l
#define USELABEL(l) ADD ## l
#include "insns/add.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

(a) Implementation of a superinstruction.

```
switch (PTAG(v1)) {
case FIXNUM:
TLadd_fixnum_any:
  switch (PTAG(v2)) {
  case FIXNUM:
  TLadd_fixnum_fixnum:
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    regbase[r0] = cint_to_number(s);
    break;
 ...
 }
...
}
```

(b) Body of the original ALR instruction （add.inc）.

Fig. 7 Improved dispatch code sharing implementation strategy.

4.4 Specialized Dispatch

Our design in Section 3.2 guarantees that the datatype of the constant operand of every superinstruction is unique. This allows the VM generator to generate efficient type-based dispatching code specialized to the datatypes of constant operands.

Figure 8 (a) shows the addfixreg instruction in the specialized dispatch implementation strategy. Its body is generated using the VM instruction generator from the operand specification that restricts its left operand to fixnum. Figure 8 (b) shows the generated code. Because the left operand is restricted to fixnum, the dispatching code testing the left operand is omitted, and the nest of switch statements is shallower. This reduces both VM bloat and type-based dispatch time.

We have to be careful about the case where a superinstruction uses the code of its original ALR instruction. This happens when the instruction carries out a second type-based dispatch after converting the datatype of its constant operand. For example, in JavaScript, when a string operand and an integer operand are given to the plus operator, it converts the integer operand to a string and concatenates them. There are many other combinations of operand datatypes that cause type conversion. eJS handles these cases with compact code uniformly by jumping to the entry point of the outermost switch statement of type-based dispatch after type conversion (see Fig. 3 (a)). However, if the operand datatypes are restricted, the type-based dispatching code may not have an appropriate branch target for the converted datatype from the constant operand. For example, the body of the addregfix instruction cannot handle the left operand if it is converted to a string value because it is specialized to fixnum right operands.

To handle these cases, the labels in the original ALR instructions are used for the jump targets in the superinstructions if the labels are derived from the descriptions in the instruction definitions. For this reason, the USELABEL macro before including insns/addfixreg.inc adds the ADD prefix. Note that the la-

```
{
  Register r0;
  JSValue  v1, v2;
I_ADDFIXREG:
  r0 = get_first_operand_reg(insn);
  {
    int16_t i1 = get_second_operand_int(insn);
    v1 = cint_to_fixnum(i1);
  }
  v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADDFIXREG ## l
#define USELABEL(l) ADD ## l
#include "insns/addfixreg.inc"
#undef DEFLABEL
#undef USELABEL
  NEXT_INSN_INCPC();
I_ADD:
  r0 = get_first_operand_reg(insn);
  v1 = get_second_operand_value(insn);
  v2 = get_third_operand_value(insn);
#define DEFLABEL(l) ADD ## l
#define USELABEL(l) ADD ## l
#include "insns/add.inc"
#undef DEFLABEL
#undef USELABEL
}
NEXT_INSN_INCPC();
```

(a) Implementation of VM instructions.

```
DEFLABEL(HEAD):
switch(PTAG(v2)){
case FIXNUM: {
  // addition of fixnum operands
  cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
  dst = cint_to_number(s);
} break;
caes GENERIC: {
  switch(HTAG(v2)){
  case OBJECT: {
    // addition of finum operand and object-type operand
    v2 = to_number(context, v2);
    goto USELABEL(HEAD);
  } break;
...
}
```

(b) Body of a superinstruction (`addfixreg.inc`).

Fig. 8 Specialized dispatch implementation strategy.

bels defined in `insns/addfixreg.inc` are not used. Also note that superinstructions and their original ALR instruction are generated in the same scope because they share the interpreter's local variables.

