A Dissertation Submitted to Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of Doctor of Science in Mathematical and Computing Sciences

A study of superinstructions and dynamic mixin optimizations

(融合命令およびダイナミック・ミックスインの最適化)

Salikh ZAKIROV

Dissertation Chair:
Shigeru CHIBA

February 2011,

Abstract

This thesis considers the performance of the dynamic language implementation in the context when dynamic system updates such as dynamic mixin installation or removal are performed repeatedly and with high frequency. High dynamicity has not been widely studied, and many existing high-performance implementation techniques behave badly under highly dynamic load. Dynamic languages are an important step on the never ending quest of improving productivity of programmers, while the complexity of the computing systems increases together with technological progress. Dynamic features contribute to improving productivity by enabling easy-to-use interfaces and metaprogramming. Performance of dynamic language implementation still remains an issue.

We propose two optimization techniques for the implementation of dynamic languages. Merging arithmetic instructions can be used in the polymorphic interpreter to reduce the allocation rate of boxed floating-point numbers, but has no adverse impact with regard to dynamic features. Superinstructions has been used before to reduce the overhead of interpreter dispatch, but with progress of indirect branch predictors the benefits of traditional approach diminishes. We also evaluate the application of superinstructions technique to Ruby interpreter and explain why the traditional approach gives limited results.

Another proposal is optimization for dynamic mixin operations. Dynamic mixin opens new horizons for the adaptation of the existing code for new requirements or new environment. It can also be used as an implementation techniques for different programming paradigms, such as context-oriented programming or aspect-oriented programming. However, the current performance of dynamic mixin if used frequently is less than desirable. Inline caching, a primary optimization technique for method dispatch in dynamic languages, suffers from continuous invalidation due to dynamic mixin operations, as well as from too coarse granularity of invalidation. We propose a combination of three techniques: fine-grained state tracking, polymorphic inline caching and alternate caching to solve the issue of invalidation granularity and to allow efficient caching even in case of frequent dynamic mixin operations.

We show that our techniques improve performance of workloads, but does not introduce overly complex implementation or reduce the dynamicity.

Acknowledgments

First of all, I would like to express my deepest gratitude to my supervisors, Prof. Etsuya Shibayama and Prof. Shigeru Chiba. It is the combination of unrestrained and long-prospective insight of Prof. Shibayama and down-to-the-earth lets-get-this-published approach of Prof. Chiba that contributed most to the preparation of the contents of this thesis. Without their guidance, no doubt, this work would not have been possible.

I am very grateful to my wife Asuka, who took all the needed courage and patience to marry a PhD student, and provided me with constant support even in the darkest of times.

My thanks go to the current and graduated members of Chiba laboratory and Shibayama laboratory who never failed to produce a cheerful and comfort atmosphere to work in.

Finally, I would like to express my gratitude to Ministry of Education, Culture, Sports, Science and Technology of the Japanese government for the scholarship provided over the course of four years.

Salikh Zakirov
February 2011

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Programming languages have observed the increase in the productivity throughout their history. The abstraction level has been constantly rising, from native code programming to assemblers, and then to high-level programming languages. Dynamic languages for a long time has been acknowledged as providing the potential for high productivity (for example, Lisp), however, the performance and other concerns lead to low penetration to the mainstream market. This situation has changed in the last two decades, with several so called ”scripting” languages becoming mainstream. For example, six of the twenty leaders of the TIOBE programming languages index [2] are dynamic languages. The technological progress of computer hardware made this possible, making even the naïve implementation performance acceptable for the wide-spread use. Whole fields of programming enjoy the benefits of increased productivity due to dynamic languages, for example, web applications and application prototyping.

1.1 Dynamic languages

Dynamic programming languages is class of programming languages, that executes at runtime some behaviors, that are traditionally done at program compilation time. Example include adding new code to the system, extending type system, redefining semantics of operations. Dynamic languages are typically dynamically typed, that is, the type checking is performed at run-time, rather than at compile-time. This follows from the ability to extend the type system at run-time, which precludes the possibility of complete type checking before program execution.

The increase of productivity due to use of dynamic languages comes from several sources. The easiest to notice is dynamic typing, which relieves the programmer from the necessity to specify types everywhere in the program. However, apologists of static typing systems counter this with statements that lack of type checking reduces the quality of the program. Dynamic languages compensate for lack of static checking by increased ability to adapt the code to changing external requirements. The good example is the ability to test the code in different conditions, such as unit testing of individual classes, functional testing of components and end-to-end testing of the whole application. The flexibility of dynamic languages allows to choose the boundaries of components freely and quickly write the interfacing components. It is for this reason that the new paradigm of development, *agile programming* [48] has got most attention among dynamic language programmers. It does not mean that agile programming cannot be applied to static programming languages, but demonstrates that dynamic languages pose less hurdles and thus are more productive.

1.2 Tradeoffs of implementation techniques

The design of a dynamic language processor is governed by a tradeoff between several competing factors. The major factors influencing the design are:

- Run-time efficiency
- Implementation complexity
- Responsiveness to changes

Run-time efficiency is obviously important to make the computing tasks complete faster, thus increasing the overall efficiency of human activity, of which computation is part of. While run-time efficiency is important, there are cases, when other sides of the tradeoff take precedence. Exploratory programming or research into new languages require short implementation time, so maintaining low implementation complexity becomes paramount. The tasks that involve porting of a language processor to multiple hardware platforms also benefit from low implementation complexity substantially by reducing the time necessary to do the ports, in some cases by several orders of magnitude. Responsiveness to changes is a specific feature of dynamic language, because in static languages the program is assumed to be constant. In dynamic languages, however, the running program itself can modify its own behavior. If the program requires recompilation on each change, high rate of dynamic changes can severely impact the performance of the program. Some techniques that provide highest peak efficiency may prove useless if the lengthy recompilation is required to be done on each dynamic change. So a tradeoff between compilation speed and execution efficiency significantly affects responsiveness. A high rate of dynamic changes is typical for interactive debugging environments, and for certain styles of meta-programming.

Let us discuss some existing examples of the design and see where do they fall on the design space. Interpreter is the most popular implementation techniques for dynamic languages, because it maps the implementation code straightforwardly to the operational semantics of the language. In fact, many of the existing dynamic languages go as far as to define their own semantics via interpreted implementation!

A step necessary in processing of any language is parsing of the input. Some of the dynamic languages do construct a representation of its input in the form of an abstract syntax tree (e.g. Ruby [49]), and others process the input along with the parsing process (e.g. Forth [40]). After the input program has been parsed, some systems start executing the code by interpreting the elements of the abstract syntax tree (e.g. Ruby until 1.8), while others elect to do an additional processing step and transform the code from the abstract syntax tree into stream of operations, where operations can be native instructions (e.g. Javascript V8 [1]) or instruction of a virtual machine (e.g. Python). This design choice is mainly influenced by relative importance of efficiency and implementation complexity, as translation of the source code into instruction stream introduces some complexity, in return obtaining improvement in performance.

Compilation of the executable code to native code may reduce the overheads of the interpreted execution, thus improving the run-time efficiency, at the price of the much more complex implementation. Traditional ahead-of-time compilers may provide high peak performance, but with long compilation time, which makes them less responsive. Most significant improvements require more than just translation to native code. Example of techniques that improve performance by extracting some information about running program and using it for optimized execution are: static representation of objects using maps, type-specialized version of the methods. Type inference allows to deduce all possible type values and generate better code. Type prediction and devirtualization allows to replace generic method call sequences with static calls. The better execution performance is traded for the necessity of analysis algorithms and invalidation infrastructure, so as to ensure correct semantics even if some of the assumptions has been broken after the applying the optimization. Adaptive recompilation and trace-based compilation make focus on detecting frequently executed code blocks and thus get better return on optimization investment. Adaptive techniques is an example when responsiveness is traded off for maximum performance in the stable phase of the long-running application, such as server applications. On the other hand, the short-running or highly dynamic applications profit less from adaptive optimizations, and require higher system responsiveness.

1.3 The tradeoff point chosen by our techniques

In this thesis we propose two dynamic language implementation techniques that emphasize the two sides of the design tradeoff: low implementation complexity and applicability to the systems with high-frequency use of dynamic features. We focus on the case when dynamic features of the language are used sufficiently frequently. Our choice of the tradeoff point is based on the current design of Ruby, which emphasizes the metaprogramming capabilities and high responsiveness to dynamic changes, rather than execution efficiency. Our goal is to improve the execution performance of the Ruby interpreter without affecting its responsiveness.

1.4 Contributions of this thesis

In this thesis we studied application of superinstructions to the Ruby interpreter, explained and experimentally verified limited benefits of the traditional approach to superinstructions, and proposed a novel approach to superinstructions, which is beneficial to numeric benchmarks.

The second contribution of this thesis is proposal of optimization techniques for dynamic mixin. We proposed an effective scheme of inline caching that remains efficient even in presence of frequent dynamic mixin optimizations.

1.4.1 Position of superinstructions

Superinstructions has been proposed before with the goals of reducing code size and improving performance by reducing interpreter dispatch overhead. With progress in hardware, the memory size became less of an issue. The prior research has demonstrated, that superinstructions can be effective if the VM interpreter satisfies certain conditions: the average cost of a VM operation is low — on the order of 10 hardware instructions for 1 VM operation, and the overhead of interpreter dispatch is high. It has been demonstrated that on processors without indirect branch predictor the interpreter dispatch cost can be as high as 50% of total execution time. However, modern processors have greatly improved the quality of indirect branch prediction, and this reduces the potential benefit of superinstructions. In this thesis, we show and experimentally verify the limited benefit of the traditional superinstructions for the Ruby interpreter.

Further, we propose a novel way of using superinstructions to reduce boxing of intermediate results of floating-point computations. It benefits the numerical benchmarks. With regard to dynamicity, the superinstructions provide exactly the same response to dynamic changes as the original interpreter, thus it is suitable for highly dynamic applications.

1.4.2 Position of dynamic mixin optimizations

Dynamic mixin is a limited special case of a more general delegation techniques, which has been introduced by some popular dynamic languages such as Javascript. Ruby also has dynamic mixin, but in a limited way, because only the dynamic mixin installation is supported, and there is no reverse operation of dynamic mixin removal. The omitting is explained by the uncertainty about what to do with transitive inclusion. We implemented a variant of dynamic mixin removal which has copy semantics on mixin installation and does not remove linked copies on mixin removal.

The frequent use of dynamic mixin operations is a recent proposal, and has not been researched thoroughly. Existing inline caching techniques has low performance in this case. The techniques we propose provide an effective way to maintain the performance of the inline caching even in presence of dynamic mixin operations. The optimization is achieved by a combination of several techniques:

- Fine-grained state tracking

- Alternate caching
- Polymorphic inline caching

Fine-grained state tracking seems to be in use in several dynamic language systems which allow for updates at runtime, however, no detailed publications are available on the subject. We proposed a novel way of maintaining the fine-grained dependency links between inline caches, state objects and class hierarchy, which facilitates further novel improvement — alternate caching. Polymorphic inline caching has been known before, but it required small modification to be suitable for our use.

1.5 The structure of this thesis

The rest of the thesis is organized as follows:

Chapter 2: Background

This chapter gives an overview of existing implementation techniques for dynamic programming languages.

Chapter 3: Superinstructions

In this chapter we describe the application of superinstruction technique to Ruby interpreter. We evaluate experimentally the performance of the traditional approach to superinstructions and analyze the limited benefit. Further we propose a novel approach: arithmetic superinstructions, describe our implementation and evaluate it experimentally.

Chapter 4: Dynamic mixin optimization

This chapter proposes dynamic mixin optimization, structured as combination of three implementation techniques: fine-grained state tracking, polymorphic inline caching and alternate caching. We give the description of the techniques, outline the proof of correctness and give some generalizations. Experimental evaluation is given for the Ruby interpreter.

Chapter 5: Evaluation of dynamic mixin optimization in a compiled system

In this chapter we describe evaluation of the dynamic mixin optimization in a context of a dynamic language system with a compiler. We describe how the data from polymorphic inline cache can be used to produce highly performant compiled code, and evaluate the performance of this approach experimentally using microbenchmarks.

Chapter 6: Conclusion

Finally, we summarize and conclude the thesis in Chapter 6.

Chapter 2

Background

The implementation techniques for dynamic programming language has been researched for a long time, starting from the first Lisp systems in the late 60s. A surge in new developments and techniques occurred in the 80s, related to the development of Smalltalk and Self languages. From the 90s, so called scripting languages came into popularity, but somehow did not profit from the wealth of implementation techniques. The emergence of "Web 2.0" and proliferation of rich internet applications brought about a renaissance in dynamic language implementation research, this time mainly focused on Javascript. In this chapter we review the currently existing approaches to the implementation of dynamic languages and notable implementation techniques.

2.1 Interpreters

The easiest way to implement dynamic languages is interpretation, because interpreted systems make few assumptions on dynamic state of the program, and just recompute everything as needed. Accordingly, interpretive execution has a fair amount of the execution overhead. The overheads come from several sources: the execution of the program involves reading its representation, deciding what to do, and ultimately performing the computation itself.

One can classify interpreters based on the program representation during program execution, where on one side of the spectrum will be the systems with interpretation made over abstract syntax tree or over the program text immediately, and on the other side sophisticated representations of the code. Execution of the abstract syntax tree or program text requires repeated interpretation, that is, going from representation to the computation that needs to be done. For this reason, efficient interpreters use linear representation of the program, where individual elements more directly reference the computation necessary for the program execution [24]. The operation of such an interpreter can be described in terms of instruction set of an abstract hardware architecture referred to as a *virtual machine* (VM), because usually no hardware implementation of an instruction set exists.

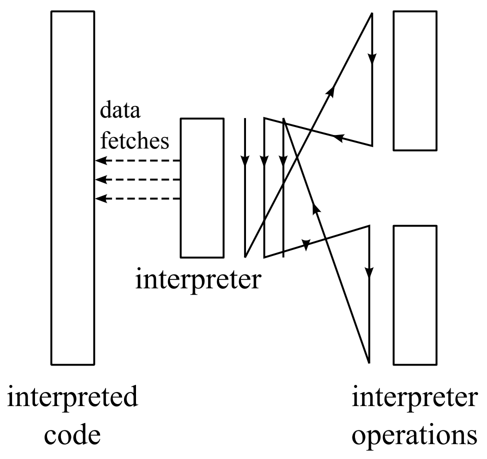
The simplest and most straightforward implementation of virtual machine interpreter uses linear representation of the program in the form of bytecodes — numeric codes for encoded operations, often chosen to match the byte size — a minimal addressable unit on many current architectures (**Fig 2.1**). Switch-based interpretation can be implemented in ANSI C, compilers for

```
while (1) {
    switch (code[PC++]) {
        case ADD:
            stack[--sp] = stack[sp] + stack[sp+1];
            break;
        case CONST:
            stack[++sp] = code[PC++];
            break;
        // ...
    }
}
```

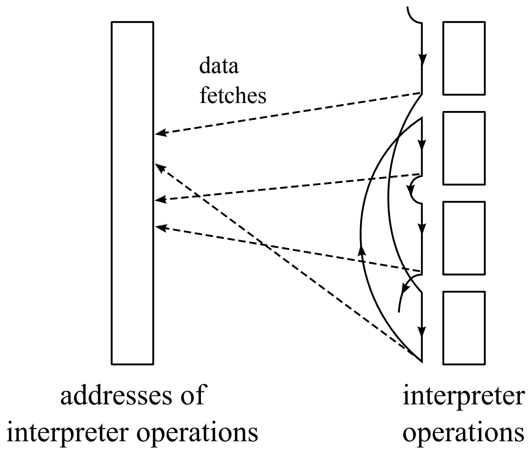
Figure 2.1. Switch-based interpreter

which are widely available on many platforms. However, compiled form of a switch-based interpreter typically executes at least two branch instructions for each bytecode, and uses the same code site for dispatch of all instructions, thus making the job of the branch predictor unit hard, and thus causing a performance overhead due to branch misprediction penalties.

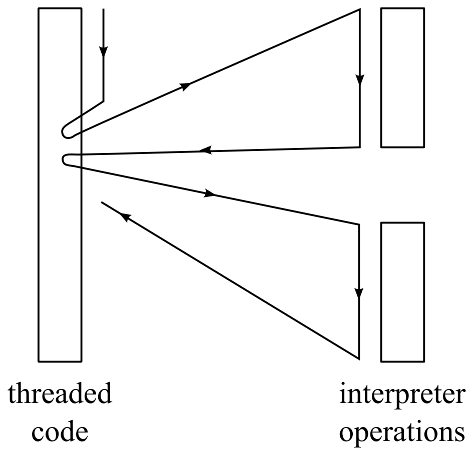
Direct threading [7] has been proposed as a techniques for compact code representation, by using a custom instruction set that matches a problem at hand. Using the carefully chosen set of software subroutines, the desired algorithm can be coded very compactly using just the references to the subroutines. The executable program is encoded as the sequence of instructions, each represented as the address of the implementation subroutine (**Fig. 2.2(b)**). Program counter points to the instruction stream, and dispatch to the next instruction involves loading the address of the subroutine, branching to it and incrementing the program counter. Threaded interpreter can be implemented using label-as-addresses extension of the widely available gcc compiler suite [66]. The figure 2.3 shows how threaded interpreter can be implemented using GCC.



(a) Switch-based interpretation



(b) Direct-Threaded interpretation



(c) Call-threaded interpretation

Figure 2.2. Approaches to interpreter implementation

Threaded code has higher performance than other interpretation tech-

```
        void *code[] = { &&CONST, (void*)1, &&ADD, ... };
        void **pc = code;
        goto **pc++;
ADD:
        stack[--sp] = stack[sp] + stack[sp+1];
        goto **pc++;
CONST:
        stack[++sp] = (intptr_t)*pc++;
        goto **pc++;
```

Figure 2.3. Direct-threaded interpreter

niques. It is currently used in many implementations [24], including the Ruby 1.9 implementation, which we used in our studies.

However, in some language implementation direct threading techniques may cause access patterns — instruction reads interleaved with data reads in nearby locations — which are handled inefficiently by some architectures that prohibit the data cache and instruction cache to have duplicating entries. Thus instruction reads followed by data read can lead to thrashing of the cache line between instruction and data caches.

