日本ソフトウェア科学会第 27 回大会 (2010 年度) 講演論文集

Towards JIT compilation for IO language

ザキロフ サリフ　千葉 滋　柴山 悦哉

IO [6] is a relatively new pure object-oriented language, providing dynamic features comparable with popular scripting languages like Python or Ruby. IO has simple but flexible syntax, allowing for metaprogramming at the syntax tree level. Some aspects of the syntax, such as operator and assignment reshuffling, make IO source code feel natural. Availability of runtime code modification and simple grammar makes IO a good choice as a dynamic programming language research vehicle.

In this work we describe our approach to dynamic compilation of programs in IO. Inline caching and speculative inlining provide high performance benefits if the program behavior is stable. We explore programs with multiple or alternating behaviors, and propose multi-version speculative inlining with fine-grained invalidation.

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 [10] and context-oriented programming [14].

In our previous work [16] we proposed a caching optimization for dynamic method dispatch, which takes dynamic mixin into account. We evaluated the performance of the technique implementation it in the mainline Ruby interpreter (version 1.9.1) by modifying the inline cache handling code. However, we noted that high cost of the method dispatch in the 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 work we set out to experimentally verify that claim. More long-term goals of this project include testing a hypothesis whether it is possible and practical to use unrestricted dynamic language as a base system language, and implement other languages on top of it.

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 [11] as the back-end system, which fa-

Salikh Zakirov, 東京工業大学, Tokyo Institute of Technology, Dept. of Mathematical and Computing Sciences.

Shigeru Chiba, 東京工業大学, Tokyo Institute of Technology, Dept. of Mathematical and Computing Sciences.

Etsuya Shibayama, 東京大学, University of Tokyo, Information Technology Center.

cilitates 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 [6]. It satisfies requirements as a target language of our research 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.
- Methods source code is available for introspection and modification in the form of abstract syntax tree (AST).
- IO has minimal, but highly readable syntax.

The above leads us to believe that challenge of compiling IO includes all of the challenges applicable to the mainstream dynamic languages. However, the highly unified concept world of IO object model gives hope on unified handling of issues traditionally handled separately, such as namespaces, dynamic dispatch and code modification. The generality of IO language also gives us a hope, that IO-specific optimization can be reused directly by implementing other languages on top of the IO object model.

2 Delegation and dynamic mixin

Prototype object model, on which IO language is based, 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. In **Fig. 1** the example of the context switching is shown. In IO every syntactic construction is a message send, messages are separated 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. setProto() dynamically modifies delegation pointer of a receiver. Arguments to methods are passed in unevaluated AST from, and can be evaluated on demand, arbitrary number of times and in arbitrary context, which allows expressing of control flow constructs and new method definitions as regular method calls.

```
foo := method(bar println)
contextA := Object clone do(
    bar := method("in context A") )
contextB := Object clone do(
    bar := method("in context B") )

thisLocalContext setProto(contextA); foo
> in context A
thisLocalContext setProto(contextB); foo
> in context B
```

図 1 Context-oriented programming in IO

Dynamic mixin is a technique to temporarily modify object hierarchy by inserting a mixed-in object (**Fig. 2**). Dynamic mixin insertion can be done in a straightforward way by assigning the delegation pointer to insert a new object into delegation chain, so it is a special case of dynamic delegate pointer modification. While being less general than arbitrary delegation pointer assignment, dynamic mixin can be used to represent useful techniques, such as aspect-oriented programming [10] and context-oriented programming [14]. In these

```
Dog := Object clone do(
    bark := method(
        "woof-woof /Hey!/" println) )
Pochi := Dog clone do(
    beg := method(
        bark
        "wooo /I want that/" println) )

GoodManners := Dog clone do(
    bark := method("....." println) )

Pochi beg
> woof-woof! wooo

Pochi setProto(GoodManners) //insert a mixin
Pochi beg
> ..... wooo
Pochi setProto(Pochi proto proto) //remove
```

図 2 Dynamic mixin example

uses dynamic mixin operations (insertion or removal) are performed at high frequency, for example, at every invocation of a particular method. Many current implementations of dynamic languages make an assumption that object hierarchy is not changing frequently, and leave it as an expensive operation, so programs using dynamic mixin exhibit low performance [16]. We believe that dynamic mixin is a useful operation and that its potential uses have enough regularity so as to enable efficient optimization.

