2nd Workshop on Implementation, Compilation, Optimization of Object-oriented Languages, Programs and Systems (ICOOOLPS) 2007. Co-located with ECOOP 2007, pages 17-22, Technische Universität Berlin, Berlin, Germany, July 30, 2007 (ISSN 1436-9915)

Type Feedback for Bytecode Interpreters

Position Paper
ICOOOLPS 2007

Michael Haupt[1], Robert Hirschfeld[1], and Marcus Denker[2]

[1] Software Architecture Group
Hasso-Plattner-Institut
University of Potsdam, Germany
[2] Software Composition Group
Institute of Computer Science and Applied Mathematics
University of Berne, Switzerland
{michael.haupt,hirschfeld}@hpi.uni-potsdam.de, denker@iam.unibe.ch

Abstract. This position paper proposes the exploitation of type feedback mechanisms, or more precisely, polymorphic inline caches, for purely interpreting implementations of object-oriented programming languages. Using Squeak's virtual machine as an example, polymorphic inline caches are discussed as an alternative to global caching. An implementation proposal for polymorphic inline caches in the Squeak virtual machine is presented, and possible future applications for online optimization are outlined.

1 Introduction

Bytecode interpreters are small in size and comparatively easy to implement, but generally execute programs much less efficiently than just-in-time (JIT) compilers. Techniques like threaded interpretation [9, 11, 2] focus on speeding up bytecode interpretation itself, and caching [4, 5, 1] improves the performance of message sends—the most common operation in object-oriented software [7].

It is interesting to observe that, while threading mechanisms are used naturally to a varying degree in bytecode interpreter implementations, such systems usually employ only *global caching* to speed up dynamic method dispatch. A global cache is clearly beneficial with respect to overall performance. Still, it does not provide optimal support for polymorphic message send sites, and it does not allow for exploiting type information (we provide details on these issues in the following section). In our opinion, the employment of *polymorphic inline caches* (PICs) [5] instead can provide means for achieving significant speedups in bytecode interpreters while exhibiting only a moderate increase in memory footprint and implementation complexity.

In the next section, we briefly discuss global caches. The bytecode interpreter we use as a case study throughout this paper is that of the Squeak [8, 12] virtual machine (VM) [13]. Section 3 proposes an approach to the implementation of

PICs in the Squeak bytecode interpreter. Finally, section 4 gives a summary of the paper and an outlook on possible future optimizations in bytecode interpreters that are encouraged by the introduction of type feedback mechanisms.

2 Global Caching: Discussion

The Squeak VM bytecode interpreter uses a global cache for improving method lookup performance. This cache has a fixed size and maps <*target class, selector*> pairs to concrete method implementations (compiled methods). It significantly contributes to the overall performance of the Squeak interpreter.

However, such a cache has several shortcomings. Since it is global, collisions are relatively frequent and lead to longer method lookup times. The cache has to be flushed as soon as changes in the class hierarchy or in method implementations occur. For changes in method implementations, the cache is not entirely flushed, but only for the entries that refer to implementations of the *selector* in question.

Moreover, flushing is required whenever the garbage collector performs heap compaction, as hashing is done based on target class and selector object *addresses*. After a flush, the cache needs to be repopulated, during which and overall performance is lower.

The global cache suffers from being global. It cannot react to local changes in an adequate way—i. e., by an update operation that is quasi-local in its effect on cache contents. A local change, such as a class overwriting an inherited method, actually affects only a small part of the entire class hierarchy. Nevertheless, method lookup data for large parts of the class hierarchy needs to be restored.

Also, the global cache is, due to its mapping scheme, generally not able to provide local information, that is information *per send site*. Such information typically comprises of the concrete receiver types (classes) of a message at a given polymorphic send site. It is called *type feedback information* [6] and is very interesting with regard to optimizations (cf. Sec. 4).

The performance of the Squeak bytecode interpreter is good. Still, we believe that it can benefit from a caching mechanism that supports local type feedback. These so-called *inline caches* [4] are usually used in environments that employ JIT compilation and bring great benefit in terms of dynamic message dispatch performance.

Inline caches store the most recently looked-up method address at each given send site. The address is cached at the send site, replacing the call instruction to the lookup method with a direct jump to the code of the method. Since there is a jump to the cached method, no lookup needs to be done at all. There is only a slight overhead resulting from a check at the beginning of the method verifying that the class of the receiver is the correct one. In case the receiver class has changed, the standard lookup is used instead. Obviously, simple inline caches are not an ideal solution for supporting polymorphic send sites since they already fail if the same message is sent to an alternating list of objects.