Indirect-threaded code, traditionally used in many Forth implementation, is free from this deficiency, at the cost of additional memory indirection. Originally indirect threading was devised to achieve better code size reduction for variable words, that represent both the memory location to store the variable, as well as code to put the variable address to the data stack. In order to share code for accessing data variable address, it is accessed using double indirection from the executable code.(Fig. 2.4).

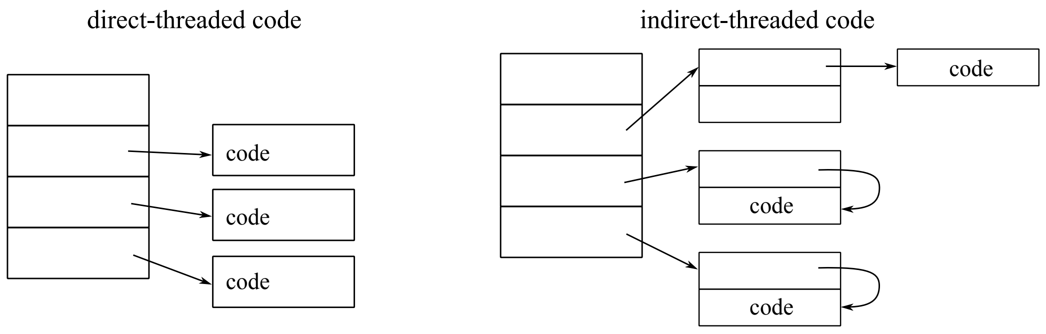


Figure 2.4. Indirect threading

Another variation of direct threaded code is *direct call threading*

(**Fig. 2.2(c)**), where dispatch to the instruction uses call instruction instead of indirect jump. It is beneficial for the portable interpreter construction, because standard ANSI C is sufficient to encode this technique. It is however slower than direct threading, although faster than switch-based interpretation [26]. Call-threading also enables easy combination of interpreted code and fragments of native code, for example, partial compilation [74].

Replacing indirect calls with direct calls, by rewriting the virtual machine instruction stream into stream of native call instructions with addresses encoded immediately much improves the branch prediction and leads to performance comparable with direct threading [8].

All described interpretation techniques of can be implemented in a single codebase, with execution details controlled via macros, which allows flexibility, for example, building the interpreter from the same source code on multiple platforms, with configuration adjusted appropriately to the available compilers for a particular platform, so that performance characteristics of the platform are taken into account.

2.1.1 Superinstructions (inline threading)

Direct threading lends itself well to further optimization. Since the dispatch between the instruction takes time, it is possible to concatenate code of instructions that are frequently executed together, and thus create a so called *superinstruction*. Inline threading is capable of reducing the overhead of operation dispatch, which is important for the interpreters with low-level instruction set [52]. As we show in chapter 2, the overhead of interpreter dispatch has became less important for processors with good branch predictor.

2.1.2 Quickening

Quickening is the technique that dynamically rewrites some operations in the interpreted code with other operations, which have higher performance. It typically involves very simplistic code rewriting without sophisticated code analysis.

Quickening is especially often used in Java VMs, where field access and method call operations has complex semantics with respect to dynamic class loading and initialization. Since only the first execution of a bytecode requires complex checks and initializations, it is beneficial to rewrite the operation to a simpler one without the checks [25].

Quickening has also been used recently to speedup interpreters, for example, to implement stack caching for the variables which are normally accessed by indirect memory access [13].

2.1.3 Register-based VM vs. stack-based VM

There exist two different approaches to represent the operands of the operations in the interpreted code. In the stack based approach, the instructions take operands from the top of the stack, and the operation result is put back to the stack, thus modifying the stack contents. This lends itself for the compact code representation, because less different operations are needed. For this reason the interpreters that use bytecodes usually use stack-based virtual machine, e.g. Java [47].

The alternative representation is register-based, where a virtual machine have several registers, which can be addressed directly as operands of the instructions. This increases the number of necessary distinct virtual machine operations, thus making it hard to fit the operation code into a single byte. However, threaded implementation techniques are a good match for register-based virtual machine, since single instruction is coded with pointer-sized word, giving a plenty of space to encode instruction operands. Obviously, register-based representation leads to increased size of the individual commands, though it has been suggested that longer operations are compensated with fewer instructions [20]. Use of register-based architectures in recent efficient programming language implementations such as Dalvik VM [11] and explicit comparisons [64] suggest that register-based interpreters provide higher overall performance.

Stack-based virtual machine is an easy target for writing source code compilers, since it removes the necessity of doing register allocation and enables simple one-pass code generators. On the other hand, register-based virtual machines with sufficient number of registers use code representation, which is close to representation used in advanced compiler optimizations such as dead code elimination and constant folding, thus making compiler building simpler.

2.1.4 Stack caching

The gap between performance of stack-based and register-based virtual machines can be compensated somewhat using *stack caching* optimization, which effectively stores the values at the top of the stack in registers (Fig. 2.5), with static or dynamic cache management [26]. In static cache management, a source code compiler is aware of a cache state, and it generates appropriate instructions. It allows to completely reduce some stack-manipulation instructions, as it suffices to for the compiler to note the state change. One can consider stack caching optimization as a kind of automatic register allocation policy, which uses simple finite state automaton to guide register allocations. However, this approach adds some complexity in instruction generator.

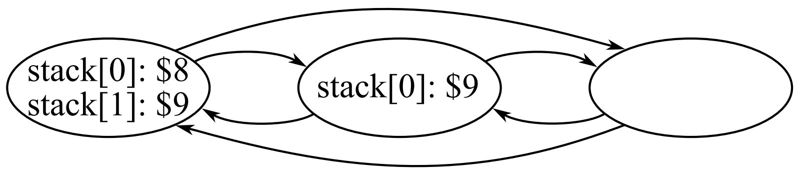


Figure 2.5. Stack caching

Dynamic approach to stack caching, on the other hand, makes management of the cache state a responsibility of the interpreter. In effect this requires duplicating each interpreter operation for each cache state, so the dispatch procedure needs to dynamically select between variants. The additional overhead of dispatch between cache states cancels out benefit of stack caching and makes the dynamic approach impractical.

2.1.5 Dynamic interpretation

Dynamic interpretation [73] uses a special non-linear intermediate representation of the executed program, where each instruction is specialized on the types of its operands, and special type-directed nodes are used to dispatch on the type of the operation result. The dynamic representation is built on demand during first execution of the regular static representation. The dynamic representation is a flow graph, where nodes contain the specialized instructions, and edges direct the program flow, based on control and type changes in a program. Dispatch on the dynamic representation is more expensive than in traditional interpreters, but this is compensated by higher efficiency of the type-specialized instructions, effectively moving the type checks out of the instructions and into the type-directed edges. The main promise of dynamic interpretation is the profile information built automatically in the dynamic code representation, which potentially can be used for compilation to more efficient native code, or even for optimizations on the dynamic representation.

2.2 Inline caching

One of the possible observations on execution of dynamic programs is distribution of object types on method calls. It has been noted [36] that many call sites call only one method during the whole program execution. Even for the call sites which call more than one target method, it is often true that one method is called much more often than others. This makes feasible prediction of the target method based on observation of earlier execution. Inline caching [23] associates a cache with the call site, in some implementation the code and cache data can be in the same memory area, which is why it is called *inline* caching. Typically inline cache stores the target of the last method invocation, as well as type of the method receiver. On the subsequent call site execution, the type of the receiver is checked against the one stored in the inline cache, and if the type matches, the target method from the cache is used without full method lookup. In the case of dynamic languages, where method implementation can be redefined or overridden by a dynamic mixin, inline caching requires some means of invalidation, for example by storing state counter snapshot. We elaborate on this in the chapter 4.

Inline caching has been extended in the past to store more than one target method for different receiver types [36]. The authors found out, that in a typical Self program the call sites can be classified into three categories: monomorphic — with only one receiver type, polymorphic — with a few receiver types, and megamorphic — with very many receiver types. Polymorphic inline caching allows to store up to a fixed small number of different method targets for different receiver types and thus to improve performance of polymorphic call sites. The information that polymorphic inline caches collect in the course of program execution can also be used for other optimizations, such as inlining during recompilation of the code, for example, as we describe in the chapter 5.

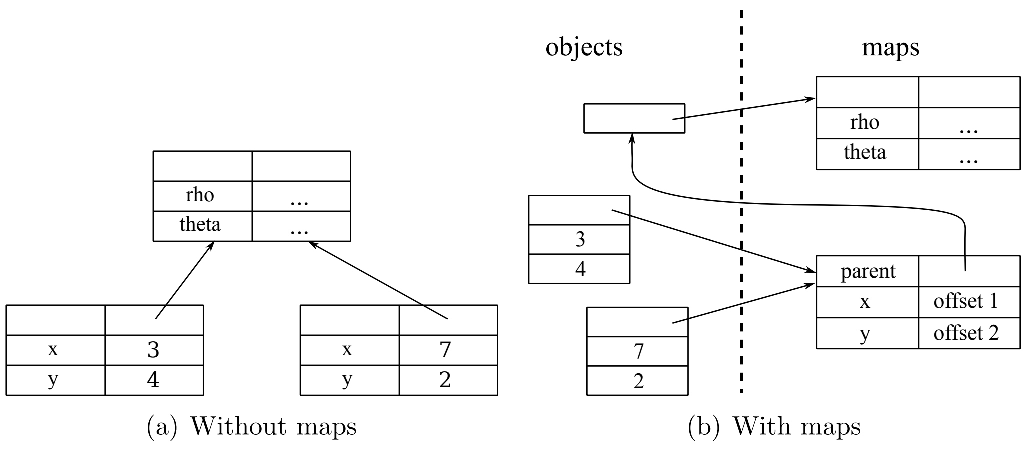
2.3 Compilation techniques

Compilation of dynamic languages usually involves more advanced techniques than interpretation. This increases implementation complexity considerably. Compiled code removes certain overheads of interpretation, for example, dispatch overhead between instructions. Since compilation already involves nontrivial code transformation, it is natural that typical compiler implementations for dynamic language do other transformations to increase performance.

A general approach is to detect some repeated patterns in program execution and specialize the code for that pattern of execution.

2.3.1 Object maps

Objects in dynamic languages typically do not have their structure fixed, instead, new fields can be added or removed at any time. Straightforward approach like representing each object as a hashmap would cause space and time overheads, especially because the majority of the objects during program execution have similar contents, as for example would have objects corresponding to the records in the database. An *object map* technique [14] addresses the overheads resulting from dynamic object representation. Though the structure of an object can change at any time, in fact the majority of objects have constant structure during its lifetime. With maps, objects are represented as simple list of the slots containing values of object fields, and an additional link to an object map, which describes the layout of object: the names of the object fields and their offsets. Maps are immutable and are shared by all objects with the same structure. In the case of a rare event when object changes its structure, its object map pointer is updated to point to a new map.



(a) Without maps (b) With maps

Figure 2.6. Object maps

2.3.2 Method specialization

The difficulty of efficient compilation of dynamic languages mostly arises from the fact that no exact type information is available, and compiled methods must accept arguments of any types. *Customized compilation* [14] creates a separate specialized version of compiled code for each receiver type. Due to this, the static type (or an object map) of the receiver is known during compilation, and some operations like field accesses or calls to other methods of the same object can be compiled efficiently. This is similar to generic programming, where generic description is used to create customized concrete code for each set of parameters. So each method is implicitly generic with respect to concrete receiver type.

2.3.3 Message inlining

Inlining is the substitution of the method body in place where the method is called. Inlining in dynamic programming language implementation is impeded by the fact that the type of the receiver may not be known until run time. However, customized compilation and message splitting allows to increase the amount of type information, available to the compiler. In cases, where the type of the message receiver is known at the time of compilation, compiler may look up the method and compile the method body directly instead of emitting the method call [14]. In some cases, the target of the method call can be determined unambiguously even if the type information is incomplete, for example when only a single method implementation with the given name exists in the system.

The same approach can be used for compiling primitives operations, which are commonly encoded as method calls with predefined method names. Inlining the primitive operations, which may consist of just several hardware instructions, significantly improves performance compared to using the calls into the external functions of the virtual machine.

Message inlining can be applied even in cases where the type of the receiver is not known statically, but only probabilistically is known that some type of the receiver is more likely than others. To maintain correctness the additional check on the type of the receiver is necessary. The technique is called *guarded inlining* in this case. A number of checks can be employed to guard the inlined code, such as type test or the lookup and method test [22].

Inlining obviously removes the overhead of the method call, but the large part of the improvement comes not from the removed overhead, but from the optimizations, that can be applied to a larger code body resulting from the inlining. Thus inlining is an enabling optimization.

Redefinition of the methods or dynamic loading of new code is a possible operation in dynamic languages, and thus some of the assumptions made during method inlining may be invalidated later. This requires guards or an compiled method invalidation infrastructure to discard the code that relies on assumptions that are no longer true. Especially difficult is invalidation of the currently active methods. *On-the-stack replacement* [27] is an approach that restores the source-level state of an active methods, recompiles the method without the broken assumption and reconstructs the machine-level state of for the new version of the compiled method. It has been reported informally that on-the-stack replacement involves significant engineering difficulties. The easier approach is using guards in all inlined methods or dynamic code patching.

2.3.4 Type inference

Using type inference allows compiler to acquire more information about types of the objects, by doing dataflow analysis. However, in the case of dynamic languages, type inference is hindered by the dynamic dispatch, so that the target of the method calls is not known until the runtime. Constraint-based type inference [3] deals with this problem by using iterative analysis, and using the inferred type information to determine targets of the method calls which were not known in advance. The iteration is terminated after reaching a fixed point or after exceeding a threshold for a number of iterations. The information produced by type inference can be used to guide other optimizations such as method inlining.

2.3.5 Message splitting

Sometimes the static information about values (e.g. the information about the literal value type) can be lost in merges in control flow graph, because the same variable can get different values from different merged branches. The compiler may elect to duplicate the code following the merge for each of the merged branches, so that the available information could be used for optimizations like the method inlining. Message splitting is particularly important for language like Self, where even control flow is represented as closures passed as parameters to methods.

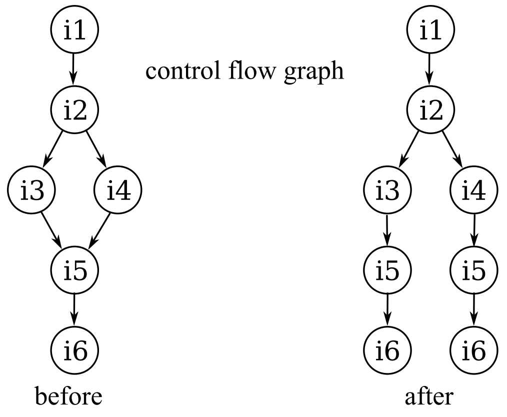


Figure 2.7. Message splitting

2.4 Adaptive techniques

Dynamic languages challenge implementers with insufficient information about the program, types of variables and targets of method calls. To make efficient execution possible, more information is needed than is contained in the code itself. Adaptive techniques work around this problem by observing execution of the program with specific input data, and making optimization decisions based on observation data. The time period, during which optimization is postponed directly affects the resulting responsiveness to changes, making adaptive techniques more suitable for long-running applications.

2.4.1 Just-in-time compilation

Just-in-time compilation [23] has been devised as a tradeoff between higher efficiency of compiled method execution and higher space requirement for the compiled code. Method can be compiled from original virtual machine bytecode to native executable code dynamically, when the method is executed for the first time. Since the original representation remains available, a code cache of limited size can be used when the memory is scarce. When code cache overflows, some compiled methods can be discarded and recompiled later if needed.

2.4.2 Mixed-mode execution

Mixed-mode execution deals with space and time overhead of pure compilation system by allowing execution of code without prior compilation. Since a good deal of the typical program code is executed only once, there is no point in expending substantial effort in compilation of that code. In mixed-mode execution [4], all code is initially executed using an interpreter, without incurring cost of compilation. At the same time methods and code blocks are profiled using counters, and as soon as counter for some method exceeds threshold value, the compiler is activated. In this way, the overhead of compilation is restricted to frequently executed methods and loops, where it is likely to be compensated by benefit in reduced execution time.

2.4.3 Adaptive recompilation

Availability of dynamic compilation infrastructure opens the possibility for collecting profiling data about running application and using it for optimized recompilation of frequently executed code [6]. Typically an adaptive compilation system has two or more levels of optimization — an unoptimized level, that allows to collect information about target of the called methods, and several optimization levels. The base unoptimized level is usually implemented as an interpreter or fast non-optimizing compiler, because it must handle a lot of code, including initialization code, which is run only once. The profiling information allows to select hot methods or other fragments of code that are suitable for expending large effort for optimized compilation, because it is highly likely that these pieces of code will be executed many times more. While just-in-time compilation or mixed-mode execution already provide the benefit of concentrating the compilation effort on the code that brings maximal benefit, use of profile information enables other optimizations, such as profile-guided inlining, code positioning, instruction scheduling and multi-version code.

Adaptive techniques also allow for continuous program optimization, when profiling is performed on optimized code as well as on non-optimized, and if profiling detects that conditions changed significantly from the time when the optimized methods were compiled, it may be beneficial to recompile the code to better match to changed conditions. For example, LLVM [45] is an example of intermediate code that is targeted for lifelong program optimization. Profiling of the optimized code is usually involves using low-overhead sampling profiling instead of code instrumentation.

2.4.4 Type feedback-directed inlining

Using a run-time profiling is a viable alternative to type inference in guiding inlining decisions, and can be more effective [35]. Inlining decisions on method calls are guided not by the statically provable type of the receiver, but by observation of the real behavior of the application. Information about frequency and receiver types allows to apply inlining precisely in places where it is going to bring most benefits.

2.5 Trace compilation

Trace compilation [30] differs from traditional compilers in its choice of compilation unit. Instead of using methods as compilation units, trace compiler observes and compiles the real execution trace. Trace can include operations from several different methods, thus providing inlining automatically, and it excludes the operations that were not executed during trace collection. To maintain correctness, it is necessary to insert guards into the trace to ensure that method dispatch and conditional operations will branch in the same way was as during trace collection. If the condition check fails and execution must proceed in a way different from original, trace execution typically exits the trace and falls back to interpreted execution.

2.5.1 Trace recording

The typical approach to detect a good trace head is to count executions of backward branch targets. Since loop heads are backward branch targets, counting execution frequency allows to detect loop headers of hot loops. The trace collection proceeds to record operations executed by the interpreter and terminates either if the loop header is encountered again, thus closing the loop, or if the trace recording fails due to overly long trace or unsupported by compiler condition, such as exceptions or calls to native methods.

