IPSJ Transactions on Programming Vol. 3 No. 2 1–8 (Mar. 2010)

Regular Paper

How to Select Superinstructions for Ruby

Salikh Zakirov,[†1] Shigeru Chiba[†1]
and Etsuya Shibayama[†2]

Superinstruction is well-known techniques of improving performance of interpreters. Superinstructions eliminate jumps between VM operations (interpreter dispatch) and enable more optimizations in merged code. In past, processors with simple BTB-based branch predictors had high misprediction rate when executing interpreted code, resulting in high overhead of interpreter dispatch, so superinstructions were used to reduce it. However, this assumption is incorrect for Ruby on current hardware. Accordingly, using superinstructions for eliminating jump instructions only marginally improves performance. In this paper, we consider applying superinstructions differently to improve performance of floating point computation. We note that high percentage of objects allocated during numeric computation are boxed floating point values, meanwhile garbage collection takes significant part of the execution time. Using superinstructions composed from pairs of arithmetic operations we were able to reduce allocation of boxed floats by up to 36%, and obtain improvement in performance of up to 22%.

1. Introduction

Ruby language is one of the dynamic programming languages that received much attention recently. Dynamic languages in general and Ruby in particular are convenient languages for prototyping software systems. However, performance of Ruby is a subject of much concern. Speed of numeric benchmarks is particularly often quoted, where Ruby version sometimes is 100 or more times slower than implementation in C. Thus, improving Ruby performance is an important problem. Performance of interpreters has been thoroughly studied in the past. Interpreter dispatch has been found to be a major factor contributing to execution time. Many techniques have been proposed to improve interpreter performance. Among others, threaded interpreter and superinstructions are already implemented in Ruby 1.9.

Superinstructions [8] have been proposed in past as a means to reduce the overhead of interpreter dispatch — jumps between pieces of executed code. This comes from the assumption that overhead of interpreter dispatch is high, based on past research of some interpreter systems. However, as results of our measurements show, this assumption is incorrect for Ruby interpreter on current hardware. Misprediction rate is low at 5–17%, and overall overhead of interpreter dispatch is 0.6–3%. So we propose a new way of using superinstructions to improve performance of numeric workloads. According to our analysis, a majority of object allocated in numerical workloads are boxed floating point numbers, and a large part of execution time is spent in allocating and garbage collecting them. That is where superinstructions can help to reduce the number of produced boxed floats.

The contributions of this paper are:

- We explain why prior approach to superinstructions does not produce substantial benefits for Ruby, based on experimental data.
- We propose using superinstructions composed of pairs of arithmetic operations to reduce allocation of boxed floating point numbers and experimentally show speedup of up to 22%.

2. Superinstructions

The Ruby programming language is implemented as a virtual machine interpreter since version 1.9. Interpreter performance was a topic of close attention of much research in the past. Among many approaches to interpreter performance, superinstructions have got much attention. In this paper, we specifically concentrate on studying the effects and benefits of superinstructions.

Superinstructions apply to virtual machine interpreters, which typically work with bytecode representation of the program. Superinstruction is a sequence of two or more VM instructions, which have their implementation merged together. Superinstructions may affect the program performance for multiple reasons:

(1) Jumps between merged instructions are eliminated.
(2) Merged instructions have multiple copies (merged and unmerged) of their

†1 Department of Mathematical and Computing Sciences, Tokyo Institute of Technology
†2 Information Technology Center, The University of Tokyo

© 2010 Information Processing Society of Japan

Table 1 Branch misprediction in baseline version.

benchmark	mandelbrot	nbody	partial_sums	spectral_norm
total cycles	$600 \cdot 10^8$	$790 \cdot 10^8$	$730 \cdot 10^8$	$690 \cdot 10^8$
br. misp. stall cycles	$16 \cdot 10^8$	$13 \cdot 10^8$	$10 \cdot 10^8$	$4 \cdot 10^8$
hw. instructions	$1{,}019 \cdot 10^8$	$1{,}287 \cdot 10^8$	$1{,}097 \cdot 10^8$	$1{,}161 \cdot 10^8$
VM ops	$9 \cdot 10^8$	$10 \cdot 10^8$	$8 \cdot 10^8$	$18 \cdot 10^8$
indirect branches	$17 \cdot 10^8$	$21 \cdot 10^8$	$18 \cdot 10^8$	$23 \cdot 10^8$
mispredicted ind. br.	$2.9 \cdot 10^8$	$3.5 \cdot 10^8$	$2.8 \cdot 10^8$	$1.2 \cdot 10^8$
misprediction rate	17.0%	16.4%	15.5%	5.1%
misprediction overhead	2.6%	1.7%	1.4%	0.6%

implementation.