4.5 Improved Specialized Dispatch

In the improved specialized dispatch implementation strategy, type-based dispatch is specialized to the datatypes of constant operands, similarly to its unimproved version. In the improved version, a superinstruction jumps to the appropriate operation code of its original ALR instruction after a type-based dispatch. **Figure 9** shows the bodies of the `add` and `addfixreg` instructions in this implementation strategy. Each fragment of operation code in `add.inc` has a label like `TLadd_fixnum_fixnum`.

```
switch (PTAG(v1)) {
case FIXNUM:
  switch (PTAG(v2)) {
  case FIXNUM:
  TLadd_fixnum_fixnum:
    cint s = fixnum_to_cint(v1) + fixnum_to_cint(v2);
    regbase[r0] = cint_to_number(s);
    break;
 ...
 }
...
}
```

(a) Body of original ALR instruction (`add.inc`).

```
switch (PTAG(v2)) {
case FIXNUM:
  goto TLadd_fixnum_fixnum;
...
}
```

(b) Body of superinstruction (`addfixreg.inc`).

Fig. 9 Improved specialized dispatch implementation strategy.

`addregfix.inc` jumps to the appropriate label among them after the type-based dispatch.

5. Compiler

We modify the eJS compiler so that it uses superinstructions implemented in the VM. The compiler originally had a constant propagation optimizing pass. This pass replaces an ALR instruction with a constant loading instruction when both source operands are determined to be constants. However, it replaces the ALR instruction only when both operands are constants because the normal eJSVM does not have ALR instructions one of whose operand is a constant.

We extend this constant propagation pass in a way that it replaces an ALR instruction with a superinstruction if one of its operands comes from a single constant loading instruction, and if the appropriate superinstruction is available in the VM. Even if the pass successfully replaces an ALR instruction, it leaves the constant loading instruction. This is because the constant can be used by other instructions as well. If no other instructions use the constant, the constant loading instruction will be eliminated as a redundant instruction in a following optimization pass.

Figure 10 (a) shows an example. This program is compiled to the code shown in Fig. 10 (b), where the variable `ppfix` is put in a register. The first operand of each instruction is its destination register. If the VM only has the superinstruction `addregstr`, whose right operand is a string constant, the program is compiled to the code shown in Fig. 10 (c) using this superinstruction.

A superinstruction may be executed quicker because its type-based dispatch is specialized to the datatype of its constant operand in some implementation strategies. Therefore, the compiler uses superinstructions aggressively even if the constant loading instructions cannot be eliminated.

6. Evaluation

We implemented all the implementation strategies described in Section 4 in eJSTK and evaluated them. We used a subset of the

```
function f(msg) {
  var ppfix = "=";
  return ppfix + msg + ppfix;
}
```

(a) JavaScript program.

```
string r1 "="
add r2 r1 r2
add r1 r2 r1
```

(b) Compiled code without superinstructions.

```
string r1 "="
add r2 r1 r2
addregstr r1 r2 "="
```

(c) Compiled code with a superinstruction.

Fig. 10 Compilation using superinstructions.

```
function triple(a, b, c) {
  return "(" + a + "," + b + "," + c + ")";
}
function main() {
  for (i = 0; i < 500000; i++)
    triple("a", "b", "c");
}
main();
```

Fig. 11 triple benchmark.

SunSpider benchmark suite [*3] and the triple benchmark shown in **Fig. 11** as a representative program using a superinstruction with a string constant operand. The SunSpider benchmarks were modified in the same way as in the article [2]: the number of iterations were adjusted to be suitable for the evaluation of eJSVM, for example.

We generated VMs for each benchmark program so that the VM had the superinstructions listed in **Table 2**. We selected the superinstructions in the following way. First, we counted the number of executions of each instruction in an execution of the benchmark program on the eJSVM that has all the possible superinstructions. Then, we picked those that are executed in more than 1% of the total execution count. For the improved dispatch code sharing implementation strategy, when we added both left-constant and right-constant superinstructions made from the same ALR instruction, we made eJSVM so that more frequently executed instructions could skip the type-based dispatch.

Although users collect profiling information in a normal use case not only for superinstructions but also for limiting operand datatypes of instructions, one may interested in the performance of unspecialized VM to a particular application in the sense of selection of superinstructions. Thus, in addition to these specialized VMs with a dedicated set of superinstructions for each benchmark program (indicated by "specialized" in the figures), we measured two unspecialized VMs that had commonly used superinstructions: "generic 1" and "generic 2". The "generic 1" VM had all superinstructions listed in Table 2. The "generic 2" VM had those five superinstructions that appear twice or more in Table 2, as indicated by the dagger marks (†). We generated these unspecialized VMs in every implementation strategy described in Section 4.