2.5.2 SSA-based trace optimizations

Compilation of a linear trace allows for cheap and easy generation of static-single-assignment (SSA) form, which in turn facilitates aggressive optimizations such as common subexpression elimination, dead code elimination and loop invariant hoisting. Linear structure of the code make implementation of the optimization both easy and efficient.

2.5.3 Trace linking

Loops rarely have strictly linear control flow, and conditional branches frequently occur in the loop. Trace compilation has several approaches to treat branched control flow. *Trace tree* compilation [29] continues trace collection from trace exit, where the guard check failed, and later restarts the trace compiler from the branching point, reusing the state of the compiler at failed guard. This allows for direct use of hardware branch instructions between compiled traces. The tail of the loop is traced and compiled separately for each branched trace to keep the code linear. In case of highly branched loop body this may cause significant code bloat, as the code after any branch needs to be duplicated in every compiled trace. Another problem of trace tree compilation on highly branched trace trees is the amount of recompilation, because the trace tree needs to be recompiled on each new trace addition, and so compilation cost is quadratic on the number of traces.

Trace linking [16] instead starts a new trace compilation at the branch point of the original trace. This requires some compensation code to transition from the compilation state at the branch point to the new trace, but allows to reuse traces in the case when control flow merges back after branching. Since the merge can be detected only when compiling a second or later traces in a loop, at least one copy of the tail of the loop will have been already compiled. Trace merging is effective only for third and later traces that pass through the same control flow nodes, but nevertheless is an improvement compared to trace trees. In the case of highly branching loop, the trace tree approach causes high overhead for repeated compilation of the duplicated loop tail, and trace linking allows to limit the overhead.

2.5.4 Trace nesting

The high-level structure of the program source code can help to the trace collector to determine the exact extent of the loop in the bytecode, and thus to collect traces for the outer and inner loop separately [31]. Having a strict correspondence of traces to loops in the structured high-level program provides solution for the code bloat caused by "inside-out" compilation of outer loop as inner traces. To implement trace nesting, the trace collector

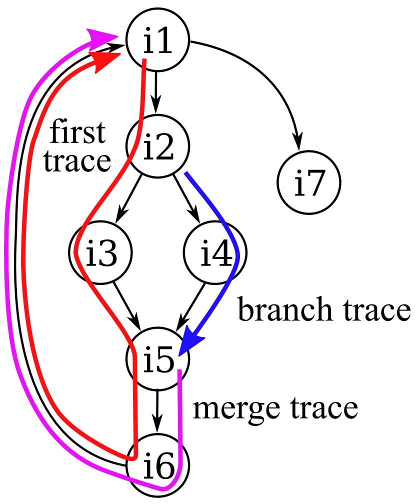


Figure 2.8. Trace linking

tracks the current loop, and terminates the trace collection on leaving the loop, or switches to the inner loop trace on entrance to a nested loop.

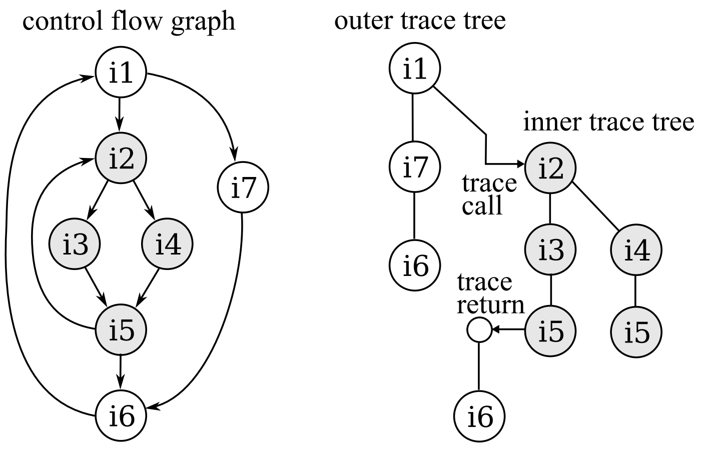


Figure 2.9. Trace nesting

2.5.5 Just-in-time specialization

Just-in-time specialization is a variation on the trace compiler, which make focus on specialization rather than on trace collection.

Psyco [57] does not separate the trace recording and compilation steps, instead it compiles the code and executes it by small steps. The specializer then tries to guess specializing decisions, that can be profitable in the future. The specializer saves continuations of the compiler for specialization and dispatch points, and in case of guard failure restarts execution of the specializer from the point where execution branched differently from prior executions.

2.5.6 Meta-techniques

Combination of existing implementation of execution techniques allows for interesting application [58]. PyPy project [10] proposed to implement interpreter for the dynamic language in RPython — a statically typeable restricted subset of Python. Applying tracing JIT involves two level of interpretation — a language interpreter, which is written in RPython and which interprets user program, and a tracing interpreter, which traces operations of the RPython interpreter and compiles them. A single hottest loop executed by the tracing interpreter is the bytecode interpretation loop of the language interpreter. Thus, if tracing compilation is used naïvely, the number of traces through interpretation loop will be equal to the number of operations in language interpreter. In order to increase the benefit of tracing compiler, the tracing compiler uses a combination of program counter of tracing interpreter and language interpreter, and tries to detect loops in the user program, rather than in the language interpreter. In order to do that, the developer of the language interpreter needs to provide hints on what is the language interpreter program counter, and what operations are backward branches.

While the performance of the two-level interpreter can be several orders of magnitude slower than direct interpretation, application of trace compilation allows to produce a direct interpreter, or even compile specialized user application loops, resulting in competitive performance. At the same time, construction of an interpreter in a high-level language as RPython makes implementation much more simple and maintainable. Thus, meta-technique provide a good potential for dynamic language research.

Chapter 3

Superinstructions

3.1 Introduction

Dynamic languages are characterized by fully dynamic typing, that is, the program text does not make or enforce any conditions on types of variables, and the same program may be fed values of different types during single run. This prompts implementer of dynamic languages to provide for a full generality, proliferating type checks and conversions as a frequent operation. A prominent example is an interpreted implementation, where individual operations are prepared to handle values of any incoming type. Ruby 1.9 is an example of such an implementation. However, fully polymorphic instructions of the interpreter come at a price: each instruction need to perform type checks. Another disadvantage of a fully dynamically typed interpreter is that it must treat all values uniformly, which in practice results in using a pointer-sized encoding of data, representing either a pointer to an object on a heap or a bit-encoded data directly. In particular, the data that cannot fit into the pointer-sized word, such as double precision floating point number on a 32-bit architecture, needs to be stored on the heap, even in the case when it is only a temporary computation result.

Performance of Ruby is a subject of much concern. Speed of numeric benchmarks is particularly often quoted, where Ruby sometimes is 100 or more times slower than implementation in C. Thus, improving Ruby performance is an important problem. 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 [55] 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 mispredictions due to 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 chapter 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 23%.

3.2 Superinstructions

The Ruby programming language is implemented as a virtual machine interpreter since version 1.9. Among many approaches to interpreter performance, superinstructions have got much attention. In this chapter, 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 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. describe [24], 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 3.1). 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.

Table 3.1. 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	$1019 \cdot 10^8$	$1287 \cdot 10^8$	$1097 \cdot 10^8$	$1161 \cdot 10^8$
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%

Moreover, interpreter dispatch does not constitute large proportion of execution time of Ruby on numeric benchmarks. Our experiments with straight-

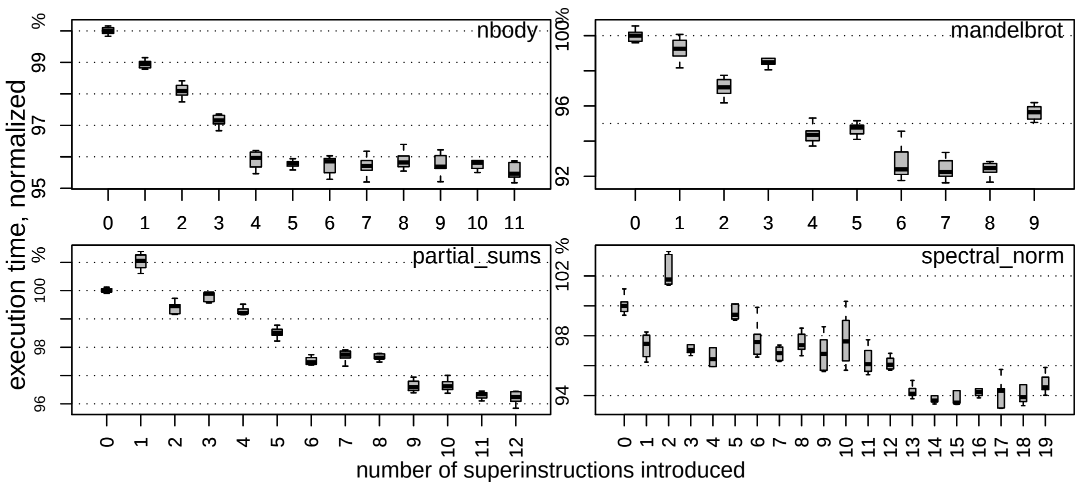


Figure 3.1. Effects of introducing naive superinstructions one by one.

forward static superinstructions implemented in Ruby 1.9 [60] (referred to as “naive superinstructions”) showed limited benefit in performance of about 4% (see section 3.4). Selection of superinstructions is based on frequency of occurrence of instruction pairs in benchmark execution trace.

Figure 3.1 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 figure 3.2 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 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.

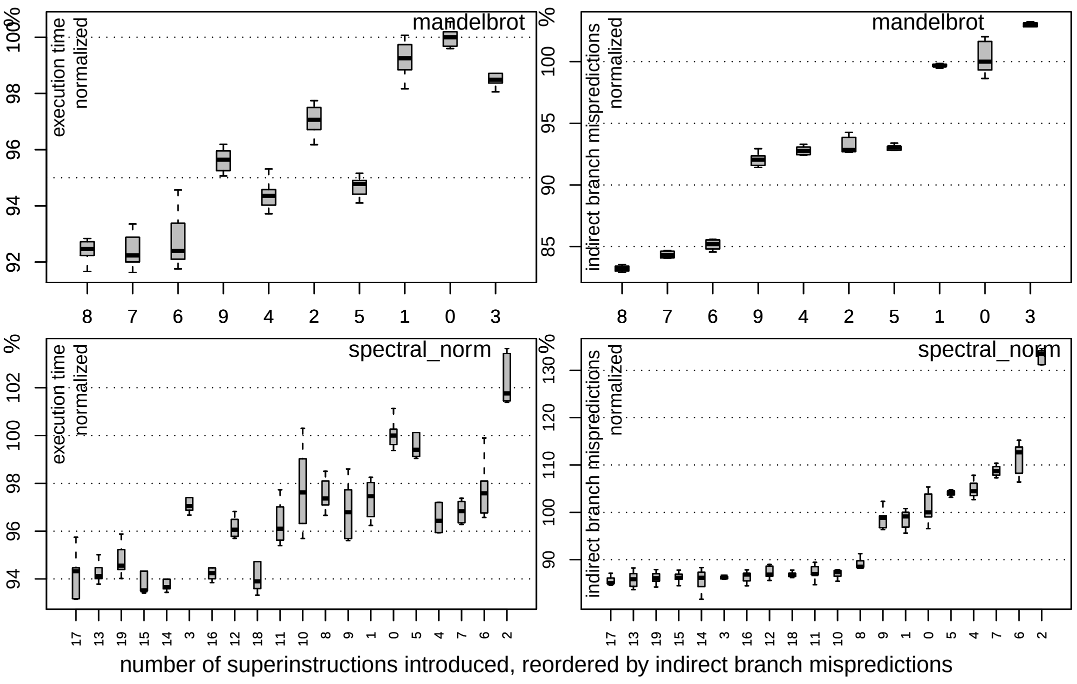


Figure 3.2. Comparison of execution time with indirect branch mispredictions.

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.

3.3 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 3.4.4).

3.3.1 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 using tagging, and floating point number are implemented with boxing.

Ruby has five instructions that deal with floating point numbers: `opt_plus`, `opt_minus`, `opt_mult`, `opt_div`, and `opt_mod`. Figure 3.3 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:

Figure 3.3. Source code of `opt_plus` implementation

```
DEFINE_INSN opt_plus
()     /* immediate parameters */
(VALUE a, VALUE b)  /* stack inputs */
(VALUE val)   /* stack output */
{
    /* ... */
    if (HEAP_CLASS_OF(a) == rb_cFloat && /* check types of arguments */
        HEAP_CLASS_OF(b) == rb_cFloat &&
        BASIC_OP_UNREDEFINED_P(BOP_PLUS)) { /* and validity of optimization */
            val = DBL2NUM(RFLOAT_VALUE(a) + RFLOAT_VALUE(b));
    } else {
        PUSH(a);
        PUSH(b);
        CALL_SIMPLE_METHOD(1, idPLUS, a);
    }
}
```

```
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.

3.3.2 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. Local variable access instructions occur between arithmetic instructions especially frequently, and thus deserve creation of special superinstructions of length 3, *arith. op–var. access–arith. op.*

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 these 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.

Arithmetic 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.

3.4 Experiments

3.4.1 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`.

3.4.2 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.

3.4.3 Results

Figure 3.4 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 figure 3.5 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:

```
tr = zrzr − zizi + cr
```

```
getdynamic zrzr, 3
getdynamic zizi, 3
opt_minus
getdynamic cr, 3
opt_plus
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.

Tables 3.2 and 3.3 give some profiling numbers on naive and opt-opt version for comparison with baseline version.

3.4.4 Profiling data

Profiling data illustrates why benefit from opt-opt superinstructions is possible. Garbage collection takes big share in execution time (figure 3.6). 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),

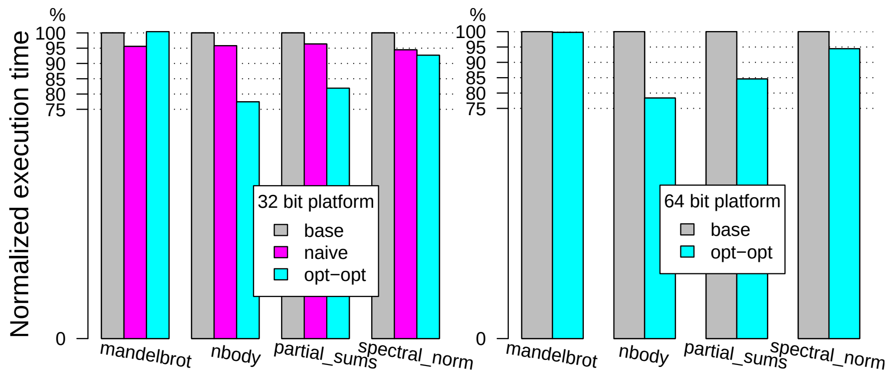


Figure 3.4. Execution time of benchmarks, 32 bit mode (left) and 64 bit mode (right). The lower, the better.

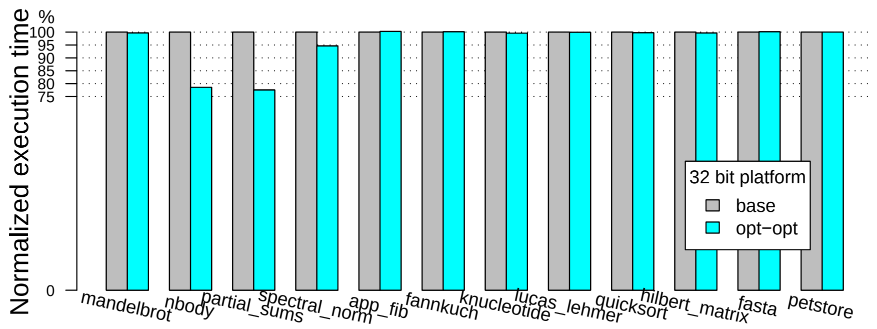


Figure 3.5. Execution time of other benchmarks, 32 bit mode

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 3.4), it is quite reasonable that any reduction in boxed values allocation will produce visible benefit in performance. The **table 3.5** shows the measurements of number of executed arithmetic operations in the benchmark runs, the number of garbage collections and observed improvement. It can be seen that the number of garbage collections and reduction in garbage collection count due to use of superinstructions very closely matches. The expected benefit due to reduction of garbage collections, received as the product of garbage collection ratio to the reduction in garbage collection number, for two benchmarks, *nbody* and *partial_sum* ex-

Table 3.2. 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	$1000 \cdot 10^8$	$1300 \cdot 10^8$	$1100 \cdot 10^8$	$1100 \cdot 10^8$
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 3.3. Branch misprediction in opt-opt version

benchmark	mandelbrot	nbody	partial_sums	spectral_norm
total cycles	$640 \cdot 10^8$	$640 \cdot 10^8$	$640 \cdot 10^8$	$650 \cdot 10^8$
br. misp. stall cycles	$14 \cdot 10^8$	$8.2 \cdot 10^8$	$8.6 \cdot 10^8$	$3 \cdot 10^8$
hw. instructions	$1057 \cdot 10^8$	$997 \cdot 10^8$	$922 \cdot 10^8$	$1114 \cdot 10^8$
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%

plains the great part of the run time reduction. In the case of *spectral_norm* benchmark, the obtained reduction is less than expected from GC reduction. For *mandelbrot* benchmark, the pure arithmetic superinstructions did not apply, so there were no reduction.

Table 3.6 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.

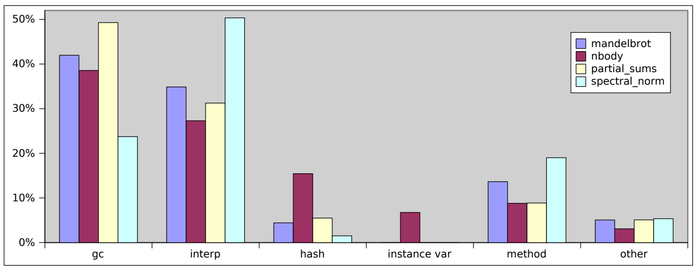


Figure 3.6. Sampling profiling

Table 3.4. 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%

3.5 Related work

An alternative approach to reducing overhead of boxing floating point numbers in Ruby has been evaluated by Sasada [62]. 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 0–23% improvement of opt-opt superinstructions.