PICs [5] store, for a given send site, the method addresses of N past message sends, where N is the cache size. A PIC stores <*receiver class, method address*>

key-value pairs. The Strongtalk [14] VM is a prominent example of such an environment. It is comprised of a bytecode interpreter and a JIT compiler; the interpreter already stores type feedback information in PICs to speed up message sending. Information stored in these PICs is later exploited by JIT-compiled native code.

3 Polymorphic Inline Caching in Bytecode Interpreters

We believe that PICs can be beneficial also in solely interpreting VM implementations, such as the Squeak VM [13]. In this section, we outline an implementation proposal for PICs in that environment.

An interpreter does not generate binary code for methods, thus PICs cannot store memory addresses of code. In Squeak, the bytecode is stored in compiled method objects. Here, PICs can store a reference to the compiled method object instead.

The implementation affects both the Squeak VM and reflectively the Smalltalk *image*. At the image level, each Smalltalk method is represented as an instance of the `CompiledMethod` class. The format of `CompiledMethod` instances needs to be modified to store send site type feedback information. In the VM, the interpreter logic must be augmented to support storing and updating of said information.

Squeak `CompiledMethod`s have the layout shown in Fig. 1. The standard *object header* provides information about the object itself, its class, hash value, etc. The subsequent *method header* contains information on the method in question, such as the number of arguments, local variables and literals. After that, there are several pointers to the method's *literals*, each referencing a given constant occurring in the method. For example, all send bytecodes reference their selector (the name of the method to call) as an offset in the literal frame. The *bytecode* array represents the method code, and the *trailer* carries additional information about the method's source code location.

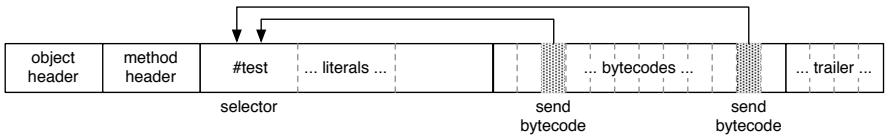

Fig. 1. Object layout of a Squeak `CompiledMethod` instance.

For the PIC implementation in Squeak, the *literals* region in `CompiledMethod` instances is of special interest. In the current Squeak system, the compiler generates one slot in the literal frame for any unique selector. This means that send bytecodes sending the same selector reference the same slot in the the literal array (cf. Fig. 1). The Smalltalk compiler needs to be modified to generate one

entry in the literal array for each send, without the sharing property mentioned above.

Thus we have as many literal slots storing selectors as there are send bytecodes in the `CompiledMethod` (cf. Fig. 2). Each of these slots can hold a reference to an object carrying type feedback information. To this end, several dedicated classes (described below) are to be introduced into the image.

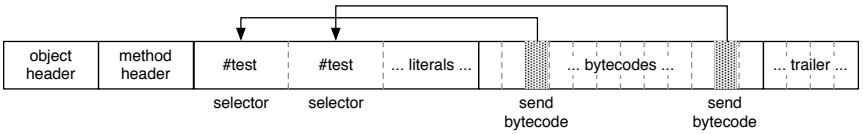

Fig. 2. A `CompiledMethod` instance without selector sharing.

Initially, all selector slots contain selectors. Once a send site is visited by the bytecode interpreter and the corresponding message is sent, the selector is replaced by a reference to an inline cache (IC) object (cf. Fig. 3), an instance of the `InlineCache` class. An IC object contains five values: the selector, the most recent target class of the send, the address of the most recently looked-up method for the selector, a hotness counter, and a trip counter.

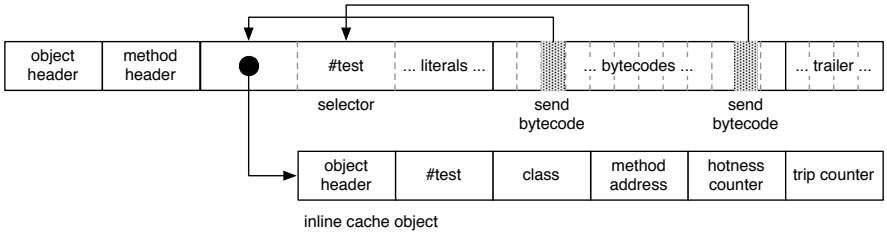

Fig. 3. A former literal slot referencing an IC object.