We measured the execution times on Raspberry Pi 3. The configuration is listed below.

Compiler: gcc 8.2.0 (arm-linux-gnueabihf)
Optimization: -Os -marm
OS: Raspbian GNU/Linux 9

We used two operand specifications: any and fixnum. With the any operand specification, all the VM instructions accept any datatype of operands. fixnum is a normal use case of eJS. With fixnum, the add instruction accepts operands of both fixnums or both strings, and the other ALR instructions accept only fixnum operands. The VMs generated with fixnum could not execute some of the benchmarks, because these VMs were specialized to certain applications in the sense of the operand datatype restriction.

[*3] https://webkit.org/perf/sunspider/sunspider.html

Table 2 Implemented superinstructions.

program	superinstruction	execution ratio (%)
3d-cube	addregfix†	5.35
	lessthanregfix†	1.05
access-binary-trees	subregfix†	2.44
	lessthanfixreg	1.66
	mulfixreg	1.63
access-fannkuch	addregfix	6.68
	rightshiftregfix†	1.01
access-nbody	addregfix	1.84
access-nsieve	addregfix	8.10
bitops-3bit-bits-in-byte	bitandregfix	9.99
	bitandfixreg	9.99
	leftshiftregfix†	6.66
	lessthanregfix	3.36
	addregfix	3.34
	rightshiftfixreg	3.33
	rightshiftregfix	3.33
bitops-bits-in-byte	lessthanregfix	10.42
	leftshiftregfix	8.33
	addregfix	5.21
bitops-bitwise-and-func	addregfix	9.09
controlflow-recursive	subregfix	6.38
	lessthanregfix	3.23
	equalregfix	2.34
math-cordic	addregfix	3.90
	lessthanregfix	3.90
	mulregfix	1.20
math-spectral-norm	addregfix	10.28
	divregfix	3.39
	divfixreg	3.39
string-fasta	addregfix	1.11
triple	addregstr	10.71
	addstrreg	3.57
	addregfix	3.57

6.1 Effectiveness of Superinstruction

Figure 12 shows the execution times of the benchmark programs on each VM. The results are normalized to the counterpart on the VM without superinstructions. The suffixes of the benchmark names, any and fixnum, indicate the operand specifications.

Figure 13 shows the sizes of the interpreters, where dashed lines indicate the size of the interpreter of the VM without superinstructions. The legend keys from "S1" to "S5" correspond to those in Table 1. The sizes of the built-in functions are not counted as part of the interpreter sizes, because we expected that only built-in functions that are actually used would be implemented in applications of memory constraint embedded systems. The size of the libc library was not counted either, because although eJSVM depends on some standard library functions, such