Kawai [41] 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 interme-

Table 3.5. Dynamic instruction counters

	benchmark	mandelbrot	nbody	partial_sums	spectral_norm
base	arith ops	204253320	240000373	227650007	524460431
base	float allocation	203253333	232000433	274750027	157327596
base	garbage collections	26115	30170	35065	11291
opt-opt	merged arith ops	0	84000105	70650000	157325290
opt-opt	floats allocated	203253333	148000328	204100027	106239988
opt-opt	garbage collections	26115	19252	26080	7702
	reduction in allocation	0%	36%	26%	32%
	reduction in GC	0%	36%	26%	32%
	GC share	38%	42%	50%	24%
	expected reduction	0%	15%	13%	8%
	reduction in runtime	0%	23%	18%	7%

Table 3.6. Applicability of opt-opt superinstructions in benchmarks

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

diate floats. Stack allocation of intermediate computation results was earlier discussed by Steele [67].

Owen et al. [51] proposed lazy boxing optimization in context of compiler for Java-like language with generics over value-types. Lazy boxing works by allocating 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 [55] 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. [52] achieved good performance improvements for the low-level interpreter with RISC-like instruction set and somewhat smaller improvements for Objective Caml interpreter. Interpreters they studied 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. [37] 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. [56] 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.

3.6 Summary

We evaluated the effect of naive superinstructions for Ruby interpreter, and verified, that it has limited effect due to high quality of indirect branch predictor in modern hardware, as well as due to VM operations being complex. The result of experimental evaluation confirms the expectation. Improvements of performance using naive superinstructions is about 4%.

Further we proposed and evaluated a novel way of constructing superinstructions for Ruby 1.9 — arithmetic superinstructions. Reduction in floating point number boxing provides a visible reduction in GC overhead and thus improves overall performance by 0–23%.

Chapter

4

Dynamic mixin optimization

4.1 Introduction

Many dynamic constructs available in popular scripting languages such as Ruby, Python and Perl are used in constrained ways. For example, definition of a new method at runtime is being used to adapt the programming system to the environment and to create helper or convenience methods. Typically this is done during program or library initialization stage. Although optimization of dynamic dispatch is well-explored [71], the case of dynamically changing class hierarchy has not been widely considered. The performance of current implementations in case dynamic method redefinition occurs is not as good as we would like it to be. For this reason current applications use dynamic techniques at program run time sparingly. Together this forms a kind of chicken-and-egg problem, because developers of interpreters and virtual machines do not expend much effort in optimizing components that are not widely used by applications.

However, there exist certain applications, that would greatly benefit from dynamic technique such as dynamic mixin, which allows to insert mixed-in class at arbitrary place in class hierarchy. For example, during live development on an interactive system without stopping a program, a code modification can be introduced as a dynamic mixin installation, which overrides old versions of code with new versions. The widely known examples of live systems are Lively Kernel [38] and Squeak [39]. The programming systems steadily acquire complexity, and programming in a live environment proves to be a challenge. According to anecdotal evidence from developers of Lively Kernel system, it is useful to restrict the scope of modifications to some parts of the program. Dynamic mixin provides a means to control the application of overriding methods by inserting or removing the mixin dynamically.

Another example of application of dynamic mixin is refactoring of preexisting applications to add support for context-awareness. Recent mobile applications running on mobile phone have a life-cycle very different from traditional desktop applications. The geolocation data obtained by the means of GPS receiver or scanning Wi-Fi access points can be used to modify the behavior of software to match the context. Context information can be provided by other sensors in the mobile device, such as luminosity sensor or microphone. Dynamic mixin technique makes it possible to introduce variation in behavior without redesigning the original context-unaware application. Dynamic mixin technique can also be used as a substrate for implementation of paradigms such as dynamic aspect-oriented programming [54] and context-oriented programming [34].

Implementation of dynamic mixin in Ruby is currently not well-tuned to these use cases. It is short of being useful in full measure for two reasons: the operation of mixin removal is not provided, though it can be implemented in straightforward way; second, global method cache and inline caches — the optimization of method dispatch — rely on global state tracking, which can cause significant overhead if dynamic mixin functionality is used with high frequency. The particular issue with frequent changes is inefficient cache invalidation.

In this chapter we propose fine-grained state tracking as a solution for efficient cache invalidation in Ruby, which allows to reduce the overhead of cache flushes on dynamic mixin inclusion or other changes to classes. Our solution associates state objects with method lookup paths, which provides strong guarantees in case of unchanged state object. We assume that an executed application dynamically changes its behavior by mixin inclusion with high frequency, and that the system is alternating between few states. This happens, for example, when mixins are used to represent advice application at *cflow* pointcut, by including mixin on each entry to dynamic context of pointcut and excluding it on exit. Switching layers in context-oriented programs can be straightforwardly represented by dynamic mixin inclusion and can happen arbitrarily often. Strong guarantees that fine-grained state tracking provides allow us to improve performance of alternating program behavior. We propose an algorithm of caching alternating states, based on fine-grained state tracking and polymorphic inline caching. We saw six-fold performance improvement on a microbenchmark, and 48% improvement on a small dynamic-mixin heavy application. We also consider generalization of the caching algorithm to the delegation object model with thread-local changes to class hierarchy [33], and develop an extension to the caching algorithm — thread-local state caching, which is important for efficient implementation of context-oriented with-active-layer program construct.

The contribution of this chapter is as follows: we propose an algorithm of fine-grained state tracking to optimize dynamic method lookup, and show how its application to Ruby can reduce overhead of global method cache. Further, we propose an algorithm of caching alternating states that makes dynamic mixin inclusion and exclusion much more amenable to performance optimization by preventing inline cache misses. We also describe thread-local extension to caching algorithm.

4.2 Ruby and dynamic mixins

Ruby is a dynamic pure object-oriented language, which has got much attention recently. As Furr et al. found in [28], the majority of applications in Ruby use dynamic features to some extent, either directly or via standard libraries. The purposes range from adapting to environment to reducing amount of typing, while providing rich and convenient API. However, in majority of cases use of dynamic features is limited, and can be expressed without resorting to use of eval construct.

When talking about dynamism of application written in object-oriented language, one can classify the extent to which dynamic features are used. The lowest level involves creating a fixed class hierarchy and relying on dynamic method dispatch and polymorphism to achieve desired program behavior. This limited use of dynamic features is quite common and is possible even in statically typed languages like C++ or Java. Furthermore, the majority of research on dynamic language performance concerns exactly this level of dynamism. The next level of dynamism arises when the program can modify its behavior by changing the relationships between existing classes, but rarely creates code on the fly. The main assumption is that changes in class hierarchy and reuse of code happen much more often than creation or loading of new code. This is exactly the scenario we are targeting with our research. The highest level of dynamism we can think of is complete and thorough inclusion of meta-programming, a system that repeatedly generates substantially new code and incorporates it into the running system, constantly changing its state without much repetition or code reuse. We consider that a subject of future research.

Ruby as a programming language supports all three levels of dynamism. The lowest level of dynamism is implicit in the semantics of the method call. Mix-in composition of modules provides a flexible way to handle and rearrange units of behavior, representing the second level of dynamism. The third and highest level of dynamism is achievable by constructing arbitrary source code strings and passing them to eval function.

In recent research the delegation object model [46] has been suggested as a universal substrate for implementation of many programming paradigms. For example, substantial part of aspect-oriented programming can be formalized and implemented on top of delegation object model [33]. So is context-oriented programming [63]. The fundamental operation, on which these implementations are based is dynamic insertion of an object to a delegation chain or its removal. We call that operation *dynamic mixin inclusion* (respectively *exclusion*). Dynamic mixin inclusion can represent weaving of the aspect, activation of the context layer, or even smaller change in program state, as we describe below. Contrary to the typical use of mixins as they were conceived by the inventors [12], dynamic mixin composition happens not at the application compile time, but at runtime. Moreover, dynamic mixin inclusion cannot be treated as a solitary or infrequent event. For example, programming in context-oriented style may involve frequent change of layers. Recent trend in application of dynamic aspect-oriented programming is *self-tuning* aspect [70], which can dynamically install and remove other aspects or even reweave an optimized version of itself. Representation of some constructs, such as *cflow* and *within* pointcuts of AspectJ [42], is possible through repeated operations of mixin installation and removal for each entry into and exit from the dynamic context of the specified pointcut, which can happen with arbitrary frequency. For example, the pointcut cflow(call(A.f())) && call(B.g()) can be implemented as two mixins, one of which is permanently inserted to intercept the call to A.f(), and the other is inserted and removed dynamically. Each time method A.f() is called, the first mixin will (1) insert a mixin with advice implementation at B.g(), then (2) execute the original method A.f() by a superclass call, and after it gets control back from a superclass call, (3) remove the advice mixin. In this way, the advice mixin inclusion and exclusion at B.g() happens for each call of A.f(). The performance of programming systems that perform mixin inclusion operation with high frequencies have not been studied much, and our research goal is to try to fill the gap. We make dynamic mixin the main target of our consideration.

Mix-in composition in Ruby is provided by modules — a kind of class that can be dynamically inserted as a superclass at arbitrary place in class hierarchy (we use the term *mixin* to denote module that is used for dynamic inclusion). The inclusion can be done at program run time. **Fig. 4.1** gives an example of a server application. The mixin ServerMonitor, after having

```
class Server
  def process() ... end
  ...
end

class NetworkServer < Server
  ...
end

module ServerMonitor
  def process()
    ...       # monitor request
    super # delegate to superclass
  end
end

# this can be done during program execution
NetworkServer.class_eval do
  include ServerMonitor
end
```

Figure 4.1. Example of using mixin inclusion in Ruby

been included to the class NetworkServer, intercepts calls to the methods of the base class Server, and provides monitoring functionality, while delegating the operation itself to an original implementation in Server. Within an overriding method implementation, it is possible to call the next overridden implementation in superclass chain by using super keyword; the system automatically passes all the arguments to a superclass method. The construct class_eval is equivalent to opening the class definition, but can be used in an arbitrary context, even deep in the call chain, while regular class definition is only allowed in the top-level context of the source file. The class hierarchy before and after mixin inclusion is shown in **Fig. 4.2**.

Current Ruby implementation has a drawback concerning dynamic use of mixins, as it allows dynamic inclusion of a mixin, but does not provide a symmetric operation of mixin removal. A remove operation reportedly has been omitted because of ambiguity of mixin remove operation due to the fact, that mixin inclusion involves creating a copy of the original mixing object, so that a mixin can be installed multiple times in different places in class hierarchy. We chose a simple semantics for the remove operation, which does not affects copies of a mixin. Mixin removal implementation is straightforward and is very similar to mixin inclusion. Operation of mixin removal greatly facilitates implementation of context-oriented programming constructs in Ruby, because it allows expressing the with-active-layer construct with a bunch of mixin inclusion operations on entering the dynamic scope of layer activation, and mixin exclusion on exit. In the above example, it is useful to be able to remove the monitoring mixin after monitoring is no longer needed.

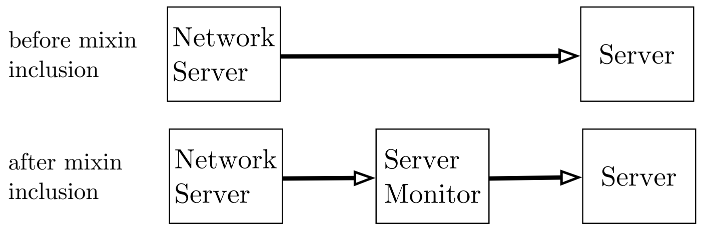


Figure 4.2. Class hierarchy before and after mixin inclusion

We implemented mixin removal operation using a development snapshot of CRuby 1.9.2 as a base. Let us consider the method lookup algorithm first. Each object has an embedded class pointer. When a method is called on an object, first the method name is looked up in the method table of the class object. If the method is not found, search continues in the superclass of the class, and so on. The algorithm is common to many object-oriented programming languages. If implemented naively, it incurs significant overhead on each method call. That is why Ruby employs several optimizations to reduce method dispatch overhead.

Ruby uses *method cache* optimization [44]. Method cache is a global hash table, indexed by a combination of a method identifier and a class identifier, which is consulted before doing regular method lookup. In Ruby, method cache has 2048 entries by default. Since normal method lookup typically involves multiple hash lookups, and hit in method cache requires just one lookup, method cache usually benefits the system performance.

To further improve method dispatch performance, Ruby includes an implementation of *inline caching*, techniques pioneered in Smalltalk [23]. Inline caching heavily relies on an assumption that most call sites are monomorphic, i.e. they mostly dispatch to a single target method during application run, so cached in the call site prior lookup result can be used in future instead of normal method lookup. In CRuby 1.9, the program is compiled to a stream of interpreted instructions, which may have arguments embedded into the code stream. To implement inline caching, a method call instruction (alternatively called "message send" instruction) has an additional argument, which holds pointer to the cache object, allocated and associated with the method call instruction at compile time. The cache object holds the class pointer and a snapshot of a state counter (explained in next paragraph). During first execution, the receiver class and current value of the state counter are stored into the cache object. On subsequent executions the inline caching code checks whether the receiver is of the same class as in the first execution and if the saved state value matches the current value of the state counter. On successful check, it uses cached method pointer without further search, otherwise, if class is different or cache has been invalidated by change of the state value, it does a full lookup and updates values stored in the cache (**Fig. 4.3**).

```
def send(name, args, cache)
  receiver = args[0]
  if cache.class == receiver.class and cache.state == state
    method = cache.method
  else
    method = lookup(receiver.class, name)
    cache.state = state
    cache.class = receiver.class
    cache.method = method
  end
  method(args)
end
```

Figure 4.3. Inline caching

One important detail not described in the pseudo-code in Fig. 4.3 is the nature of the *state*. In current Ruby implementation, it is a global integer variable, which represents the state of the whole system. Global nature of the state counter makes invalidation very coarse-grained: any action that potentially can change dispatch of any method triggers increment of the global state counter, and thus invalidates all inline caches, as well as global method cache. Global method cache further affects performance, because it does not save a snapshot of global state counter to ensure validity, and for this reason, on mixin inclusion or other modification of global state, the whole method cache needs to be scanned in order to invalidate affected entries. With the size of the cache of 12 kb this may incur substantial overhead if the rate of global state changes is high.

Our target application, similar to example of the Fig. 4.1, exhibits very high overhead due to global state tracking. During application run, a monitoring mixin is repeatedly installed on each request. As a result, the application has very high rate of mixin inclusion and exclusion, each of which causes bump of the global state counter, and subsequent complete clearing of method cache. Flushing the method cache may take more than one quarter of the execution time, as can be seen in **Table 4.1**, the line *method cache*. The application is described in more detail in Section 4.5.

Table 4.1. Profile of the application benchmark on baseline ruby

item	runtime share
method cache	27.8 %
hash lookup	13.8 %
interpreter loop	13.2 %
method call	4.0 %
other	41.2 %

4.3 Fine-grained state tracking

To overcome the inefficiencies of global state, we devised a fine-grained state tracking technique. The goal is to factor the dependencies between inline and method caches and classes, so that modification of class methods or class hierarchy would require invalidation of fewer caches. Fine-grained state tracking replaces single global state counter with multiple state objects, each responsible only for the part of method dispatch space of classes and method names. Change in method definitions or class hierarchy propagate to associated state objects, which in turn invalidate dependent method cache entries and inline caches. Further, we propose several techniques to realize the benefit from the fine-grained state tracking.

4.3.1 Method lookup path-based state tracking

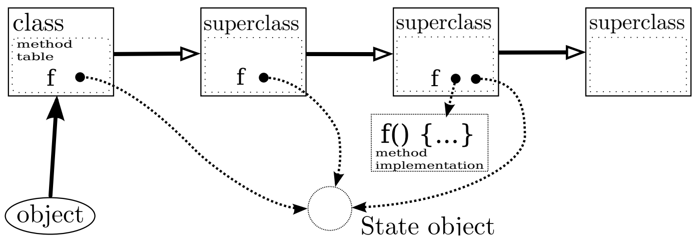


Figure 4.4. State object association with method lookup path

Our algorithm extends the standard method lookup procedure with state handling. We associate a state object with the method lookup path, which starts at the class provided as a parameter of *lookup* function, and continues on to its superclasses until the matching method implementation is found. Association is implemented by adding a method table entry to each class along the lookup path, and storing a pointer to the state object there (**Fig. 4.4**). A state object is allocated on the first lookup and reused on later lookups. A detailed definition of the *lookup* procedure is given in **Fig. 4.5**. For each method selector we add a method table entry to all classes that are accessed during the method lookup, and allocate a single state object. For some time, every distinct method implementation will have its own state object, but after dynamic mixin insertion, lookup of overridden method will find existing state object and reuse it for overriding method, so that overriding and overridden methods will get associated with the same state object. The algorithm guarantees that any set of methods that have been dispatched to with the same method selector and with the same type of receiver object will get the same state object. A special case when calls to overriding and overridden methods have caused creation of separate state objects and later turned out to be callable with the same receiver class is discussed at the end of this section. A property of reusing existing state objects gives us a bound on the total number of allocated state objects: it cannot be greater than total number of methods in a program.

State object is implemented as an integer counter. To make use of the state information, we change the inline caching code in the way shown in **Fig. 4.6**, and use the pstate pointer, returned by the *lookup* procedure. We modified method cache to store the last returned state object and a snapshot of state object counter. On method cache lookup, a validity check is performed, and if the check is successful, the cached method pointer and state object pointer are returned. In this way fine-grained state tracking is applied to both method cache and inline caches.

Using information collected during method lookups, we enforce an invariant: a state object must change its value on any change in object method tables or class hierarchy that might affect the outcome of lookup. To maintain an invariant, it is sufficient to do the following:

- On addition of a method to a method table we increase the corresponding state object counter, if there is a matching method entry.
- On mixin inclusion or exclusion, the method table of modified class contains precisely the list of methods, whose lookup may be affected by the change of delegation pointer. So we loop over method table and increase the counter in each state object.