Dynamic 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, invalidate cached value or recompile binary code. In our previous work [16] we proposed the invalidation 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, it also stores a snapshot of counter, and checks it on each use. Related objects also receive a pointer to the state object, and increment counter in a state object on each modification that can affect the outcome of the method dispatch. State objects are allocated dynamically and associated with sets of polymorphic methods that are called from the same call site, and a pointer to the state object is installed in method table in each object that is being traversed during method lookup. 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 rewind system to the prior state on mixin removal. On mixin insertion, we record old and updated value of state objects in the cache associated with object, to which mixin is installed. On mixin removal we can check if there were any other invalidations, 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 objects. Call sites that see several alternating targets of the dispatch can use cache with multiple entries (similar to polymorphic inline caching), 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 work we set out to explore if we can benefit from this

caching scheme in the dynamic compilation system, by compiling several versions of the code according to the alternating states of the system.

3 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 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 interpreter. When inline cache has multiple dispatch targets recorded, we generate code for each of the cached values separately, effectively producing an inlined version of polymorphic inline cache. Code for control flow constructs such as `while()` and `if()`, and arithmetic operations is generated by compiler intrinsics.

Limitations

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

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

4 Evaluation

Due to the incomplete compiler implementation, our current evaluation options are limited to microbenchmarks. A microbenchmark is structured as a tight loop with a object method call in it, and a target method does integer arithmetic. By varying the number of target methods and compilation options we intend to evaluate the costs of type guard, state check, multi-version inlined code. Performing the experiments and measurements remains a task to complete in near future.

5 Related work

The problem of compilation of dynamic languages has been thoroughly researched to date. Smalltalk [7] was first to use just-in-time compilation for a dynamic language. Self [4] is the project that pioneered many of the commonly used techniques, including polymorphic inline caches and type-split compilation. Psyco [12] 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 [8] [17] 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 [3], Javascript [5] [9]. PyPy [13] is notable for its use of a statically typable language RPython [1], which is a proper subset of full dynamic Python language. 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 [15] and PHP [2].

6 Conclusion

We devised and started 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 benchmarking a small integer numeric benchmark code in interpreted and compiled modes, with and without using dynamic mixin. Completing of the compiler to cover the IO language completely remains our major future task.

Acknowledgments

We thank Sebastian Günther for helpful comments on a draft of this paper.

参 考 文 献

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

[2] Biggar, P., de Vries, E., and Gregg, D.: A practical solution for scripting language compilers, *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, New York, NY, USA, ACM, 2009, pp. 1916–1923.

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

[4] Chambers, C., Ungar, D., and Lee, E.: An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes, *LISP and Symbolic Computation*, Vol. 4, No. 3(1991), pp. 243–281.

[5] Chang, M., Smith, E., Reitmaier, R., Bebenita, M., Gal, A., Wimmer, C., Eich, B., and Franz, M.: Tracing for web 3.0: trace compilation for the next generation web applications, *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, New York, NY, USA, ACM, 2009, pp. 71–80.

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

[7] Deutsch, L. P. and Schiffman, A. M.: Efficient implementation of the smalltalk-80 system, *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, New York, NY, USA, ACM, 1984, pp. 297–302.

[8] Gal, A. and Franz, M.: 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.

[9] Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M., and Franz, M.: Trace-based just-in-time type specialization for dynamic languages, *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, ACM, 2009, pp. 465–478.

[10] Haupt, M. and Schippers, H.: A Machine Model for Aspect-Oriented Programming, *Proceedings ECOOP '09, LNCS 4609*, Springer Berlin / Heidelberg, 2007, pp. 501–524.

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

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

[13] Rigo, A. and Pedroni, S.: PyPy's approach

to virtual machine construction, *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, ACM, 2006, pp. 953.

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

[15] Williams, K., McCandless, J., and Gregg, D.: Portable Just-in-time Specialization of Dynamically Typed Scripting Languages, (2010), pp. 391–398.

[16] Zakirov, S., Chiba, S., and Shibayama, E.: Optimizing dynamic dispatch with fine-grained state tracking, *Proceedings DLS'10*, 2010. to appear.

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