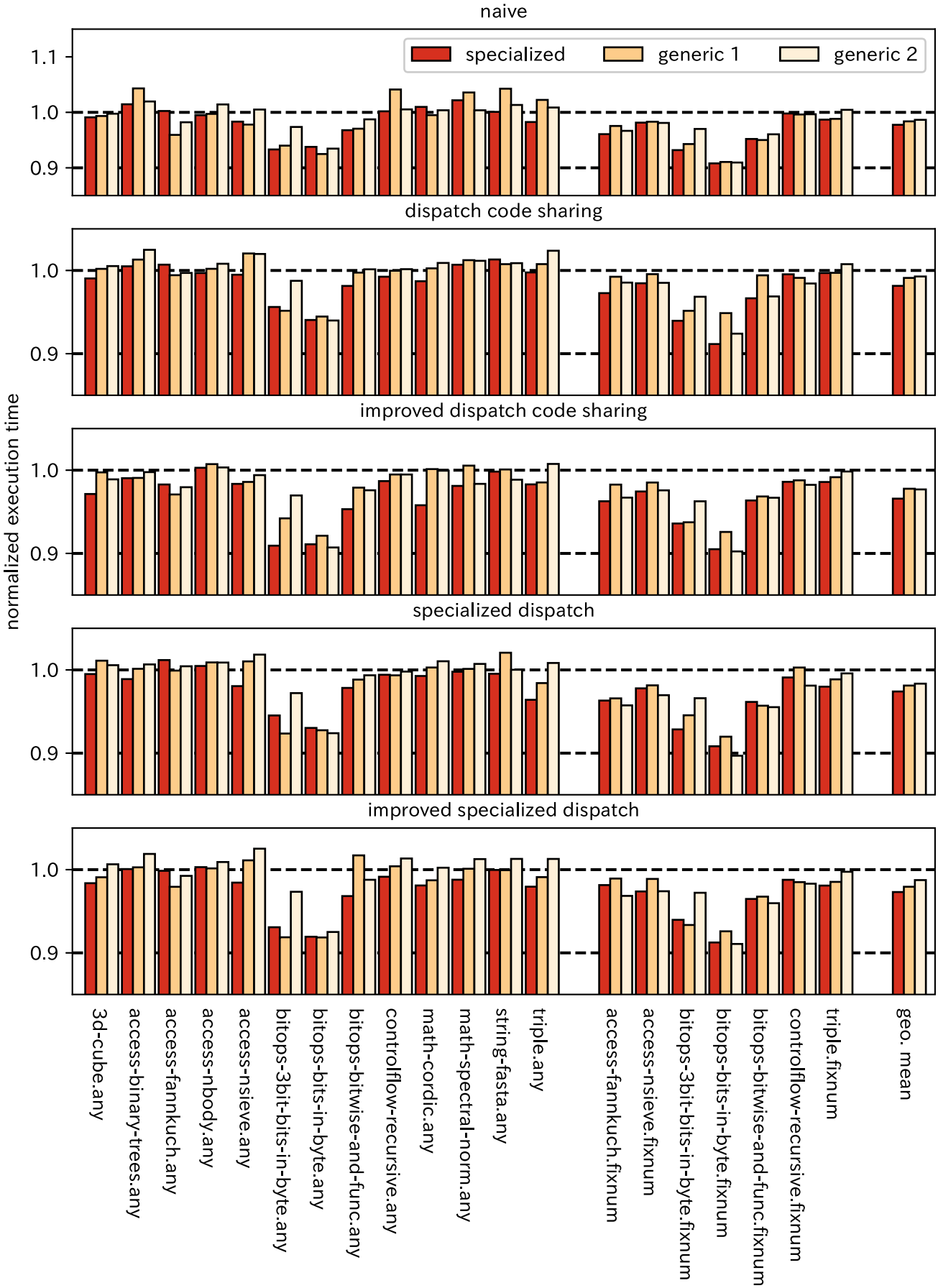

Fig. 12 Normalized execution times.