A rare case when we encounter more than one existing state object during method lookup requires special handling. An example code that illustrates how this can happen in the setting of example of Fig. 4.1 is shown in **Fig. 4.7**. At the call s.process in the line 3 a new state object s_1 is allocated and associated with lookup path (Server,process), where first element of pair denotes the starting class, and the second — method selector. At later call in the line 8 another state object s_2 is allocated for path (NetworkServer,process), which starts at class NetworkServer and ends in the mixin ServerMonitor, where the method is found. The mixin ServerMonitor is removed in lines 9–11. The method lookup (NetworkServer,process) in the line 12 starts in the class NetworkServer, and ends in the class Server, finding both s_2 and s_1. In this case the call site will use the state object s_1 associated with (Server,process), because lookup algorithm gives precedence to a state object found later, i.e. higher in the class hierarchy (line 13 in Fig. 4.5). The state object s_2, earlier recorded at (NetworkServer,process) will be overwritten by s_1. Since existing call sites may still hold pointers to overwritten state object s_2, we add a dependency link $s_1 \to s_2$ between the two state objects (lines 10–11 in Fig. 4.5). Increment of a state object counter recursively triggers increment in all dependent state objects, which guarantees cache invalidation in all affected call sites. To assure correctness it is enough to do recursive invalidation only once, because subsequent execution of a call will do a full lookup and cache a pointer to the correct state object. One-time recursive invalidation is sufficient for correctness, however, it is not enough to ensure convergence to a single state object per call site. In this example, the link to the state object s_2 remains recorded in the mixin ServerMonitor and could reappear after next mixin inclusion. It is possible to prevent further use of s_2 by flagging it as "overridden" and recording a link to an "overriding" state object $s_2 \to s_1$. Similar treatment is necessary to ensure that polymorphic call sites use a single state object. We have not implemented this though.

4.3.2 Scheme of correctness proof

The properties of the invariant outlined in the previous subsection can be proved using following lemmas. We assume that the lookup procedure takes the pair $(class, methodname)$ as an input, and stores the lookup result — tuple $(stateobjectpointer, targetmethod, stateobjectsnapshot)$ in the inline cache.

We use the the term *lookup path* to denote the set of classes which has been traversed during lookup, so formally

$$lookuppath(class, methodname) = \{\text{set of classes referenced during lookup}\}$$

Lemma 1: *For any inline cache that has been initialized with a pointer to the state object, and for any pair $(class, methodname)$ in the inline cache contents, either the state object is referenced from the $lookuppath(class, methodname)$, or the inline cache is in an invalid state.*

Proof : First we note, that an inline cache is initialized with a pointer to the state object during lookup, which by definition of the lookup algorithm stores a pointer to the state object in all classes traversed during lookup, i.e. on the lookup path. This gives us the base of induction on the number lookup and dynamic mixin operations.

For the induction step we consider an inline cache and assume that the statement of the lemma holds. Then, if a lookup operation occurs, it may traverse the classes on our lookup path. By definition of the lookup algorithm, it either reuses the state object encountered during lookup (the same state object referenced from the inline cache), thus does not change the pointers on our given lookup path, or it may choose to use another state object, in which case the original state object is either invalidated or transitively linked from the overwritten state object, in either case preserving the condition.

If a dynamic mixin operation occurs anywhere on the given lookup path, then it changes the value of the state object and thus invalidates the contents of the inline cache. □

Lemma 2: *For any inline cache, it is either invalid, or its contents caches correct target of the lookup.*

Proof : Consider a call site with an inline cache. We prove the lemma using induction on the number of lookups and dynamic update operations, i.e. dynamic mixin installation or removal operations and executions of the call site.

Initial case. Inline caches are allocated and constructed in an invalid state after allocation, thus the condition is satisfied.

Induction step. First consider the execution of the call site. If the inline cache has been invalid before the execution of the call site, then during the execution the full lookup would be performed, which will update the pointers to the state object on the lookup path and update the inline cache contents with valid information. If the inline cache has been in valid state, then by the induction assumption it already had a correct target recorded.

If the inline cache has been valid, then its contents was correct according to the induction hypothesis, and the execution uses the correct cached target without updating it, thus maintaining the invariant.

In case when a dynamic mixin operation is performed, the inline cache is either already invalid, thus the condition holds, or it is correct. Then using lemma 1 we know that all classes on the lookup path of $(class, methodname)$ has a pointer to the state object. A dynamic mixin operation, that can affect the outcome of the lookup for our given inline cache must be performed somewhere on the lookup path. And since anywhere on the lookup path we have a pointer to the state object, the dynamic mixin operation will necessarily invalidate the cache by changing the value of the state object. Thus, after the dynamic mixin operation either the outcome of the lookup is not affected, and inline cache still contains correct value, or the inline cache is invalidated – which is exactly the induction assumption. □

The correctness of the inline caching with fine-grained state tracking follows from the lemma 2, because for any executed call site with an inline cache, it either will be in an invalid state, and thus a full lookup will be executed, or it will have a correct cached contents.

4.3.3 Polymorphic inline caching

To meet our goal of optimizing alternating changes (mixin inclusion and exclusion), it is necessary to cache information about multiple method dispatch destinations, so we implemented polymorphic inline caching [36]. We extend the inline cache object to include array of inline cache entries instead of a single method pointer and a state snapshot. Each inline cache entry includes a class pointer, a state value and a method pointer. The call site is associated with a single state object, but existence of multiple entries allows to cache multiple different methods, potentially belonging to different classes, each with its own snapshot of state object. The pseudo-code of method call operation using polymorphic inline caching (PIC) is shown in **Fig. 4.8**.

At the moment of allocation of a polymorphic inline cache, we allocate only a single entry, and further entry allocation happens as needed, after subsequent cache misses. To record a new entry in filled up polymorphic inline cache, we use random eviction policy, following the advice of the original inventors of polymorphic inline caching [36], which guarantees a constant overhead on cache miss independently of number of entries in cache.

4.3.4 Caching alternating states

The guarantees of fine-grained state tracking allow us to benefit from repeated behavior of our target application. Unchanged value of state object guarantees that associated method has not been changed or overridden by mixin. This invariant allows to implement caching optimization for temporary changes in class hierarchy. When installing a mixin object, we snapshot the state of method table of the class, where mixin is included. We record it in a *snapshot cache* of the modified class. A snapshot includes values of state objects for each method in the method table, before ("old") and after ("new") the superclass pointer change. Later on, when the mixin is removed, the snapshot cache is looked up for the corresponding snapshot, and the found snapshot is compared against current method table. If the state object value for a method matches the "new" state value recorded in the snapshot, it means that no other changes affected this method lookup, and thus we can safely rewind state object value to "old" value from the snapshot, instead of regular invalidation by increasing the state object counter. A separate cache is maintained for each class, and several alternating state snapshots are stored in a cache with LIFO eviction policy. In case several dynamic mixins override the same method, they will use the same state object. Since the cache contains pairs of state transitions, the caching technique is effective if the scopes of mixin inclusion are properly nested with respect to each other. Dynamic inclusions of mixins that are disjoint with respect to set of methods does not interfere (except for potential overflow of snapshot cache).

This technique relies on availability of polymorphic inline caching to realize the benefit. After mixin inclusion, calls of the overridden methods will miss in the inline caches, and so the new value of state object together with the overridden method pointer will be recorded in cache. However, there is high probability that a prior state and prior method pointer will be retained. Thus, when the mixin is later excluded, and the value of a state object is restored to prior value, this prior value still is present in polymorphic inline caches, and method lookup can be served from the cache. If the mixin is included again at the same place in class hierarchy, the system finds a matching snapshot in the snapshot cache and updates the state object values for all entries in the method table using the same algorithm as on a mixin removal, but with reversed pairs of state values: on mixin removal the state changes from "new" value to "old", and on mixin insertion from "old" to "new" (where "old" and "new" refers to the first mixin inclusion). This liberal reuse of state object values places some restriction on the state object invalidation, because simple counter increment by 1 does not guarantee that a new value has not been used in some snapshot. To avoid this problem, on invalidation of a state object we generate fresh values using global counter, so that the new value of a state object never coincides with earlier cached values.

In practice there exist a problem of wrapping around the maximum value of the counter, which can produce values that were already seen in the system. To prevent the incorrect operation in case when a global counter wraps around, it is necessary to walk over all inline caches in the system and clear their contents. We expect that counter wrap-around is a rare event and the overhead of clearing all the caches is acceptable.

4.3.5 Correctness of alternate caching

In order for the alternate cache to be useful, a number of conditions need be hold. For example, our polymorphic inline cache has only one pointer to the state object, and it is necessary to show that all of the methods that are called from a single call site will eventually use the same state object.

Lemma 3: *Given that the number of classes and methods in the system is bounded, any call site will eventually settle on a single state object.*

Proof : First, the number of allocated state objects is bounded by the number of distinct method in the system, because lookup algorithm only allocates a new state object if it has not found any on the lookup path.

Second, whenever lookup algorithm encounters more than one state object referenced from the inline cache and on the lookup path, the state object merge is initiated, so that one of the state objects will go out of use.

Lets consider a call site with an inline cache. After a first lookup, the inline cache receives a pointer to a state object. The call site can have the state object pointer changed only on call site executions, that missed in the inline cache and caused full method lookup. And whenever the lookup returns a state object different from the one originally referenced from the inline cache, that means that the former object has been put out of use, and so it cannot be returned by any subsequent lookup. Thus, lookup operation on a call site cannot infinitely return different state objects, because that would mean that the number of state objects out of use is monotonically growing, while this very same number is limited. □

The correctness of the alternate caching can be shown using the same approach as for the plain fine-grained state tracking plus the following lemma.

Lemma 4: *Each value of a state object can have at most one corresponding shape of the part of class hierarchy, which has pointers to the given state object.*

Proof : We construct the proof by induction on the number of dynamic updates. Induction base is trivial, because the class hierarchy has not been changed yet and thus only one shape exists.

Proving the induction step is trivial in the case of regular invalidation, since state object always obtains a fresh value, which has not existed in the system before. A case when state object gets an old value from alternate cache is more interesting.

Let's consider an entry in an alternate cache $(class, methodname)$, which has a pointer to a state object $state$, and assume that a dynamic mixin removal updates the state object $state$ from a value s_1 to an old value s_0. Alternate caching only returns the state object to an old value when its current value matches the recorded value

$$state = s_1$$

From the induction hypothesis follows, that the shape $lookuppath(class, methodname)$ is currently exactly the same as it was just after the mixin installation, which changed the state object value from s_0 to s_1. Thus the mixin removal operation will return the lookup path to exactly the same state as before mixin installation, and the induction hypothesis of correspondence of state object values to class hierarchy shape is preserved.

The case of mixin removal and installation pair is completely analogous.
□

Using the invariant from the lemma 4, it is easy to see that alternate caching correctly interacts with polymorphic caching, because any value of the state object uniquely determines the relevant class hierarchy shape (where relevant part of class hierarchy is a union of all lookup paths dependent on the state object), and thus the method dispatch target is also determined uniquely. Given the lookup target was correct when it was recorded in the inline cache for the first time, and it does not change, it is obvious that information stored in the inline cache remains correct.

4.4 Generalizations

4.4.1 Delegation object model

The techniques described in previous section can be generalized and applied to much wider range of systems than just Ruby. One particularly interesting model is delegation object model, which is used in Javascript and other prototype-based languages. Dynamic mixins in Ruby object model can be seen directly as delegate objects, because class-based method lookup algorithm conforms to that of delegation object model, and superclass calls have the same semantics as message resend. Of course, Ruby object model is limited by restrictions it places on dynamic delegation changes: delegation can be changed only by mixin inclusion or removal, and only for classes, but not for instance objects.

Despite of differences, algorithm for fine-grained state tracking remains valid for more general delegation object model without significant changes. A minor change is necessary in the algorithm of caching alternating states, because operation of delegation pointer change does not provide information on whether it is used for mixin installation, removal, or even entirely arbitrary change in delegation structure. We identify each change of delegation pointer as a pair of pointer values `(old,new)` and attach this identifier to the snapshot in the snapshot cache. On delegation pointer change, we check whether the snapshot cache has a matching pair on record, and apply it if found. The difference with mixins is that the check for matching pairs in cache needs to be done in both directions (old,new) and (new,old). If a match is not found, a new state change snapshot is recorded and inserted into the snapshot cache using LIFO eviction policy. Cache of alternating states is maintained separately for each delegating object, and is allocated only for objects, for which delegation is in fact changed during program execution.

We implemented general form of proposed techniques in C, using an object model similar to id [53], with behavior kept in separate objects, and delegation being possible between behavior objects. Behavior objects in this object model play the role similar to classes in Ruby object model. This implementation is further referred to as "C-impl". We implemented dynamic method dispatch functionality using macros, and inline cache is represented by local static variables. The implementation structure is very close to that Objective-C [19], and with some changes could be used as runtime library for Objective-C with dynamic extensions. We believe fine-grained state tracking can be applied to Javascript systems too.

4.4.2 Thread-local state caching

Application of the dynamic mixin as a base for implementing aspect-oriented or context-oriented constructs in a multi-thread system requires thread-local delegation. For example, layer activation in COP and thread-specific advice application requires that a mixin be applied in one thread, and not applied in other. Representation of *cflow* construct of AspectJ [42] requires even more complicated structure, where pointcut selection is represented by a mixin, which dynamically installs or removes advice mixin on entering and leaving pointcut shadow. The program execution state is different on different threads, and so has to be advice application. To resolve these issues, thread-specific delegation — an extension to delegation object model — has been proposed [33].

We noted that the scheme of caching alternating states can be extended to cover thread-specific delegation as well, by extending the state objects to hold thread-specific values. The algorithm described in Section 4.3.4 is extended in the following way

- State objects can have thread-local as well as global value. Global value of a state object includes a flag to indicate presence of thread-local values.
- Cache entries record the state value with cleared thread-local flag. If state object has its thread-local flag set, checks of global value against

inline cache entries will fail, leading to a separate "thread-local" branch in inline caching code.

- "Thread-local" branch reloads the thread-local value of the state object, which has the thread-local flag masked out.
- Thread-local mixin inclusion changes the thread-local value of affected state objects, either with a fresh value to invalidate all dependent caches, or with a value used in the past to switch to one of the cached states.

With this modifications in place, polymorphic inline caching and caching alternating states works as is. **Fig. 4.9** shows an example of a mixin insertion in thread T_1: the delegation graph is different when viewed from thread T_1 and other threads, and so is thread-local value of the associated state object. Accessing the thread-local value of state object is more costly than load of global value. Using the global value in the fast path and loading the thread-local value only after initial check failure removes the overhead of thread-local delegation in call sites where it is never used.

Thread-local delegation technique is not applicable to current Ruby, because Ruby does not allow multiple interpreter threads to be running simultaneously due to so called *global interpreter lock* arrangement [61]. As a result, multiprocessing in Ruby applications is typically implemented using multiple separate processes, and the issue of thread-local mixin installation does not occur.

4.5 Evaluation

It is hard to find a good application benchmark to evaluate the techniques proposed in this chapter, because dynamic mixin inclusion has not (yet) become a popular techniques. For this reason, to evaluate the techniques we use microbenchmarks, as well as a small application, which we specifically created to exercise dynamic mixin inclusion. We specifically look to establish the following:

- Proposed techniques in fact can reduce the inline cache misses on target application.
- Proposed techniques provide performance advantage on the target application, and reduce the overhead due to global method cache described in section 4.2.

- What impact on dynamic mixin performance proposed techniques make.
- What overhead proposed techniques have on regular method calls, when no dynamic mixins are involved.

We cover these questions in the reverse order. All experiments were performed on an Intel Core i7 860 running at 2.8 GHz with 8 Gb of memory, with Ubuntu Linux 9.10 installed with GNU libc 2.10.1 and gcc 4.4.1. Measurements were repeated at least 7 times, and an average value and sample standard deviation is shown.

4.5.1 Overhead on a method call

To evaluate the overhead of fine-grained state tracking and polymorphic inline caching can have on performance of a single call, we used a simple microbenchmark. Since techniques of caching alternating states only affects dynamic mixin inclusion, it is not evaluated by this microbenchmark. The microbenchmark executes a tight loop, and on each iteration calls an instance method on an object, which increments a global variable. We determined the cost of loop and counter increments by running the same loop without a method call, and subtracted this time from times measured in other benchmark runs, so that the table includes pure time for the method call and return. Measurements are presented in **Table 4.2**. Method cache hit implies prior inline cache miss, and the case of full lookup implies that method lookup missed both in inline cache and in method cache. The ratio column gives the ratio of measurements between modified Ruby versions and baseline Ruby, for each of the cases separately, with PIC hit ratio computed against baseline inline cache hit. Versions are written in the leftmost column sideways, *base* meaning baseline Ruby version, *fgst* — Ruby with fine-grained state tracking, *fgst+PIC* — version with both fine-grained state tracking and polymorphic inline caching, and *C-impl* — our implementation in C. C-impl does not have method cache, and has unoptimized implementation of method lookup. On Ruby, fine-grained state tracking overhead is barely visible in case of inline cache hit, as the difference of 1% is below measurement noise level. In case of lookup in method cache and full lookup, the overhead is more pronounced due to increased bookkeeping costs, 9% for the case of method cache hit and 16% for the case of full method lookup. Polymorphic inline caching incurs higher overhead, 49% for the case of monomorphic inline cache hit. After conversion of inline cache to polymorphic, overhead changes to 78%.

Overhead in cases of method cache hit and full lookup is even higher, up to almost 2 times. Despite high overhead of PIC, it still can be beneficial, as we show in further experiments. The inline cache hit and PIC hit numbers for *C-impl* provide peek into what level of method call performance would be possible without overheads of method call, such as arguments marshaling or boxing, which make method call in Ruby much more slow. To give a scale for absolute numbers, in the last table section we show the typical cost of static function call in C on the same hardware.