The bytecode interpreter increases the hotness counter each time it executes the corresponding message send. It then also checks whether the target class is the same as it was when the send was executed the last time. If so, the stored method address is used to retrieve the method implementation to be executed. If the receiver class check fails, the trip counter is increased, and the correct class and implementation are looked up and stored.

If such a send site causes actual lookups too often, its IC object reference can be replaced with a PIC object reference (cf. Fig. 4). A PIC object is organized much like an IC object: it also carries the selector in question and type feedback

information. The notable difference is that a PIC object carries up to 8 triples of receiver class, method address, and hotness counter.

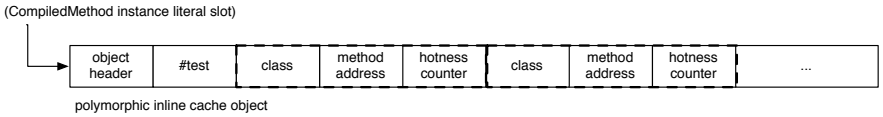

Fig. 4. A PIC object referenced from a `CompiledMethod` instance.

The bytecode interpreter, when executing the corresponding send, iterates over the PIC and checks for the correct receiver class. Once it finds one already stored, the respective stored method is executed and the pertaining hotness counter is increased. If no matching receiver class is found, lookup proceeds as usual and the result is stored in a free slot of the PIC object if available. It is not necessary to store a hotness counter alongside with each PIC object entry, but future optimizations can benefit from this information (see below).

4 Summary and Future Optimizations

In the previous sections, we have discussed caching optimization mechanisms for bytecode interpreters. In our opinion, global caches are, although helpful with regard to performance, not fully supportive of dynamic optimizations possible in interpreters. For that we propose the introduction of PICs based on local type feedback.

The *locality* of type feedback information is a feature of PICs that can be exploited beyond performance improvements in the VM. Type feedback information made available at the *image level* facilitates optimizations above the abstraction barrier imposed by the Squeak VM.

The AOStA (Adaptively Optimizing Smalltalk Architecture) project [10] relies on type feedback from the VM to dynamically and adaptively optimize Smalltalk bytecodes in the image using bytecode manipulation. An example of such an optimization is method inlining: based on type feedback information and hotness counter data, methods frequently invoked from a given send site can be inlined directly at the image level, without the need to create stack frames and method context objects.

Originally, AOStA has been conceived for the VisualWorks VM [3]. The VisualWorks JIT compiler generates PICs at send sites. A system like AOStA can be beneficial also to purely interpreted systems like Squeak Smalltalk if the underlying interpreter supports type feedback using PICs, as proposed in this paper.

References

1. M. Arnold, S. J. Fink, D. Grove, M. Hind, and P. F. Sweeney. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, 93(2), 2005.
2. M. Berndl, B. Vitale, M. Zaleski, and A. D. Brown. Context Threading: A Flexible and Efficient Dispatch Technique for Virtual Machine Interpreters. In *CGO '05: Proceedings of the international symposium on Code generation and optimization*, pages 15–26. IEEE Computer Society, 2005.
3. Cincom Home Page. `http://www.cincomsmalltalk.com/`.
4. L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 297–302. ACM Press, 1984.
5. U. Hölzle, C. Chambers, and D. Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38. Springer-Verlag, 1991.
6. U. Hölzle and D. Ungar. Optimizing Dynamically-Dispatched Calls With Run-Time Type Feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336. ACM Press, 1994.
7. U. Hölzle and D. Ungar. Do Object-Oriented Languages Need Special Hardware Support? In *ECOOP '95: Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 283–302. Springer-Verlag, 1995.
8. 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 *Proc. OOPSLA 1997*, pages 318–326. ACM Press, 1997.
9. P. Klint. Interpretation Techniques. *Software—Practice and Experience*, 11(9):963–973, 1981.
10. E. Miranda. A Sketch for an Adaptive Optimizer for Smalltalk written in Smalltalk. unpublished, 2002.
11. I. Piumarta and F. Riccardi. Optimizing Direct Threaded Code by Selective Inlining. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 291–300. ACM Press, 1998.
12. Squeak Home Page. `http://www.squeak.org/`.
13. Squeak Virtual Machine Home Page. `http://www.squeakvm.org/`.
14. Strongtalk Home Page. `http://www.strongtalk.org/`.