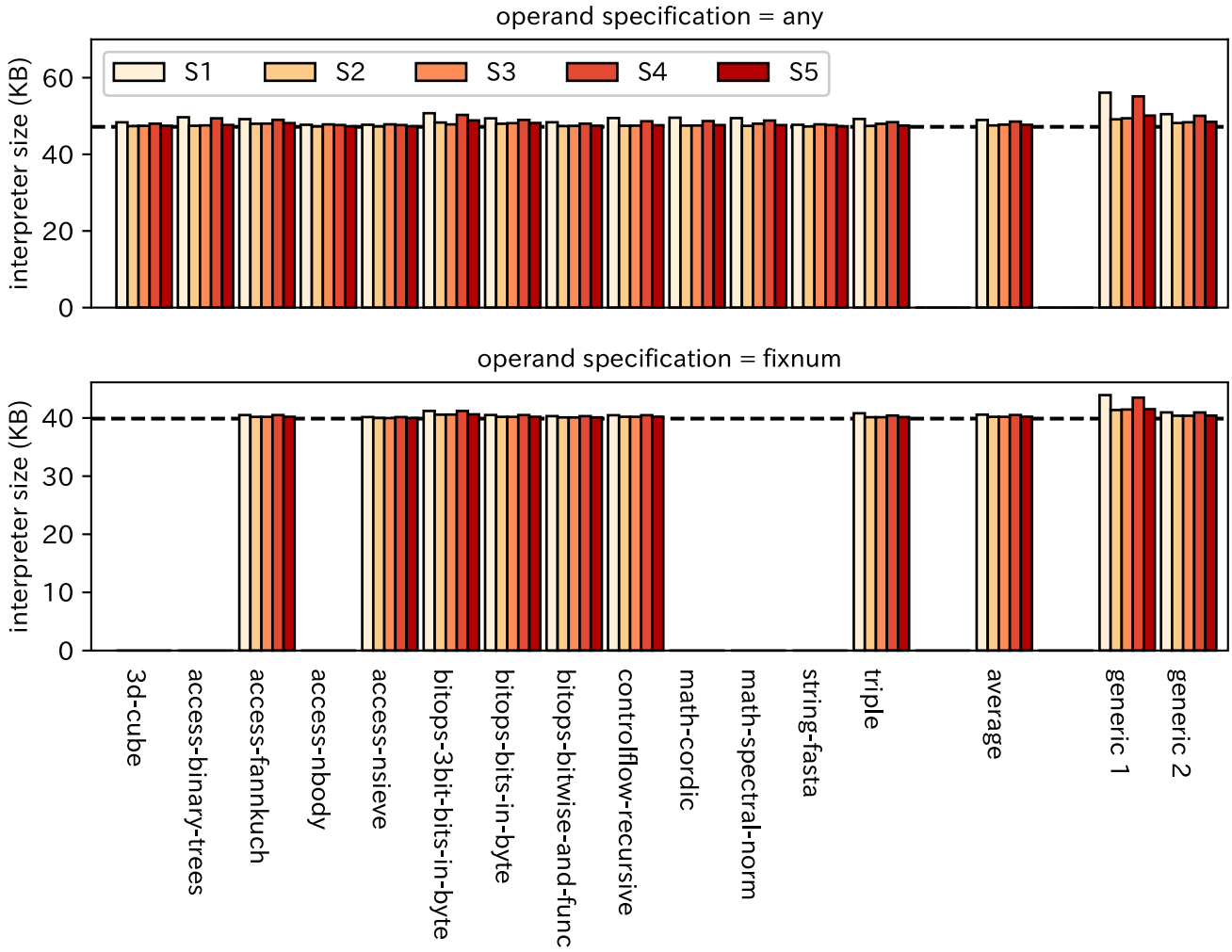

Fig. 13 Interpreter sizes. Dashed lines indicate the sizes of the interpreters without superinstructions.

as string parsing functions for numbers, we plan to implement such functions with our own short code in the future.

6.1.1 Execution Time

For the VMs with dedicated sets of superinstructions generated using the any operand specification, the results fell into two categories. For bitops-3bit-bits-in-byte and bitops-bits-in-byte, the execution times were substantially reduced by superinstructions. For example, for improved dispatch code sharing, they were reduced by around 10%. These benchmarks are those that execute superinstructions rather frequently (see Table 2). For the other benchmarks, there were few differences even when the execution times were reduced; some of the benchmarks became slightly slower when certain implementation strategies were used. The results for fixnum operand specification showed a similar tendency, but with a greater reduction in execution time. These VMs are specialized in the sense of the operand datatype restriction.

6.1.2 Size of Interpreter

The sizes of the interpreters increased by adding superinstructions. The increases differed depending on the number of superinstructions added and the implementation strategy. For bitops-3bit-bits-in-byte, we added the most atasuperinstructions among the VMs with dedicated sets of superinstructions. Depending on the implementation strategy, the interpreter became between 1 KB and 4 KB larger when the any operand specification was used.

6.1.3 Merits of Dedicated Set of Superinstructions

The VMs with dedicated sets of superinstructions outperformed those with commonly used superinstructions (“generic 1” and “generic 2”) in terms of execution time and interpreter size, though the generic VMs also reduced their execution time (Fig. 12). On the other hand, users have to make special effort to use a VM with a set of superinstructions dedicated to their application. This effort includes collecting profiling information from the application in order to select the superinstructions and generating the VM. However, users would still have to do this in a normal use case of eJSTK, even if they do not use superinstructions, as we have described.