Table 4.2. Single call performance

		Case	single call time	ratio
Ruby	base	inline cache hit	33.5 ± 0.9 ns	100 %
		method cache hit	43.1 ± 0.9 ns	100 %
		full lookup	52.7 ± 0.5 ns[1]	100 %
	fgst	inline cache hit	33.9 ± 0.8 ns	101 %
		method cache hit	47 ± 1 ns	109 %
		full lookup	61 ± 1.5 ns[1]	116 %
	fgst+PIC	inline cache hit	49.8 ± 0.6 ns	149 %
		PIC hit	59.6 ± 0.8 ns	178 %
		method cache hit	78 ± 1 ns	181 %
		full lookup	105 ± 1 ns[1]	199 %
C-impl	global	inline cache hit	3.1 ± 0.1 ns	
		PIC hit	6.0 ± 0.1 ns	
		full lookup	71 ± 1 ns	
		C static call	< 2 ns	

4.5.2 Microbenchmarking dynamic mixin

To evaluate performance of dynamic mixin inclusion and exclusion, we use another microbenchmark. It is designed to measure effects in the extreme case, when frequency of mixin inclusion and exclusion is the same as frequency of method call. In this benchmark, two classes and a mixin object are arranged in structure shown in **Fig. 4.10**, with class A inheriting from class B. Even iteration of a microbenchmark inserts and odd iteration re-

[1]Cost of full method lookup obviously depends on class hierarchy. The number shown is for a single method defined directly in the object class, and so can be thought of as *minimal* cost of full lookup.

moves the mixin M between classes A and B, which is graphically depicted as a bold arrow between alternating values of superclass pointer of class A. Besides mixin insertion or removal, each iteration also calls an empty method f on an object of class A. To analyze the contribution of method call and mixin switch to microbenchmark run time, we separately report time for the loop, which includes only mixin switch (column *switch only*), and for the loop with both the method f call and mixin switch (column *switch+call*). Three cases of the benchmark exercise different dependencies of method dispatch on mixin.

- *below* — the method f is defined in class A, so that mixin M inclusion does not have any effect on dispatch of method A.f;
- *non-affect* — the method f is defined in class B, and mixin M does not override it, so mixin inclusion does not have effect on dispatch of method A.f, but it has the potential;
- *affect* — the method f is defined both in class B and in mixin M, so mixin inclusion overrides implementation in class B and causes different outcome of dispatch of method A.f.

Measurements of microbenchmark runs on Ruby are shown in **Table 4.3**, in the upper half. *Altern* refers to Ruby with all proposed techniques implemented, including fine-grained state tracking, polymorphic inline caches and caching alternating states, *fgst* denotes Ruby with just the fine-grained state tracking. *Base* is the baseline Ruby version, and *mc* is the baseline Ruby with method cache flush implemented through check against saved value of global state, rather than complete clearing of the hashtable. We included *mc* version in comparison, because as we discovered in table 4.1, the clearing of method cache constitutes large overhead, and it makes sense to question ourselves, which part of improvement is due to fixing this performance bug, and which part is due to advantages of fine-grained state tracking. Version with our techniques drastically outperforms baseline Ruby version, for several reasons. First, the overhead of clearing method cache on each mixin inclusion or exclusion has been eliminated, as can be seen by reduction of *switch+call* time from 3200 ns for baseline version to 750 ns for *mc* version. Second, fine-grained tracking further improves performance by 28% on average. The f method call is not the only call in the microbenchmark loop, because insertion and removal of the dynamic mixin is also performed by method calls, so improvement over *mc* version in both *switch* and *switch+call* times can be attributed to reduced inline cache misses. *Below* case for *fgst* version is visibly faster than *non-affect* and *affect* cases, because mixin inclusion does not flush inline cache at method f call site in *below* case. Third, with all of our techniques (*altern* line) method calls consistently hit in PIC, and in this particular case the cost of PIC hit is less than the cost of full method lookup in *fgst* version. This effect can be seen by comparing times over 500 ns of *fgst* version with times less than 500 ns in *altern* version: reduction by 12% on average. So despite of high per-call overhead of PIC in comparison with monomorphic inline cache hit, it still delivers benefit by reducing inline cache miss rate.

Measurements of the same microbenchmark on C-impl are shown in the lower half of the Table 4.3, with *global* showing results when object delegation is modified globally, and *thread* — with delegation modified using thread-local delegation, as described in section 4.4.2. In all cases, our techniques allow to run the microbenchmark with method calls consistently resulting in inline cache hits, as can be seen by low difference between measurements of loop with both switch and method call, and just a mixin switch. Thread-local caching support doubles the cost of mixin switch (about 40 ns vs. 19 ns) and more than doubles the cost of inline cache hit due to necessary use of PIC and thread-specific state values (about 10 ns vs. 4 ns), as can be seen from *non-affect* and *affect* cases of C-impl with thread-local caching. Note, that *below* case does not incur that cost on method call, because the method f dispatch is not affected by mixin inclusion, and is served by regular global inline caching mechanism.

4.5.3 Application benchmarking

To evaluate application-level impact of proposed techniques we developed a small application. We tried to imitate style typical for Ruby-on-Rails web application framework, using reflection for system configuration. Client resides in the same Ruby application, and no network connectivity is involved. The application structure is similar to the example in Fig. 4.1. To exercise dynamic mixins, it installs a monitoring mixin on each client request and removes it after request is processed. Monitoring mixin overrides processing method with another method that immediately calls superclass method and does nothing else. For benchmarking purposes, a client repeatedly executes a fixed scenario of requests to the server. We measure execution time of a fixed number of repetitions and report the average time per request.

Application benchmark is designed to stress the mix-in inclusion and exclusion as much as possible, to the extent that baseline ruby has 79% of inline

Table 4.3. Microbenchmark performance

		Case	switch + call	switch only
Ruby	base	below	3200 ± 30 ns	3100 ± 30 ns
		non-affect	3200 ± 30 ns	3100 ± 30 ns
		affect	3200 ± 30 ns	3100 ± 30 ns
	mc	below	750 ± 10 ns	640 ± 10 ns
		non-affect	750 ± 10 ns	640 ± 10 ns
		affect	750 ± 5 ns	640 ± 10 ns
	fgst	below	500 ± 3 ns	413 ± 6 ns
		non-affect	565 ± 6 ns	414 ± 5 ns
		affect	562 ± 9 ns	416 ± 5 ns
	altern	below	464 ± 5 ns	397 ± 4 ns
		non-affect	479 ± 8 ns	400 ± 4 ns
		affect	495 ± 4 ns	426 ± 3 ns
C-impl	global	below	23 ± 1 ns	19 ± 1 ns
		non-affect	23 ± 1 ns	19 ± 1 ns
		affect	23 ± 1 ns	19 ± 1 ns
	thread	below	42 ± 1 ns	39 ± 1 ns
		non-affect	49 ± 2 ns	40 ± 2 ns
		affect	51 ± 4 ns	41 ± 3 ns

cache misses on this benchmark. Using cache of alternating states prevents inline cache misses during steady state, as can be seen in bottom graph in **Fig. 4.11**. The graphs depict the number of inline cache hits, method cache hits and full lookups during the first five iterations of the application benchmark. The x axis represents the time measured in number of calls, and the y axis represents the percentages of outcomes, aggregated by 50 calls. From 0 to about 200 on the x axis, the system is in initialization phase. From that moment, the system enters steady state. Baseline Ruby version has rate of full lookups oscillating between 60 and 80% (the upper line in the upper graph), but with our techniques implemented, the majority of calls result in PIC hits (the upper line in the below graph). We report (monomorphic) inline cache hits and PIC hits separately, because in our implementation both monomorphic caches and PICs can coexist, and the cost of PIC hit is higher than the cost of hit in monomorphic inline cache. The results shown were measured with with the version of Ruby that unconditionally used polymorphic inline caches everywhere, that is why PIC hits dominate the lower graph in Fig. 4.11. We also tried a version where all inline caches are initially monomorphic, and are converted to polymorphic on first real inline cache miss with little difference in results, only the overhead of PICs was less visible.

Caching alternating states completely eliminates inline cache misses in steady-state phase, however some low but non-zero number of method cache hits and full lookups remain. We investigated this in detail, and found out that method cache hits and full lookups were caused by superclass calls and use of Ruby reflection API method respond_to? (question mark is a part of method name), which is used to find out if an object has an implementation of a method. Both superclass method call and implementation of respond_to? starts the method lookup from the method cache, which is occasionally invalidated by mixin inclusion or exclusion operations. As our techniques of caching alternating states is effective only for inline caches, it does not eliminate method cache and full lookups originating from places other than method call instruction. Application of our inline caching techniques to superclass calls is straightforward, though we have not implemented it yet. We defer to the future the question of whether it is worthy and how to apply caching alternating states to reflection API methods.

Table 4.4. Application benchmark results

version	time	ratio
baseline	20.7 ± 0.3 ms	100%
mc	14.5 ± 0.1 ms	70%
fgst	12.1 ± 0.2 ms	58%
PIC + fgst	12.5 ± 0.1 ms	60%
altern + PIC + fgst	10.7 ± 0.2 ms	52%

We evaluate the impact of the proposed techniques by measuring the request time on our application benchmark. The results are shown in **Table 4.4**. Eliminating clearing of method cache (by storing a state value in each cache line and checking it on cache lookup, so that individual lines can be invalidated just by change of global state value) improves the application performance by 30% (*mc* line). Fine-grained state tracking further improves performance by 12 percent points (*fgst* line). Introducing polymorphic inline caches, as we have seen above, incurs overhead on method calls, and this can be seen by 2 percent point increase in request time (the line marked as *PIC+fgst* in the table). However, caching of alternating states (line *altern+PIC+fgst*) improves application performance by 8 percent points. To verify that the overhead of flushing global method cache has been eliminated, we conducted the same profiling as that of Table 4.1 with our version of Ruby. While method cache management constituted about 27% of execution time in baseline Ruby version due to excessive flushing (line *method cache* in Table 4.1), our modifications reduced it to 0.2% (**Table 4.5**).

Table 4.5. Profile of the benchmark on modified Ruby version

item	runtime share
method cache	0.2 %
hash lookup	4.6 %
interpreter loop	21.1 %
method call	6.9 %
other	67.2 %

Regarding memory footprint of the proposed techniques, we can obtain some upper bounds from the definition of caching algorithm. The number of allocated state objects for fine-grained state tracking is bounded by the total number of methods in system. Number of polymorphic inline cache objects is bounded by the total number of call sites in the system. These bounds can be quite high in case of larger systems, and some heuristics may be needed to limit allocation. We leave detailed evaluation of memory use with large applications for the future work, however, to give a glimpse of memory usage of our implementation in the **Table 4.6** we show the number of minor page faults during benchmark application run, measured with Unix `time` utility. The page size is 4 kb.

Table 4.6. Memory usage approximation

version	number of minor page faults
baseline	12410 ± 10
fgst	12450 ± 10
PIC+fgst	12460 ± 10
altern+PIC+fgst	9830 ± 30

4.6 Related work

Fine-grained dependency tracking has been proposed and evaluated in the context of dynamic compilation systems with the main focus on reducing amount of recompilation. Self system [14] maintains dependency graph between compiled methods and slots, on which the compiled method depends, triggering recompilation of dependent methods on slot changes. Java virtual machines such as HotSpot [43] typically include some sort of the fine-grained state tracking functionality to minimize recompilation needed in case of dynamic class loading, but it appears that not much detail is available in published articles. Chambers et al. [15] proposed an elaborated system for managing complex dependency graphs, including several optimizations, such as introduction of filter nodes and factoring nodes. They do not consider a possibility of repeated change between alternating states. Since the typical cost of invalidation in their system is recompilation, they consider recomputing method lookup a viable choice for preventing a potentially much more costly method recompilation. On the other hand, we apply fine-grained state tracking to a Ruby interpreter with the purpose of preventing additional method lookups. Our system also can prevent method lookups in the case of system alternating between several states by repeated mixin inclusion and exclusion.

A similar scheme of tracking state and inline cache invalidation was implemented for Objective-C by Chisnall [18]. It proposes fixed association of state objects with methods. Our scheme generalizes this notion by associating state object with method lookup path, rather than with just the result of method lookup, and as a result allows for caching of multiple method entries, for distinct receiver types and multiple system states. The same author also proposed dynamic extensions to Objective-C in the form of *mixins* and *traits* [17], to which our techniques can be applied.

The general approach of using counters for selective invalidation has been widely studied in the field of distributed cache coherence [68], for example, in [65] the "one-time identifiers" are proposed for selective cache invalidation and [50] proposes using of the timestamps for the same purpose. One-time identifier is closer to the scheme used by our technique, since it relies on matching identifiers, rather than on monotonically growing timestamp, which allows also less-than comparison. A global counter had been introduced in Ruby implementation by Sasada and Maeda in 2005.

Polymorphic inline caching has been first proposed in Self [14], with the purpose of optimizing so-called polymorphic call sites, which dispatch on objects of several different types. In Java world, polymorphic inline caching has been successfully used for optimizing interface calls [32]. In our work, polymorphic inline caching is extended to cover not only distinct classes, but also distinct states of the same class, potentially containing multiple entries for the same class of receiver, but with different targets.

4.7 Summary

We proposed a specific way of fine-grained state tracking for highly dynamic languages, that allow changing of class hierarchy by using dynamic mixins, or even arbitrary delegation. Using the polymorphic inline caches and caching of alternating state, we have been able to significantly reduce rate of inline cache misses when the application repeatedly includes and excludes a dynamic mixin. On a small dynamic mixin-heavy Ruby application, our techniques eliminated the overhead of the global method cache (30%), and provided additional improvements in performance: fine-grained state tracking (17%), caching alternating states (12%). Due to high cost of method call in Ruby and low difference between performance of inline cache hit and miss, the benefits of proposed techniques are limited to applications with high rate of dynamic mixin inclusion, however, numbers for C-impl suggest that benefit would be higher on systems with more streamlined method call.

As inline caching has found use in compiled systems in the form of speculative method inlining [22], we expect our techniques to be applicable and beneficial with PIC objects used to create multi-version compiled code in dynamic compilation system, and we show that in the next chapter.

```
def lookup (klass, name, pstate)
  cur = klass
  while cur do
    entry = cur.method_table[name]
    if !entry
      cur.method_table[name] = Entry.new
    else
      if entry.pstate
        nst = entry.pstate.override || entry.pstate
        if pstate and pstate != nst
          nst.add_onetime_dependent(pstate)
          pstate.override = nst
        end
        pstate = entry.pstate
      end
      break if entry.method            # method found
    end
    cur = cur.super
  end
  return (nil, nil) if !cur
  pstate = State.new if !pstate
  cur = klass
  while cur
    entry = cur.method_table[name]
    entry.pstate = pstate
    break if entry.method
    cur = cur.super
  end
  return (entry.method, pstate)
end
```

Figure 4.5. *lookup* procedure with state tracking

```
def send(name, args, cache)
  receiver = args[0]
  if cache.class == receiver.class and
      cache.state == cache.pstate.value
    method = cache.method
  else
    method, pstate=lookup(receiver.class,name,cache.pstate)
    cache.class = receiver.class
    cache.method = method
    cache.pstate = pstate
    cache.state = pstate.value
  end
  method(args)
end
```

Figure 4.6. Inline caching with fine-grained state tracking

```
# include lines 1–15 from Fig. 1
s = Server.new
s.process                    #1
NetworkServer.class_eval do
  include ServerMonitor
end
n = NetworkServer.new
n.process                    #2
NetworkServer.class_eval do
  exclude ServerMonitor
end
n.process                    #3
```

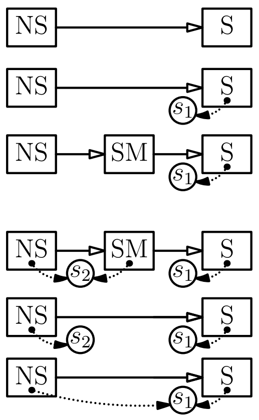


Figure 4.7. Example of situation when state objects need to be merged

```
def send(name, args, cache)
  receiver = args[0]
  if cache.class == receiver.class and
      cache.state == cache.pstate.value
    method = cache.method
  else
    if cache is polymorphic
      for entry in cache.array
        if entry.class == receiver.class and
            entry.state == cache.pstate.value
          method = entry.method
          break
      end
    else
      # convert cache to polymorphic
    end
    if method not found
      # lookup method and store result in cache
  end
  method(args)
end
```

Figure 4.8. Polymorphic inline caching

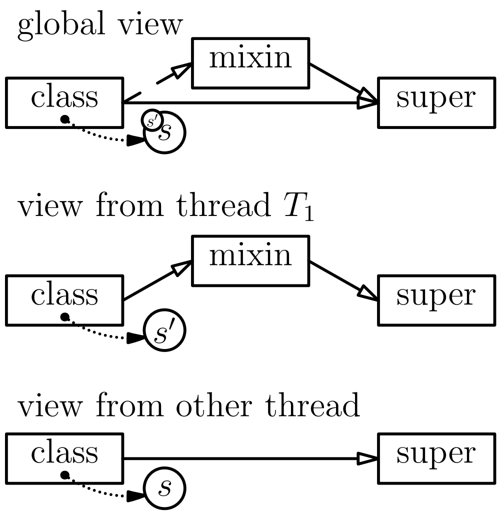


Figure 4.9. Thread-local state tracking

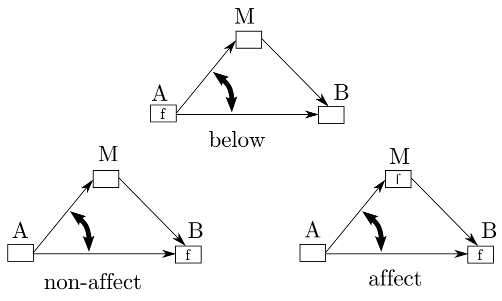


Figure 4.10. Three cases of a microbenchmark

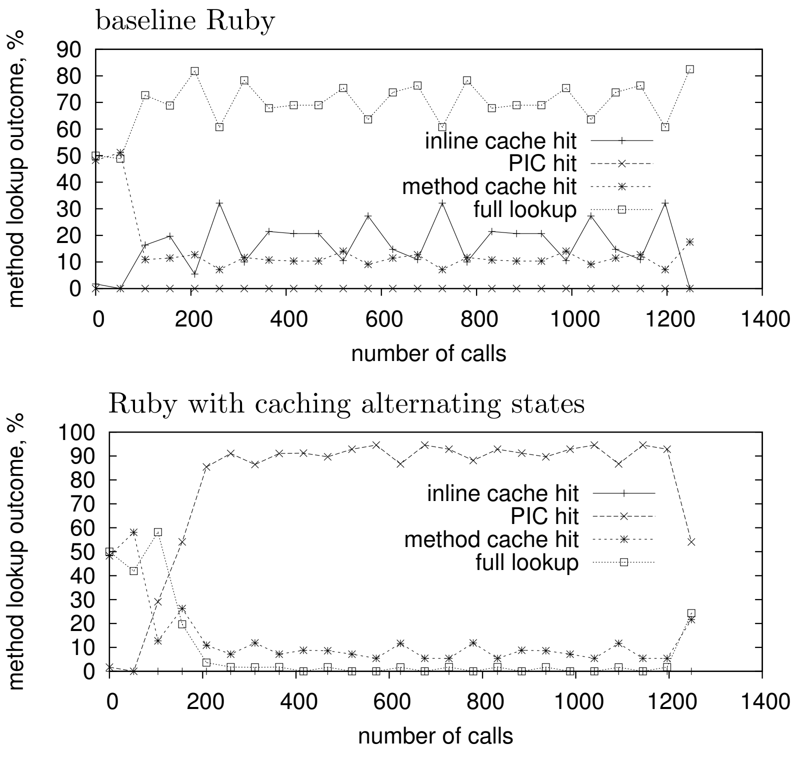


Figure 4.11. Method lookup outcome profile for the application benchmark

Chapter 5

Evaluation of dynamic mixin optimization in a compiled system

5.1 Introduction

Dynamic languages such as Ruby, Python and Javascript enjoy increasing popularity. Advanced optimization techniques, including just-in-time compilation and trace compilation are increasingly used in dynamic language implementation with good results. However, the potential of dynamic languages is not yet fully tapped. A particular technique of our interest is dynamic delegation, and its limited form, dynamic mixin. As other researchers has shown, dynamic mixin can be used as a basis for implementation of substantial subset of aspect-oriented programming [33] and context-oriented programming [63]. Dynamic mixin can be used on its own as well, for example, for run-time reconfiguration and adaptive monitoring. It is also useful for software evolution, because it makes application of some design patterns less intrusive. For example, dynamic mixin allows to use delegation pattern in a situation where normally one would use strategy pattern that requires extensive upfront planning.