(3) Merging implementation of several instructions into single non-branching block of code provides more opportunities for optimization compared to separately compiled fragments of code.

Interpreter dispatch, that is, branching between implementation of VM operations, was found to be major contributor to interpreter execution time due to indirect branch misprediction. The indirect branch predictor widely used in the past hardware consists of the *branch target buffer* (BTB), which is a cache of indirect branch targets, keyed with branch site address. Such a structure has a limit of one prediction per branch site, while in interpreters it is common for an indirect branch site to have multiple targets. As Ertl, et al. [1] describe, branch misprediction overhead can be as high as half of total execution time on a processor with simple BTB.

Since the overhead of interpreter dispatch has been found high, it is natural that most of the prior superinstruction research efforts were concentrated on studying and exploiting points (1) and (2). However, Ruby interpreter on modern hardware shows different performance characteristics.

The processor we used in experiments, Intel Core 2 Duo E8500, has an enhanced branch predictor, and is quite good at predicting indirect branches due to threaded interpreter dispatch (see **Table 1**). Scarce information is available on details of branch predictor that is used in current Intel processors, however, it is hinted, that it uses two-level scheme with branch history, and observed branch misprediction rates fully support that conjecture.

Moreover, interpreter dispatch does not constitute large proportion of execution time of Ruby on numeric benchmarks. Our experiments with straightforward static superinstructions implemented in Ruby 1.9 [10] (referred to as "naive superinstructions") showed limited benefit in performance of about 4% (see Section 4). Selection of superinstructions is based on simple heuristics, based on frequency of occurrence of instruction pairs in benchmark execution trace.

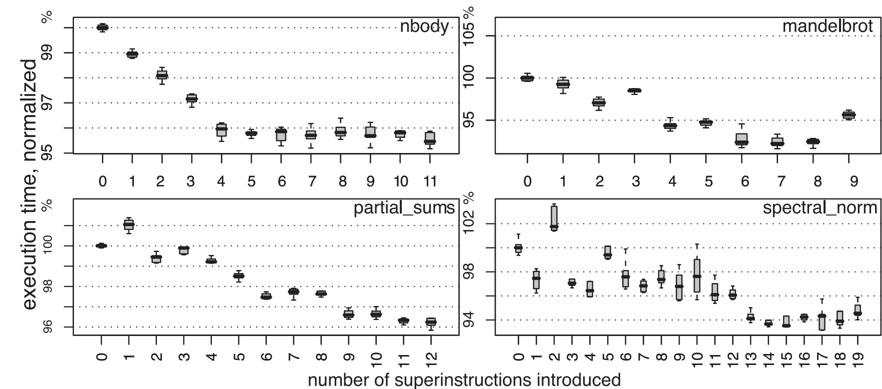

Fig. 1 Effects of introducing naive superinstructions one by one.

Figure 1 shows graphs of execution time for the naive superinstructions, introduced one by one. "Naive" denotes that no effort is made to optimize the merged superinstruction beyond what C compiler can do. Numbers on the x axis denote the number of superinstructions introduced. The instructions to merge into superinstructions are chosen according to occurrence frequency in the execution trace of the very same benchmark, so that the most frequently occurring combination of 2 instructions is introduced in version "1", the second most frequent combination is added to version "2" and so on.

While general trend matches expectation of slightly improving performance as more superinstructions are introduced, the graphs are not strictly monotone, which allows us to observe, that some factors at play have more influence than mere number of indirect branches. As **Fig. 2** shows, the characteristic that is closely related to the effect on performance is number of indirect branch mispredictions.