6.2 Comparison of Implementation Strategies

We compared implementation strategies in detail. We added seven frequently executed superinstructions in bitops-3-bit-bits-in-byte to the VM one by one from the most frequent one to the least. **Figure 14** plots the execution times against interpreter size. The data point closest to the upper-left corner on each curve represents the result of the VM with one superinstruction, and the number of superinstructions increases along the line. The execution times are normalized to the counterpart on the VM without superinstructions and the sizes are the increases relative to the

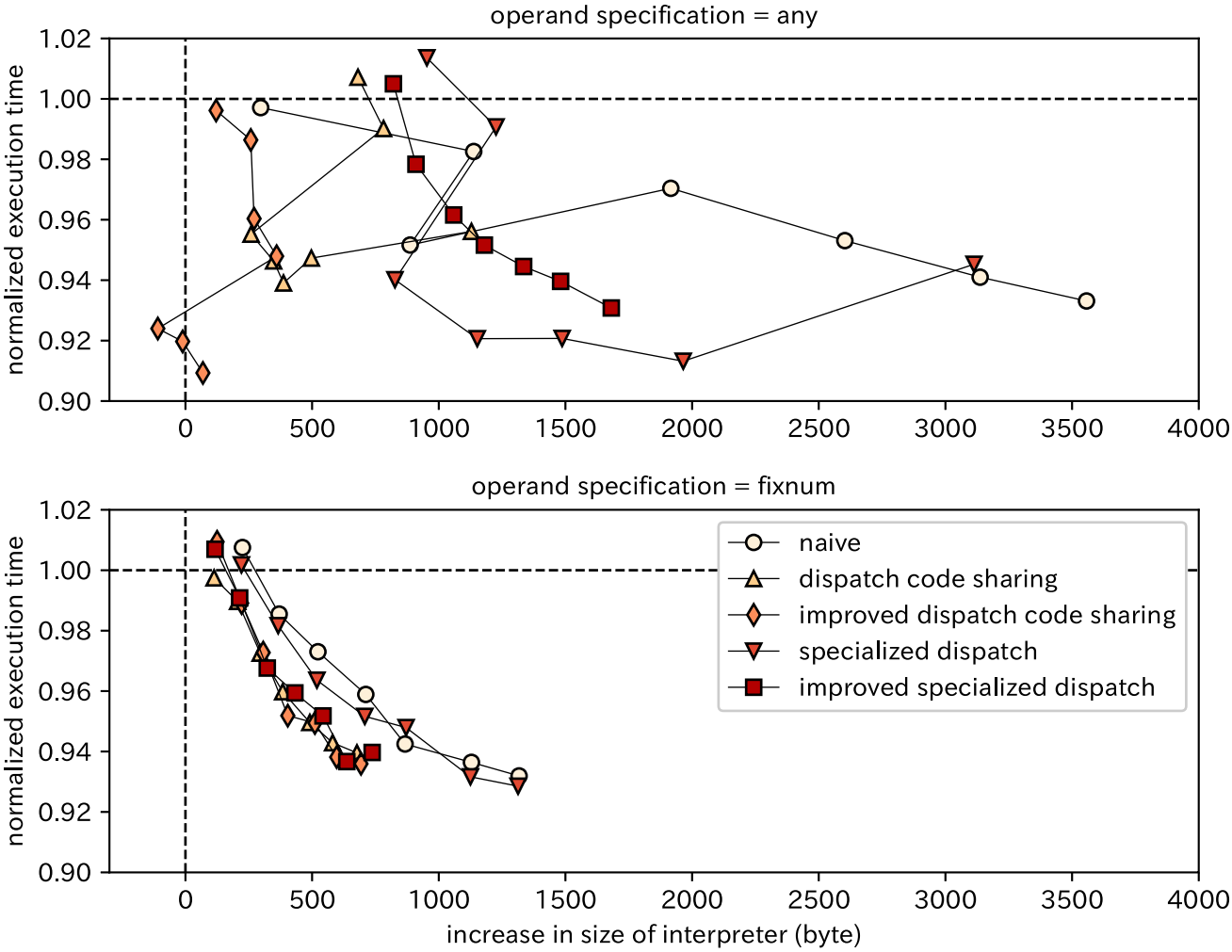

Fig. 14 Execution times and interpreter sizes for bitops-3bit-bits-in-byte.