In the chapter 4 we proposed a caching optimization for dynamic method dispatch, which takes dynamic mixin into account. We implemented the technique in the mainline Ruby interpreter (version 1.9.1) by modifying the inline cache handling code and evaluated its performance. However, we noted that high cost of the method dispatch in the Ruby interpreter make inline cache hit about 63% as expensive as inline cache miss, and the benefits of increasing inline cache hit ratio are small. We speculated further, that the technique can be used with much higher benefits in an environment with dynamic compilation. In this chapter we set out to experimentally verify that claim. To achieve maximal flexibility and control over code inlining optimization in the compiler, we chose to implement a dynamic compilation system from scratch. To expedite the development process, we use LLVM [45] as the back-end system, which facilitates development, gives a clearly defined target of compilation, and provides with ready-to-use low-level code optimizer. As the source language of our system, we chose the language IO [21]. It satisfies requirements of our research as a target language due to the following properties:

- IO is a purely object-oriented prototype-based dynamically typed language with multiple inheritance.
- IO object model is highly unified, having object slots and delegation as core concepts. Global constants are stored as slots in a context object, local variables as slots in an activation object.
- Method source code is available for introspection and modification in the form of abstract syntax tree (AST).
- IO has minimal, but highly readable syntax.

Since many of the challenges of IO implementation are typical for other dynamic languages as well, we believe that our results are not limited to the IO language, but are applicable to dynamic languages in general.

The contribution of this chapter is experimental evaluation of dynamic mixin optimization technique on a dynamic compilation system. We implemented the technique and showed that it allows to efficiently cache dynamic dispatch targets even in presence of dynamic mixin operations.

5.2 Delegation and dynamic mixin

Prototype object model is highly expressive and capable of supporting wide range of programming paradigms: popular static class hierarchy-based object models, aspect-oriented programming, and context-oriented programming. This power is based on the ability to modify delegation pointer of an object during program run time. Dynamic mixin is a particular way to modify delegation: object hierarchy is temporarily modified by inserting a mixin object into delegate chain.

In **Fig. 5.1** the example of a hypothetical application server is shown. The process method of the Server object examines the request before possibly changing delegate pointer to include additional security checks. The advantage of such use of dynamic mixin is the full availability of the BaseServer methods to the code in the mixin AdditionalSecurity. The mixin may override some of methods, and can delegate back to the original methods using resend(). In IO syntax every word is a message send, messages are separated

```
BaseServer := Object clone do(
    process := method(request, response,
        ...       // process the request
    ))
AdditionalSecurity := BaseServer clone do(
    process := method(request, response,
        ...       // do some checks
        resend(request, response)
        ...       // do more checks
    ))
Server := BaseServer clone do(
    process := method(request, response,
        if (request isSensitive,
            // install mixin
            self proto := AdditionalSecurity)
        resend(request, response)
        // remove mixin
        self proto := BaseServer
    ))
```

Figure 5.1. Application server example

by spaces, and arguments are optionally specified in parentheses. The result of one message send is the receiver for the next message, and for the first message the implicit receiver is an activation object. method() is a message to define a new method, var := expr is implicitly translated to setSlot("var", expr) before execution. The message setSlot() sets a slot in a receiver object. do() temporarily switches the context to define slots in objects other than top-level context object. proto assignment dynamically modifies delegation pointer. Arguments to methods can be passed in unevaluated in the form of abstract syntax tree, and can be evaluated on demand, arbitrary number of times and in arbitrary context, which allows expressing control flow constructs and new method definitions as regular method calls.

5.3 Dynamic mixin optimization

Dynamic method dispatch has been attracting attention of researchers for a long time, so a number of optimization techniques has been proposed: inline caching, guarded devirtualization, speculative method inlining. Application of any of these techniques to optimize method calls brings an important issue of invalidation: in the case when subsequent dynamic code redefinition occurs or object hierarchy changes it may be necessary to reconsider the optimized block of code and invalidate cached value or code. Many current implementations of dynamic languages make an assumption that object hierarchy does not change frequently, and leave it as an expensive operation, so programs using dynamic mixin exhibit low performance. However, typical use of dynamic mixin exhibits enough regularity so as to enable efficient optimization.

In the chapter 4 we proposed a state tracking mechanism based on method lookup. Dependency of inline caches on object changes is organized through state objects, which are implemented as integer counters. When the result of method lookup is cached, a snapshot of the counter is also stored, and is checked on each use. State objects are allocated dynamically and associated with sets of polymorphic methods that are called from the same call site. A pointer to the state object is installed in the method table in each object traversed during method lookup. On any modification to the method table or to the delegation pointer that can affect the outcome of the method dispatch, the counter in a state object is incremented. In this way we can maintain an invariant: whenever a target of dynamic dispatch changes, so does the associated state object.

The invariant on state objects allows to cache the state on mixin insertion and later rewind the system to the prior state on mixin removal. On mixin insertion, we record the old and updated values of state objects in the *alternate cache* associated with object, to which mixin is installed (**Fig. 5.2**). On

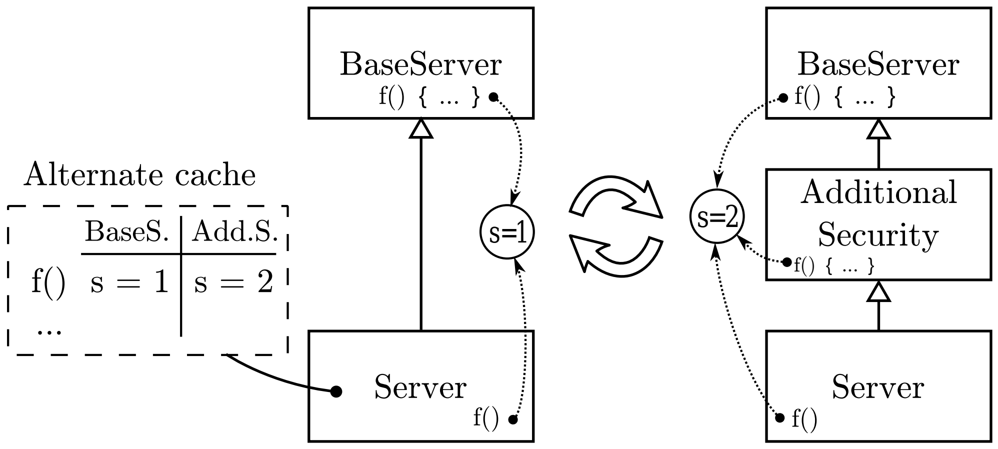


Figure 5.2. Alternate caching

mixin removal we walk over the method table and check if any methods has been invalidated by comparing the "updated" state value in the cache with current value. If there were no interfering invalidations, we can be sure that removal of the mixin brings the system to exactly the same state that it was in before mixin installation, and so we can restore the "old" values of state object from the alternate cache. Call sites that see several alternating targets of the dispatch can use cache with multiple entries, similar to polymorphic inline caching (PIC), so that all of the dispatch targets are served from cache. This techniques is applicable to dynamic mixin, and can be generalized to cover arbitrary delegation pointer changes. In this chapter we set out to explore if we can benefit from this caching scheme in the dynamic compilation system, by compiling several versions of the called methods according to the alternating states of the system, as shown in **Fig. 5.3**.

5.4 Approach to compilation

We implemented a minimal interpreter of IO language in Objective-C, and then proceeded to write a dynamic compiler, using mix of interpreted IO code and native Objective-C++ code for interfacing with LLVM infrastructure.

Since everything in an IO program is a message send, using method-based compilation would produce very small units of compilation consisting entirely of method calls, with overhead of method call dominating the execution time. For this reason it is essential to use inlining to increase the size of a compilation unit. Inlining a method requires knowledge of the target

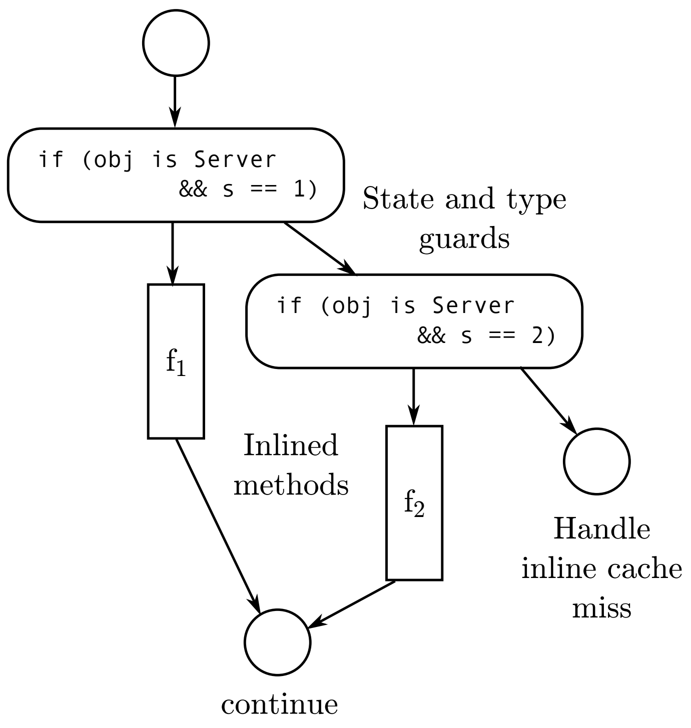


Figure 5.3. Compiled polymorphic inline cache

method, we resolve that by using the value cached during interpreted execution of the method. The cached method targets are likely to be the ones called during subsequent execution, but for the correctness it is necessary to insert a guard with a check of whether the type of the receiver and state matches the cached values. In case of guard failure, the execution falls back to an interpreter. When inline cache has multiple dispatch targets recorded, we generate code for each of the cached targets separately, effectively producing an inlined version of polymorphic inline cache. Code for control flow constructs such as while() and if(), and arithmetic operations are generated by compiler intrinsics.

Limitations

Our implementation is in its early stage of development, and has many limitations. Only a small subset of the IO standard library and a few methods necessary for compiler construction has been implemented. The compiler itself has even more limitations. Currently it is limited to compiling guarded method inlining and integer arithmetic without overflow checking. When a dynamic type guard fails, execution of the the whole method is restarted from the top under the interpreter, so this restricts compilation to methods without side effects.

Using the language under development for development itself has some benefits and drawbacks. A main benefit is the flexibility of the chosen implementation language. The drawbacks include absent error reporting, missing standard functionality and difficulties in localizing errors, because any misbehavior may be caused either by the bug in the developed code itself, or in the underlying interpreter.

5.5 Evaluation

We present the preliminary evaluation results of our prototype implementation through micro-benchmarking. The goals are to estimate the overhead of state checks that polymorphic inline caching with fine-grained state tracking introduces, and verify the performance of the dynamic method dispatch in presence of frequent dynamic mixin operations.

5.5.1 Overheads of state checks

The first experiment is to determine the cost of the state checks introduced by the polymorphic inline caching (PIC). Resulting graphs for two types of CPU are shown in the **Fig. 5.4**. The measurements are performed by the tight loop microbenchmark, which has a single polymorphic call in it. For the baseline of comparison we measured execution time of inlined call with type guard but without any state checks. The value shown on the graph is a number of CPU cycles necessary for the state checks introduced due to polymorphic inline caching, computed as difference with a baseline. The axis x represents the position of hitting entry in the inline cache with 4 entries in total. We executed several slightly different benchmarks, some with entries differing in object types, and some differing in value of the state object, i.e. corresponding to mixin in an installed or removed state. Gray dots on the graph represent individual run results.

To analyze the factors contributing to the state check overhead, we performed to hardware profiling. The factors are additional instruction execution and branch misprediction penalty. Branch misprediction penalty is also responsible for variation of the results within the group of benchmarks,

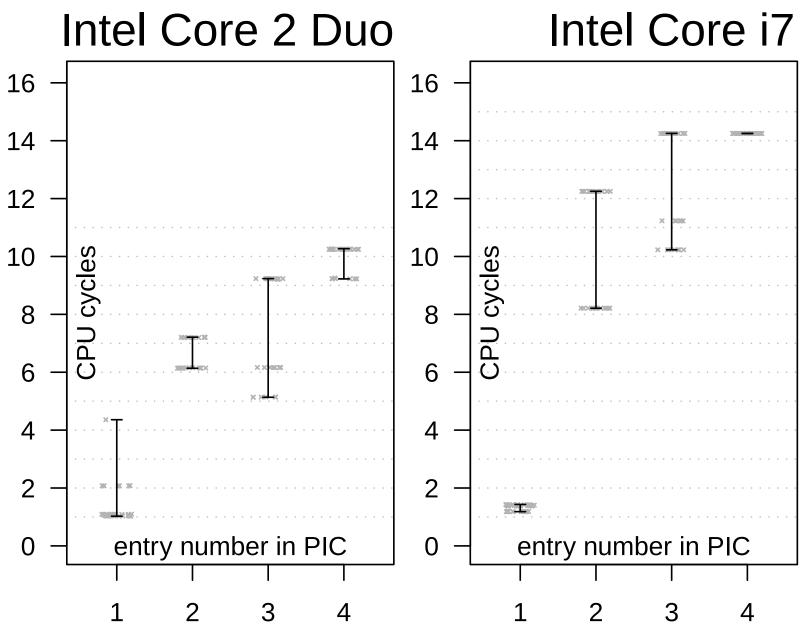


Figure 5.4. Cost of state checks in compiled polymorphic inline cache

where method dispatch hits in the same entry of the cache. There is interesting characteristic of the measurement results when there is just a single entry in the polymorphic inline cache: the overhead is typically as low as 1 CPU cycle, and so the use of state checks for call sites with a single target has very small overhead. Overall, the overhead of state checks in polymorphic inline cache is reasonable and is within fourteen instruction cycles per method call.

5.5.2 Dynamic mixin performance

To evaluate the performance of the dynamic mixin operations and their impact on the method call performance, we used the benchmark shown in the **Fig. 5.5**, which is the same microbenchmark as we used in section 4.5.2. This benchmark exercises both dynamic method dispatch and dynamic mixin operations, i.e. mixin installation and removal, in the same loop. A and B are two objects, and M is a mixin, which is repeatedly installed and removed. The microbenchmark consists of a loop which calls a method f on the object A. The two version being compared are *altern* — an implementation of alternate caching, and *nocache* — a version without inline caching, which does dynamic method lookup on each polymorphic call site (monomorphic

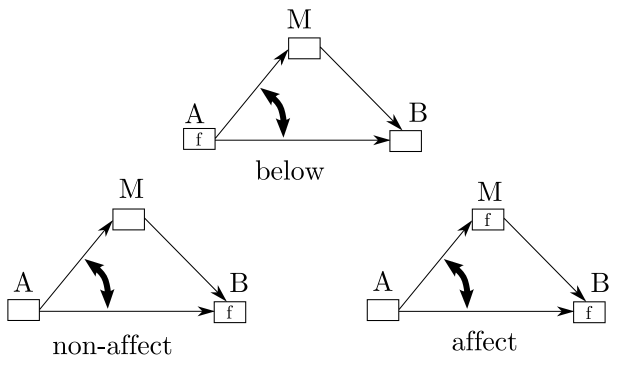


Figure 5.5. Mixin microbenchmark

call sites are handled by inlining in the same way as in *altern* version). Three different cases shown in the Fig. 5.5 have method f defined differently. In the *affect* case, there are two implementation of method f, and the dispatch is altered with dynamic mixin operations. In the *non-affect* case the only implementation of method f is defined in object B so that mixin potentially could affect method dispatch, but in fact does not. In the *below* case, the method f is defined in the object A, so that the outcome of the method lookup does not depend on whether mixin is installed or not. This benchmark is useful to discern caching efficiency and invalidation granularity of a language processor.

Table 5.1. Mixin microbenchmark execution time, mixin switch on every iteration

Version	Case	Switch+call	No call
nocache	below	2880±60 ns	2860±50 ns
	nonaff	3290±60 ns	3000±90 ns
	affect	3260±70 ns	2990±80 ns
altern	below	3500±60 ns	3550±100 ns
	nonaff	3490±60 ns	3490±70 ns
	affect	3510±60 ns	3520±80 ns

Table 5.1 shows the measurements expressed as iteration time with sample standard deviation in ns. *Switch+call* column shows the iteration time of the bench where mixin operation (installation or removal) is done on every iteration of the loop, and *no call* shows the time of a modified benchmark,

where method call is substituted by equivalent code without method call. The only clearly visible difference in the results table is the difference between *altern* and *nocache* versions: the microbenchmark loop in nocache case takes around 3 μs, but *altern* version takes about 3.5 μs. The reason for increase is high cost of the dynamic mixin operation, which has not been tuned in our implementation. The noise almost completely hides the cost of the dynamic method call.