Particularly noticeable change in performance occurs when superinstruction is

© 2010 Information Processing Society of Japan

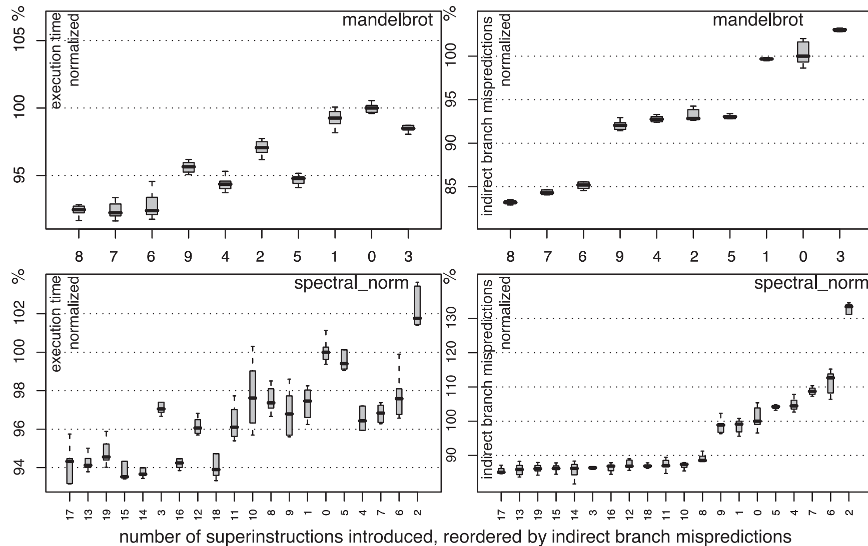

Fig. 2 Comparison of execution time with indirect branch mispredictions.

always followed by the same instruction during execution, which means that indirect branch instruction that transfers control to next operation becomes single-target. In our experiments, hardware branch predictor did an excellent job of flawlessly predicting the target of single-target indirect branches, resulting in a small but visible change in overall performance.

Introducing superinstructions may both improve or worsen branch misprediction rate. For example, because three consecutive VM operations can be merged into dual-operation superinstruction and remaining single instruction in two ways, the positive effect described above can happen or not depending on which two instructions get merged. Our experiments also showed, that even seemingly harmless changes like reordering operations in the source code of interpreter can have visible effects on branch misprediction rate and thus on performance.

As a result, the effects of introducing superinstructions cannot be predicted precisely without profiling, and so there is little hope of getting best possible performance out of superinstructions using VM operation frequency profile alone.

3. Boxing Optimization in Superinstructions

We propose to use superinstructions in Ruby to reduce GC overhead due to boxed floating point numbers. This is reasonable, as profiling of numeric benchmarks shows that overwhelming majority of allocated objects are floating-point numbers, and garbage collection has significant share in execution time (see Section 4.4).

3.1 Implementation

The boxing overhead comes from Ruby bytecode structure. In Ruby, everything is an object. Operations like addition and subtraction are in fact method calls. For example, the assignment `x = x * 2.0` is equivalent to `x = x.send(:*,2.0)` and produces the following bytecode:

```
0000 getdynamic x, 0 ( 1)
0003 putobject 2.0
0005 send :*, 1, nil, 0, <ic>
0011 dup
0012 setdynamic x, 0
```

where `:*` is notation for the symbol of multiplication and `dup` bytecode is needed because every statement in Ruby is also an expression, and have to leave a return value on the stack.

To reduce overhead due to method call, calls generated from common arithmetic operation are rewritten as separate bytecodes:

```
0000 getdynamic x, 0 ( 1)
0003 putobject 2.0
0005 opt_mult
0006 dup
0007 setdynamic x, 0
```

The bytecodes for arithmetic operations must accept arguments of any type. In order to discern between values of different types, integers are implemented with tags, and floating point number are implemented with boxing.