VM without superinstructions.

Although the execution times were usually reduced and the interpreters usually became larger as more superinstructions were added in every implementation strategy, the results substantially differed from one strategy to another. The curves were sometimes not monotonic; we think the changes to the code affected the optimization strategy of the gcc compiler used to compile the VMs.

6.2.1 Effect of Code Sharing in Limiting VM Bloat

The sizes of the interpreters generated from the any operand specification increased substantially for the naive and specialized dispatch implementation strategy. This is because they duplicated type-based dispatching and operation code. Note that, for the specialized dispatch implementation strategy, the increase was milder than the one for the naive implementation strategy because the type-based dispatch of superinstructions was specialized. For the improved specialized dispatch implementation strategy, the increase was even milder because the operation code was shared. For the dispatch code sharing implementation strategy and its improved version, the sizes increased the least, although the sizes sometimes did not increase monotonically against the number of superinstructions.

With the fixnum operand specification, the type-based dispatch of the original ALR instructions were sufficiently optimized by leveraging the restriction on the operand datatypes. Thus, the sizes of the interpreters in the naive and specialized dispatch implementation strategies were similar. Similarly, the sizes of the interpreters were similar for the other strategies. In other words, the results fell into two groups: the group of strategies that shared the operation code and those that did not.

6.2.2 Effect of Specialization of Type-based Dispatch

When we used the any operand specification, the VMs generated in some of the implementation strategies that somehow reduced the cost of type-based dispatch (the specialized dispatch, improved specialized dispatch, and improved dispatch code sharing implementation strategies) were faster than the others. In particular, the specialized dispatch and improved dispatch code sharing implementation strategies generated the fastest VMs, with the exception of the data point at which seven superinstructions were added to the specialized dispatch implementation strategy.

Regarding the fixnum implementation strategy, the VMs in the group that shared the operation code were faster than the others, regardless of whether the type-based dispatch was specialized or not. This is because the type-based dispatch had been sufficiently specialized by leveraging the information in the operand specifications.

6.2.3 Effect of Reducing Instruction Dispatch

The executions become faster even in the naive and dispatch code sharing implementation strategies. This is because the num-

ber of instruction dispatches and number of accesses to registers in memory were reduced.

6.2.4 Other Performance Factors

When we used the fixnum operand specification, the VMs in the implementation strategies that shared operation code were slightly slower than the others. This difference depends on whether the `goto` statements jumping from the superinstructions to the operation code shared with the original ALR instructions were executed or not.

7. Related Work

Adding superinstructions is a well-known technique to improve VM performance. Proebsting [3] added superinstructions to a C language interpreter to improve its performance. Proebsting generated an interpreter from the specifications of the primitive instructions and a list of instructions to be combined. Ertl et al. [6] developed Vmgen, a system to generate a VM with superinstructions from the templates of primitive operations and a specification of superinstructions. Vmgen was used to develop Java VMs for embedded systems [7], [8]. We also added superinstructions. But our target was eJSVM, which posed a new challenge. Proebsting's interpreter and the VMs generated by Vmgen assume that intermediate code is type-monomorphic. Thus, a VM generator does not have a chance to optimize the type-based dispatching. In contrast, eJSTK generates a type-based dispatching VM. We proposed optimizations of type-based dispatching that leverage the datatypes of constant operands.

For VMs running on desktop computers, multiple instructions are often combined dynamically. Piumarta et al. [9] combined instructions dynamically and improved the performance of an interpreter with a RISC-style instruction set. In approaches such as JIT compilation, a VM has a machine code optimization mechanism because combining VM instructions creates extra room for optimization of the machine code implementing the combined instruction. In contrast, we optimized ahead of time because we target embedded systems. We focused on optimizations to reduce the cost of type-based dispatch, which is another difference from previous work. Regarding optimizations for specific languages, we note that Zakirov et al. [10] proposed superinstructions for the Ruby VM to eliminate boxing of floating-point numbers.