Table 5.2. Rerun of mixin microbenchmark, mixin switch on every 100th iteration

Version	Case	Switch+call	No call
nocache	below	53.6±1.2 ns	42.1±0.8 ns
	nonaff	263±6 ns	43.4±1.2 ns
	affect	213±6 ns	43.0±0.7 ns
altern	below	38.8±0.8 ns	37.4±1.2 ns
	nonaff	38.6±0.5 ns	36.9±0.5 ns
	affect	38.5±0.6 ns	37.1±0.6 ns

To see the effect on the individual call performance better, we modified the benchmark to call dynamic mixin operation only on each 100th loop iteration. The results are shown in **Table 5.2**. The difference between values in columns *switch+call* and *no call* gives us a cost of the single dynamic method dispatch. This time the difference between columns can be clearly seen, and while in *nocache* version dynamic call takes up to 220 ns, alternate caching with multi-version code (*altern* version) has the cost of dynamic method dispatch less than 2 ns. This clearly shows the effectiveness of optimization for dynamic method dispatch. Variation in *switch+call* time for *nocache* version is due to varying costs of dynamic method lookup.

Overall, this experiment shows two things. First, the performance of the dynamic method dispatch is drastically improved by caching, compared to non-cached dynamic lookup, and alternate caching keeps caching effective even in presence of dynamic mixin operations. Second, slow implementation of dynamic mixin switch operation is a major problem of our implementation. We are considering tuning the implementation and adding adaptive choice of state tracking granularity as a possible way to improve performance.

5.6 Related work

The problem of compilation of dynamic languages has been thoroughly researched to date. Smalltalk [23] was first to use just-in-time compilation for a dynamic language. Self [14] is the project that pioneered many of the commonly used techniques, including polymorphic inline caches [36] and type-split compilation. It also had an elaborated system of fine-grained state tracking for selective recompilation of methods [15]. Psyco [57] implemented a specific form of compilation named *just-in-time specialization*, which generates code as the program executes, and generates new branches of code on type guard failures. Trace-based compilation [29, 74] similarly compiles along a particular program execution path (*trace*), but it does compilation in a bulk after the trace collection is complete. A number of dynamic languages have tracing compiler implementations: Python [10], Javascript [16, 31]. PyPy [59] is notable for its use of a statically typable language RPython [5], which is a proper subset of full dynamic Python language. An interesting feature of RPython is the bootstrapping phase, that allows use of advanced dynamic features, like extensible classes and generative programming. Portable approach to compilation by using compilation to C source code has been proposed for Lua [72] and PHP [9]. Inlining virtual methods in Java system has been studied in [22] and [69].

None of the related works tackled the problem of dynamic mixin optimization, though the approach of guarded inlining and trace-based compilation is somewhat similar and can be adapted to use fine-grained state tracking.

5.7 Summary

We devised and started an implementation of a dynamic compiler for IO language, intended as a research vehicle for optimization of dynamic languages, in particular inlining and inline caching. Our first target for evaluation is guarded multi-version compilation based on the method call profile collected in polymorphic inline cache. Due to the limitations of current compiler, we are evaluating the implementation with microbenchmarks, which include integer computations and method calls. Our experiments shown that the overhead of state checks in multi-version code does not exceed 14 processor cycles, and allows to prevent inline cache misses in the presence of dynamic mixin operations. The proposed optimizations are applicable to any dynamic language with support for dynamic mixin operations.

Completing of the compiler to completely cover the IO language and application-level benchmarking remain our future tasks.

Chapter 6

Conclusion

This thesis presented the treatment of the performance for the interpreted implementation of a dynamic language, making an emphasis on the dynamicity. Dynamicity is the characteristic of an implementation that describes the dependence of the performance on the rate of the dynamic changes in the system. Existing high-performance techniques of dynamic language implementation often have bad performance characteristics when the rate of dynamic changes is high. This necessitates further research into implementation techniques with high dynamicity.

We evaluate application of the superinstruction optimization to Ruby interpreter and show that traditional approach produces limited benefit on modern hardware. Then we propose a novel approach to superinstructions — arithmetic superinstructions. Arithmetic superinstructions produce greater benefit for numeric computation applications.

Further we consider application of dynamic mixin. Existing approach to inline caching in Ruby interpreter produces substantial overhead when dynamic mixin is used with high frequency. We propose a combination of three techniques: fine-grained state tracking, polymorphic inline caching and alternate caching, which allows to effectively capture the repeated pattern of dynamic mixin inclusion and prevent inline cache misses even when the frequency of dynamic mixin operations is high.

Our contributions are the study of application of superinstruction technique to Ruby interpreter and proposal of arithmetic superinstructions, and proposal of optimization techniques for dynamic mixin.

6.1 Superinstructions for Ruby interpreter

We describe application of the superinstruction technique to Ruby interpreter. It works by merging the subsequent interpreter operations into a single superinstruction without altering the semantics of the instructions. Due to this fact the dynamicity characteristics of the interpreter is not affected by the use of superinstructions.

Traditional approach to superinstructions is to eliminate the interpreter dispatch within the superinstructions. It reduces the number of indirect branch instructions necessary for program execution. In past, processors did not have good indirect branch predictors, and the majority of indirect branches, executed by direct-threaded interpreter resulted in processor pipeline stall and thus caused substantial overhead. In our analysis and experimental evaluation we have shown that traditional approach has reduced its efficiency with the progress of hardware, especially due to availability of better indirect jump predictors.

We proposed a different approach to superinstructions — optimization within the arithmetic superinstructions. Current Ruby interpreter has polymorphic operations, which use combination of tagging and boxing to denote the type of values. Floating point numbers are implemented using boxing, which causes significant costs of allocation and garbage collection. Using superinstruction technique to merge arithmetic superinstructions allows to reduce one boxing operation for the intermediate computation result, and thus reduces overall allocation. Reduction in allocation in turn reduces the rate of garbage collection and thus reduces the execution time, improving computation application performance.

In our experiments, the performance of numeric benchmarks has been improved by 0–23%, without negative impact on the non-numeric benchmarks.

6.2 Dynamic mixin optimization

Dynamic use of mixin is a typical feature of modern popular dynamic languages. Its use is useful for implementation of techniques such as dynamic patching or dynamic monitoring, and also can be used as the base for implementation of substantial parts of context-oriented programming paradigm or aspect-oriented programming.

Frequent use of dynamic mixin operations poses a challenge to current implementation techniques and causes high overheads. Current implementation techniques make an assumption that the target of method calls mostly remains stable, but dynamic mixin is expressly used to override some methods by implementation provided by the mixin. Current Ruby interpreter also suffers from coarse granularity of invalidation due to global nature of the state tracking. Thus any dynamic mixin operation causes flush of method cache and all inline caches, causing multiple inline cache misses.

To optimize the operation of inline caches in presence of dynamic mixin operations, we propose combination of three techniques. Fine-grained state tracking introduces use of multiple state objects, which are referenced from method tables, and are allocated and updated during method lookups or dynamic mixin operations. We provide the description of the algorithm and outline the proof of the correctness of our algorithm operation.

We propose a novel technique to capture repeated patterns of dynamic mixin operations — *alternate caching.* Alternate caching works by storing the pairs of state transitions for the individual method table entries, identified by dynamic mixin operation. On a reverse operation, the state object is returned to an older state instead of invalidating by assigning a fresh value. The correctness of the reuse of older state value is guaranteed by the invariants, maintained by the fine-grained state tracking algorithm. The pair-wise organization of alternate cache allows to account for multiple dynamic mixin operations on the same class, provided that dynamic mixin installation and removal scopes are properly nested. Dynamic mixins applied to the non-related parts of the class hierarchy are completely independent.

Well-known technique of polymorphic inline caching is used to allow caching of multiple targets of the call site. In our implementation we extended polymorphic inline cache to store multiple values of the state object value, effectively allowing to cache multiple dispatch target for the same receiver object class. In situation when dynamic mixin operation is repeatedly used to temporarily override a method, random eviction strategy of polymorphic inline cache with high probability will eventually record all targets of the method, serving the subsequent calls entirely from the cache.

We implemented the dynamic mixin optimization for the Ruby interpreter and shown that combination of techniques can produce benefits up to 6 times on a microbenchmark, and up to 48% on a small application benchmark. The most of the speedup is provided by the fine-grained state tracking, polymorphic inline caching causes small overhead, which is somewhat improved by using of alternate caching.

The inline caching techniques for the dynamic mixin can be generalized to the general delegation object model. It can also be applied in the compiled system. We evaluate dynamic mixin optimization techniques in a small compiled system and show that the overhead of state checks is very low – just a few CPU cycles. The dynamic mixin operation itself has relatively high cost and is worth of improvement.

Bibliography

[1] V8 javascript engine. http://code.google.com/apis/v8/design.html, 2008.

[2] Tiobe programming community index for november 2010, Nov 2010. URL http://www.tiobe.com.

[3] O. Agesen. Constraint-based type inference and parametric polymorphism. *Static Analysis*, pages 78–100, 1994.

[4] O. Agesen and D. Detlefs. Mixed-mode bytecode execution. *Sun Microsystems, Inc. Mountain View, CA, USA*, 2000.

[5] D. Ancona, M. Ancona, A. Cuni, and N. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *Proceedings of the 2007 symposium on Dynamic languages*, page 64. ACM, 2007.

[6] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A survey of adaptive optimization in virtual machines. *Proceedings of the IEEE*, 93 (2):449–466, 2005. ISSN 0018-9219.

[7] J. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973. ISSN 0001-0782.

[8] M. Berndl, B. Vitale, M. Zaleski, and A. Brown. Context threading: A flexible and efficient dispatch technique for virtual machine interpreters. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 15–26. IEEE, 2005. ISBN 076952298X.

[9] P. Biggar, E. de Vries, and D. Gregg. A practical solution for scripting language compilers. In *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1916–1923, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-166-8.

[10] C. Bolz, A. Cuni, M. Fijalkowski, and A. Rigo. Tracing the meta-level: PyPy's tracing JIT compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[11] D. Bornstein. Dalvik VM internals. In *Google I/O Developer Conference*, 2008.

[12] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings OOPSLA/ECOOP '91, LNCS 512*, pages 303–311. ACM, 1990.

[13] S. Brunthaler. Efficient interpretation using quickening. In *Proceedings of the 6th symposium on Dynamic languages*, pages 1–14. ACM, 2010.

[14] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes. *ACM SIGPLAN Notices*, 24(10):49–70, 1989. ISSN 0362-1340.

[15] C. Chambers, J. Dean, and D. Grove. A framework for selective recompilation in the presence of complex intermodule dependencies. *Software Engineering, International Conference on*, 0:221, 1995.

[16] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. Tracing for web 3.0: trace compilation for the next generation web applications. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 71–80. ACM, 2009.

[17] D. Chisnall. Updating objective-c. Technical report, Swansea University, 2008.

[18] D. Chisnall. A modern objective-c runtime. *Journal of Object Technology*, 8(1):221–240, jan 2009.

[19] B. Cox and A. Novobilski. *Object-oriented programming: an evolutionary approach.* Addison-Wesley, 1986.

[20] B. Davis, A. Beatty, K. Casey, D. Gregg, and J. Waldron. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 41–49. ACM, 2003. ISBN 1581136552.

[21] S. Dekorte. Io: a small programming language. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, page 167. ACM, 2005.

[22] D. Detlefs and O. Agesen. Inlining of virtual methods. In *Proceedings ECOOP '99, LNCS 1628*, pages 258–277, 1999.

[23] L. Deutsch and A. Schiffman. Efficient implementation of the Smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, page 302. ACM, 1984. ISBN 0897911253.

[24] M. Ertl and D. Gregg. The structure and performance of efficient interpreters. *Journal of Instruction-Level Parallelism*, 5:1–25, 2003.

[25] M. Ertl, C. Thalinger, and A. Krall. Superinstructions and replication in the Cacao JVM interpreter. *Journal of .NET Technologies*, 4(1):31–38, 2006.

[26] M. A. Ertl. Stack caching for interpreters. In *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, pages 315–327. ACM, 1995. ISBN 0-89791-697-2.

[27] S. Fink and F. Qian. Design, implementation and evaluation of adaptive recompilation with on-stack replacement. In *Code Generation and Optimization, 2003. CGO 2003. International Symposium on*, pages 241–252. IEEE, 2003. ISBN 076951913X.

[28] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA '09: Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 283–300, New York, NY, USA, 2009.

[29] A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical Report ICS-TR-06-16, Donald Bren School of Information and Computer Science, University of California, Irvine, 2006.

[30] A. Gal, C. Probst, and M. Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. In *Proceedings of the 2nd international conference on Virtual execution environments*, pages 144–153. ACM, 2006.

[31] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, et al. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 465–478. ACM, 2009.

[32] N. Grcevski, A. Kielstra, K. Stoodley, M. Stoodley, and V. Sundaresan. Java just-in-time compiler and virtual machine improvements for server and middleware applications. In *3rd Virtual Machine Research and Technology Symposium (VM)*, 2004.

[33] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *Proceedings ECOOP '09, LNCS 4609*, pages 501–524. Springer Berlin / Heidelberg, 2007.

[34] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[35] U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336. ACM, 1994. ISBN 089791662X.

[36] U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP'91 European Conference on Object-Oriented Programming*, pages 21–38. Springer, 1991.

[37] J. Hoogerbrugge, L. Augusteijn, J. Trum, Rik, and R. V. D. WIEL. A code compression system based on pipelined interpreters. *Softw. Pract. Exper*, 29:11, 1999.

[38] D. Ingalls. The lively kernel: just for fun, let's take javascript seriously. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 9:1–9:1, New York, NY, USA, 2008. ACM.

[39] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 318–326. ACM, 1997. ISBN 0897919084.

[40] P. Kail. Forth programming language. *SOFTWARE WORLD.*, 16(3): 2–5, 1985.

[41] S. Kawai. Efficient floating-point number handling for dynamically typed scripting languages. In *Proceedings of the 2008 symposium on Dynamic languages*, page 6. ACM, 2008.

[42] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of aspectj. In *Proceedings ECOOP '01, LNCS 2072*, pages 327–354, 2001.

[43] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot^{TM}client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008.

[44] G. Krasner, editor. *Smalltalk-80: bits of history, words of advice.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[45] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, 2004.

[46] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *ACM Sigplan Notices*, 21(11):214–223, 1986.

[47] T. Lindholm and F. Yellin. *Java virtual machine specification.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999. ISBN 0201432943.

[48] R. Martin. *Agile software development: principles, patterns, and practices.* Prentice Hall PTR Upper Saddle River, NJ, USA, 2003. ISBN 0135974445.

[49] Y. Matsumoto and K. Ishituka. *Ruby programming language.* Addison Wesley Publishing Company, 2002. ISBN 020171096X.

[50] S. MIN and J. BAER. A timestamp-based cache coherence scheme. In *1989 International Conference on Parallel Processing, University Park, PA*, 1989.

[51] T. Owen and D. Watson. Reducing the cost of object boxing. In *Compiler Construction*, pages 2726–2729. Springer, 2004.

[52] I. Piumarta and F. Riccardi. Optimizing direct threaded code by selective inlining. In *Conference on Programming Language Design and Implementation: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation: Montreal, Quebec, Canada.* ACM, 1998.

[53] I. Piumarta and A. Warth. Open, extensible object models. *Self-Sustaining Systems*, 5146:1–30, 2008.

[54] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002.

[55] T. A. Proebsting. Optimizing an ansi c interpreter with superoperators. In *In Proc. Symp. on Principles of Programming Languages*, pages 322–332. ACM Press, 1995.

[56] G. B. Prokopski and C. Verbrugge. Analyzing the performance of code-copying virtual machines. In *OOPSLA*, pages 403–422, 2008. URL `http://www.prokopski.com/publications/CC2008-Prokopski.pdf`.

[57] A. Rigo. Representation-based just-in-time specialization and the psyco prototype for python. In *Proceedings of the 2004 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 15–26. ACM, 2004.

[58] A. Rigo and C. F. Bolz. How to not write virtual machines for dynamic languages. In *Proceedings of Dyla 2007*, 2007.

[59] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, page 953. ACM, 2006.

[60] K. Sasada. YARV: yet another RubyVM: innovating the ruby interpreter. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 158–159. ACM, 2005. ISBN 1595931937.

[61] K. Sasada. *Efficient implementation of Ruby virtual machine.* PhD thesis, The University of Tokyo, Graduate school of information science and technology, 2007. In japanese language.

[62] K. Sasada. A lightweight representation of floating-point number on ruby interpreter. In *Proceedings of the workshop of programming and programming languages (PPL2008)*, 2008.

[63] H. Schippers, M. Haupt, and R. Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1944–1951. ACM New York, NY, USA, 2009.

[64] Y. Shi, K. Casey, M. Ertl, and D. Gregg. Virtual machine showdown: stack versus registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, 4(4):1–36, 2008. ISSN 1544-3566.

[65] A. Smith. Cpu cache consistency with software support and using" one time identifiers. In *Proceedings of Pacific Computer Communications Symposium*, 1985.

[66] R. Stallman et al. *Using and porting the GNU compiler collection.* 1999. ISBN 1882114388.

[67] G. Steele Jr. Fast arithmetic in maclisp. *MIT AI Memo 421*, 1977.

[68] P. Stenstrom. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990. ISSN 0018-9162.

[69] T. Suganuma, T. Yasue, and T. Nakatani. An empirical study of method inlining for a java just-in-time compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium*, 2002.

[70] A. Villazón, W. Binder, D. Ansaloni, and P. Moret. Hotwave: creating adaptive tools with dynamic aspect-oriented programming in java. In *Proceedings GPCE '09: Proceedings of the eighth international conference on Generative programming and component engineering*, pages 95–98. ACM, 2009.

[71] J. Vitek and R. Horspool. Compact dispatch tables for dynamically typed object oriented languages. In *Compiler Construction '96, LNCS 1060*, pages 309–325. Springer, 1996.

[72] K. Williams, J. McCandless, and D. Gregg. Portable just-in-time specialization of dynamically typed scripting languages. pages 391–398, 2010.

[73] K. Williams, J. McCandless, and D. Gregg. Dynamic interpretation for dynamic scripting languages. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 278–287. ACM, 2010.

[74] M. Zaleski, A. Brown, and K. Stoodley. Yeti: a gradually extensible trace interpreter. In *Proceedings of the 3rd international conference on Virtual execution environments*, page 93. ACM, 2007.