Ruby has five instructions that deal with floating point numbers: `opt_plus`,

© 2010 Information Processing Society of Japan

Listing 1 Source code of `opt_plus` implementation

```
1  DEFINE_INSN opt_plus
2  ()    /* immediate parameters */
3  (VALUE a, VALUE b)  /* stack inputs */
4  (VALUE val)   /* stack output */
5  {
6      /* ... */
7      if (HEAP_CLASS_OF(a) == rb_cFloat && /* check types of arguments */
8          HEAP_CLASS_OF(b) == rb_cFloat &&
9          BASIC_OP_UNREDEFINED_P(BOP_PLUS)) { /* and validity of optimization */
10             val = DBL2NUM(RFLOAT_VALUE(a) + RFLOAT_VALUE(b));
11     } else {
12         PUSH(a);
13         PUSH(b);
14         CALL_SIMPLE_METHOD(1, idPLUS, a);
15     }
16 }
```

`opt_minus`, `opt_mult`, `opt_div`, and `opt_mod`. Listing 1 shows somewhat simplified source code of `opt_plus` instruction (code to deal with tagged implementation of `fixint` is omitted). In line 10 macro `DBL2NUM` allocates new boxed floating point object for storing results of arithmetic operation.

Superinstructions make it possible to reduce the number of boxed floating point numbers. For example, in the superinstruction, which resulted from merging `opt_mult` with `opt_plus`, the following code is used:

```
val = DBL2NUM(RFLOAT_VALUE(a) + RFLOAT_VALUE(b) * RFLOAT_VALUE(c));
```

In this way, superinstruction allocates only the final result of the two operations, while the regular instruction allocate two numbers: the intermediate result of multiplication, and the final result.

In our implementation, we implemented all 25 combinations of 5 arithmetic instructions. These superinstructions are referred to as *"opt-opt"* superinstructions throughout this paper. Since C compiler is not capable of optimizing out excessive allocation, we chose to manually implement superinstructions.

3.2 Limitations

The approach of using superinstructions for reducing garbage collector overhead has some grave limitations. First of all, superinstructions can only be used when arithmetic operations strictly follow one another. Intermission of other instructions, such as stores to local variables, duplication, or loads restrict the applicability of superinstructions approach.

Application of superinstructions to types other than boxed floats has limited effectiveness. With fixed integers, which are implemented as tagged in-line values, there is no boxing overhead in the first place, so superinstructions have little effect. With strings and arbitrary precision integers (Bignum), boxed form is essentially the only form of existence of objects, so unboxed representation is impossible, and handling takes more time than that with floating point numbers, so potential benefit of reduced allocation is much lower.

Superinstructions longer than 2 instructions do not seem practical if implemented statically, as adding many superinstructions will noticeably increase code size, while probability of a long superinstruction being applicable in a benchmark is quite low. For this reason in this work we did not consider superinstructions of length more than 2.

4. Experiments

4.1 Choice of the Benchmarks

Since the goal of this research is to optimize handling of floating point numbers, the following numeric benchmarks from Ruby Benchmark Suite are used: `mandelbrot`, `nbody`, `partial_sums`, and `spectral_norm`.

4.2 Methodology

For performance evaluation we used repeated measurements of wall clock execution time of the benchmarks. The median of 31 individual measurements is taken. `Trace` instruction is disabled in reported data to expose more opportunities for opt-opt superinstructions, though it had little influence in our experience. The machine we used for experiments is Intel Core 2 Duo E8500 3.16 GHz with 3 Gb of memory and Gentoo Linux operating system. Ruby source code was compiled using gcc compiler version 4.1.2, which produced 8087 instructions for floating-point operations.

© 2010 Information Processing Society of Japan

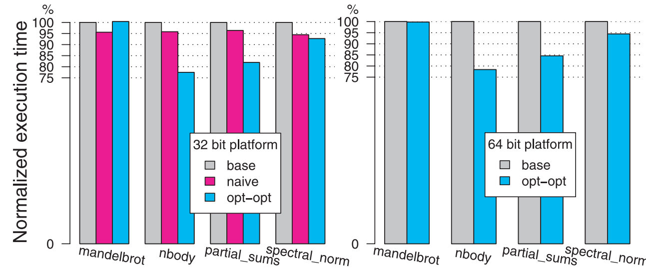

Fig. 3 Execution time of benchmarks, 32 bit mode (left) and 64 bit mode (right). The lower, the better.