Our superinstructions are ALR instructions taking constant operands created from constant loading and primitive ALR instructions. However, there is another option: to develop a single VM with a complex instruction set computer (CISC) style instruction set including ALR instructions taking constant operands. JerryScript is a JavaScript VM with such an instruction set. The body of the interpreter loop of JerryScript comprises three steps. The first step decodes an instruction and loads operands. The operand values are stored in the interpreter's variables. The second step is instruction dispatch and computation of the instruction. The third step stores the result to the appropriate location. The first and third step may access various locations such as slots on the stack and registers. Although these steps are implemented with a substantial amount of complicated code, they are shared among all instructions. Although we prototyped this approach for eJS, it degraded its performance.

In this research, we minimized VM bloat by implementing superinstructions so that they can share the type-based dispatch and operation code. Peng et al. [11] proposed a code sharing technique to avoid VM bloat due to duplication of instruction code introduced by stack caching in the context of stack-based interpreters.

8. Conclusion

We introduced superinstructions, which are combinations of constant loading and ALR instructions, to eJS, a JavaScript system for embedded systems. We compared five implementation strategies carrying out different optimization in the sense of code sharing and specialization of type-based dispatch leveraging datatypes of constant operands. Although all of the implementation strategies we tested reduced the execution time, the degree of improvement and increases in the size of the interpreters differed from one strategy to another.

Acknowledgments The authors would like to thank all members involved in the eJS project. They would also like to thank the reviewer for valuable comments.

This work was supported by the JSPS KAKENHI Grant Number 16K00103.

References

[1] Kataoka, T., Ugawa, T. and Iwasaki, H.: A Framework for Constructing JavaScript Virtual Machines with Customized Datatype Representations, *Proc. SAC 2018*, pp.1238–1247, ACM (2018).
[2] Ugawa, T., Iwasaki, H. and Kataoka, T.: eJSTK: Building JavaScript virtual machines with customized datatypes for embedded systems, *Journal of Computer Languages*, Vol.51, pp.261–279 (2019).
[3] Proebsting, T.A.: Optimizing an ANSI C Interpreter with Superoperators, *Proc. POPL 1995*, pp.322–332, ACM (1995).
[4] ECMA International: *Standard ECMA-262 - ECMAScript Language Specification*, 5.1 edition (2011).
[5] Bell, J.R.: Threaded Code, *Comm. ACM*, Vol.16, No.6, pp.370–372 (1973).
[6] Ertl, M.A., Gregg, D., Krall, A. and Paysan, B.: Vmgen - A generator of efficient virtual machine interpreters, *Software: Practice and Experience*, Vol.32, No.3, pp.265–294 (2002).
[7] Beatty, A., Casey, K., Gregg, D. and Nisbet, A.: An Optimized Java Interpreter for Connected Devices and Embedded Systems, *Proc. SAC 2003*, pp.692–697, ACM (2003).
[8] Ertl, M.A., Thalinger, C. and Krall, A.: Superinstructions and replication in the Cacao JVM interpreter, *Journal of .NET Technologies*, Vol.4, pp.25–32 (2006).
[9] Piumarta, I., Riccardi, F. and Rocquencourt, I.: Optimizing Direct Threaded Code By Selective Inlining, *Proc. PLDI 1998*, pp.291–300, ACM (1998).
[10] Zakirov, S., Chiba, S. and Shibayama, E.: How to Select Superinstructions for Ruby, *IPSJ Online Transactions*, Vol.3, pp.54–61 (online), DOI: 10.2197/ipsjtrans.3.54 (2010).
[11] Peng, J., Wu, G. and Lueh, G.-Y.: Code Sharing among States for Stack-Caching Interpreter, *Proc. IVME 2004*, pp.15–22, ACM (2004).

Tomoya Nonaka was born in 1996. He received his B.E. degree from Kochi University of Technology in 2019. He is a member of IPSJ.

Tomoharu Ugawa received his B.Eng. degree in 2000, M.Inf. degree in 2002, and Dr.Inf. degree in 2005, all from Kyoto University. He worked for a research project on real-time Java at Kyoto University from 2005 to 2008. In 2008–2014, he was an assistant professor at the University of Electro-Communications. He is currently an associate professor at Kochi University of Technology. His work is in the area of implementation of programming languages with a specific focus on memory management. He received the IPSJ Yamashita SIG Research Award in 2012.