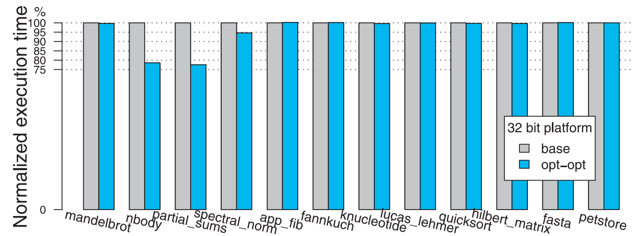

Fig. 4 Execution time of other benchmarks, 32 bit mode.

4.3 Results

Figure 3 shows performance measurements. The left column in each group is normalized to 100% baseline measurement. The right column shows execution time of our implementation.

In three cases out of four, the proposed optimization shows comparable or better results than naive superinstructions. Note, that naive superinstructions were chosen specifically for each benchmark, and opt-opt superinstructions were the same for all benchmarks. As **Fig. 4** shows, using opt-opt superinstructions results in no changes of performance on other benchmarks.

The benchmark `mandelbrot` does not show any improvement, because no superinstructions were applicable. To illustrate, we show the excerpt of hot code from mandelbrot benchmark:

Table 2 Branch misprediction in naive superinstructions version.

benchmark	mandelbrot	nbody	partial_sums	spectral_norm
total cycles	$580 \cdot 10^8$	$770 \cdot 10^8$	$700 \cdot 10^8$	$650 \cdot 10^8$
br. misp. stall cycles	$4.5 \cdot 10^8$	$6.7 \cdot 10^8$	$6.8 \cdot 10^8$	$1.5 \cdot 10^8$
hw. instructions	$1{,}000 \cdot 10^8$	$1{,}300 \cdot 10^8$	$1{,}100 \cdot 10^8$	$1{,}100 \cdot 10^8$
VM ops	$6 \cdot 10^8$	$7 \cdot 10^8$	$5 \cdot 10^8$	$11 \cdot 10^8$
indirect branches	$13 \cdot 10^8$	$18 \cdot 10^8$	$15 \cdot 10^8$	$16 \cdot 10^8$
mispredicted ind. br.	$1.8 \cdot 10^8$	$1.9 \cdot 10^8$	$1.8 \cdot 10^8$	$0.13 \cdot 10^8$
misprediction rate	13.7%	10.7%	11.9%	0.8%
misprediction overhead	0.8%	0.9%	1.0%	0.2%

Table 3 Branch misprediction in opt-opt version.

benchmark	mandelbrot	nbody	partial_sums	spectral_norm
total cycles	$640 \cdot 10^8$	$64 \cdot 10^8$	$64 \cdot 10^8$	$650 \cdot 10^8$
br. misp. stall cycles	$14 \cdot 10^8$	$8.2 \cdot 10^8$	$8.6 \cdot 10^8$	$3 \cdot 10^8$
hw. instructions	$1{,}057 \cdot 10^8$	$997 \cdot 10^8$	$922 \cdot 10^8$	$1{,}114 \cdot 10^8$
VM ops	$9 \cdot 10^8$	$9 \cdot 10^8$	$7 \cdot 10^8$	$16 \cdot 10^8$
indirect branches	$18 \cdot 10^8$	$17 \cdot 10^8$	$16 \cdot 10^8$	$22 \cdot 10^8$
mispredicted ind. br.	$3 \cdot 10^8$	$2.3 \cdot 10^8$	$2.6 \cdot 10^8$	$1.2 \cdot 10^8$
misprediction rate	17.2%	13.3%	17.0%	5.4%
misprediction overhead	2.2%	1.3%	1.3%	0.5%

```
1 tr = zrzr − zizi + cr
```

```
1 getdynamic zrzr, 3
2 getdynamic zizi, 3
3 opt_minus
4 getdynamic cr, 3
5 opt_plus
6 setdynamic tr, 0
```

A simple rewrite as `tr = cr + (zrzr - zizi)` reorders the operations, putting arithmetic operations together. Two small tweaks in `mandelbrot` benchmark can improve performance of opt-opt version by 20%. A better approach to rewriting the application would be to provide superinstructions of the form `op-x-op`, where `op` is an arithmetic operation, and `x` can be access to local variable or load of constant. We believe it will provide similar speed-up.

Table 2 and **Table 3** give some profiling numbers on naive and opt-opt version

© 2010 Information Processing Society of Japan

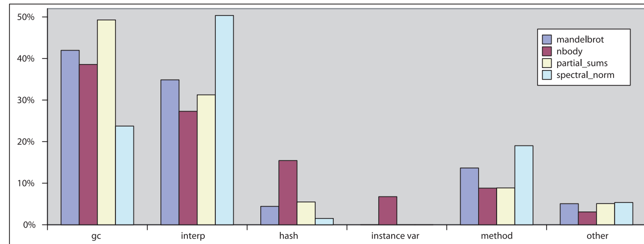

Fig. 5 Sampling profiling.

Table 4 Allocation data.

benchmark	allocated objects	allocated floats	ratio of floats
mandelbrot	204119688	203253333	99.58%
nbody	29013440	29000433	99.96%
partial_sums	87512302	87500027	99.99%
spectral_norm	1216914	1200416	98.6%

Table 5 Applicability of opt-opt superinstructions in benchmarks.

	Instructions in the inner loop				Reduction
benchmark	total	arithmetic	pairs merged	after merge	in allocation
mandelbrot	60	9	0	9	0%
nbody	103	26	9	17	36%
partial_sums	124	29	9	20	26%
spectral_norm	34	10	3	7	32%

for comparison with baseline version.

4.4 Profiling Data

Profiling data illustrates why benefit from opt-opt superinstructions is possible. Garbage collection takes big share in execution time (**Fig. 5**). Second big contributor to the execution time on Ruby is main interpreter loop (implementation of bytecodes). The other categories shown in the figure are: hash lookups (due to instance variable access, method lookup and direct use of hash maps), time spent in instance variable access (besides hash lookup), and method calls.

Considering the share of time spent in garbage collection and the fact, that overwhelming majority of allocation in numeric benchmarks is due to boxed floating point numbers (see **Table 4**), it is quite reasonable that any reduction in boxed values allocation will produce visible benefit in performance. The last column in Table 4 shows how much the number of boxed floats was reduced by using opt-opt superinstructions.

Table 5 provides data on how opt-opt superinstructions affected an inner computation loop of the benchmarks. First column gives the total number of instructions in the inner computation loop, the second — the number of arithmetic instructions, in the third column number of consecutive arithmetic instruction pairs is shown, and the forth column presents number of arithmetic instruction after application of superinstructions. The last column shows reduction in floating point object allocation resulting from use of superinstructions.

5. Related Work

An alternative approach to reducing overhead of boxing floating point numbers in Ruby has been evaluated by Sasada[9]. It works by stealing a few bits from pointer binary representation, which normally are zero due to pointer alignment, and using non-zero tag to trigger special handling of the remaining bits. Since Ruby uses double precision for its floating point arithmetic, this approach is limited to 64 bit architectures, while our approach provided similar benefits on both 32 bit and 64 bit platforms. Tag bits do not allow to store complete double precision value, so fall-through path for boxed representation is still required. Since range of values represented in-line is chosen to include most commonly occurring values, this approach allows eliminate most of overhead due to boxing of floating point values at the expense of small overhead of checking tag bits. Also, using tagged values incurs small overhead to programs that do not use floating point values at all. Overall, tagging approach resulted in 28–35% improvement in execution time, compared to 0–22% improvement of opt-opt superinstructions.

Kawai[4] studied possibilities of using limited form of garbage collection over stack-allocated heap of floating-point registers and wide stack techniques. Their approach produced 25%–50% improvements in numerically intensive programs, so is a good alternative way to reduce boxing of intermediate floats. Stack allocation of intermediate computation results was earlier discussed by Steele[11].

Owen, et al.[5] proposed lazy boxing optimization in context of compiler for Java-like language with generics over value-types. Lazy boxing works by allo-

© 2010 Information Processing Society of Japan

cating boxed objects in stack frame, and moving the object to heap only when needed, thus reducing the number of heap allocations and associated overhead. This techniques relies on static compiler analysis to detect paths where the stack object may escape scope of its stack frame.

Superinstructions were proposed in past for improving performance of interpreted systems and reducing code size. Proebsting [8] used superinstructions to optimize size and speed of interpreted execution of ANSI C programs. The operations of the virtual machine were chosen to closely match intermediate C compiler representation.

Piumarta, et al. [6] achieved good performance improvements for the low-level interpreter with RISC-like instruction set and somewhat smaller improvements for Objective Caml interpreter. Interpreters they study are much more low-level than Ruby interpreter, studied in this work, and have low average number of native instructions per VM operation.

Hoogerbrugge, et al. [3] built a hybrid compiler-interpreter system, in which time-critical code is compiled, and infrequently executed code interpreted. The system employs dictionary-based compression by means of superinstructions. Instruction set of the interpreter is also based on the native instructions of the processor used.

Recent study by Prokopski, et al. [7] predicted and verified limited effect of code copying techniques (analog of naive superinstructions in our work) on Ruby VM. Our results are in good accordance with theirs.

6. Conclusion

We proposed and evaluated a specific way of constructing superinstructions for Ruby 1.9. Reduction in floating point number boxing shows promising improvements, and suggests that even better improvements are possible with complete elimination of boxing.

References

1) Ertl, M.A. and Gregg, D.: The Structure and Performance of Efficient Interpreters, *Journal of Instruction-Level Parallelism*, Vol.5, pp.1–25 (2003).
2) Ertl, M.A. and Gregg, D.: Optimizing indirect branch prediction accuracy in virtual machine interpreters, *SIGPLAN '03 Conference on Programming Language Design and Implementation*, pp.278–288 (2003).
3) Hoogerbrugge, J., Augusteijn, L., Trum, J. and van de Wiel, R.: A code compression system based on pipelined interpreters, *Softw. Pract. Exper.*, Vol.29, No.11, pp.1005–1023 (1999).
4) Kawai, S.: Efficient floating-point number handling for dynamically typed scripting languages, *Proc. Dynamic Languages Symposium* (*DLS'08*) (2008).
5) Owen, T. and Watson, D.: Reducing the cost of object boxing, *Compiler Construction, Lecture Notes in Computer Science*, Vol.2985, pp.202–216 (2004).
6) Piumarta, I. and Riccardi, F.: Optimizing direct threaded code by selective inlining, *SIGPLAN Conference on Programming Language Design and Implementation*, pp.291–300, ACM Press (1998).
7) Prokopski, G. and Verbrugge, C.: Analyzing the performance of code-copying virtual machines, *SIGPLAN conference on object-oriented programming systems and languages* (*OOPSLA'08*), ACM SIGPLAN Notices, Vol.43, Issue 10, pp.403–422 (2008).
8) Proebsting, T.A.: Optimizing an ANSI C interpreter with superoperators, *Proc. Symp. on Principles of Programming Languages*, pp.322–332, ACM Press (1995).
9) Sasada, K.: A lightweight representation of floating-point numbers on ruby interpreter, *Proc. workshop of programming and programming languages* (*PPL2008*) (2008).
10) Sasada, K.: Efficient implementation of Ruby virtual machine, PhD Thesis, The University of Tokyo, Graduate school of information science and technology (2007).
11) Steele, G.L., Jr.: Fast arithmetic in MacLISP, *MIT AI Memo 421* (1977).

(Received September 28, 2009)
(Accepted January 5, 2010)

Salikh Zakirov was born in 1980. He graduated from Moscow State University in 2001, majoring in mathematics, and has been working since in the fields of software engineering and development related to Java Virtual Machine. He entered Ph.D. course at Tokyo Institute of Technology in 2007. His current research interests are the dynamic languages and implementation techniques of virtual machines.

© 2010 Information Processing Society of Japan

Shigeru Chiba received his M.Sc. and Ph.D. in computer science from the University of Tokyo in 1993 and 1996 respectively. He became an assistant professor at University of Tsukuba in 1997 and at Tokyo Institute of Technology in 2001. He is now a professor since 2008. His research interests include aspect-oriented programming, reflection and meta-object protocols.

Etsuya Shibayama received M.Sc. in mathematical sciences from Kyoto University in 1983 and D.Sc. in information science from the University of Tokyo in 1991. He became an associate professor and a professor in Tokyo Institute of Technology in 1993 and 2000, respectively, and a professor in the University of Tokyo in 2008. His recent interests include language-based software security and methodologies for building secure software.

© 2010 Information Processing Society of Japan