A Block-Based Bytecode Format to Simplify and Improve Just-in-Time Compilation

Christian Wimmer

Department of Computer Science
University of California, Irvine
cwimmer@uci.edu

1. Problem Description

To support portability, Java applications are compiled to platform-independent bytecodes that are then executed by a virtual machine (we base this paper on Java, but the idea can be easily applied to other current systems, especially the .NET infrastructure). The structure of the bytecode format is largely inspired by the structure of Java source code. Both share basically the same method structure. However, the Java bytecode language itself is more flexible than the Java source language, i.e., it is possible to manually generate bytecodes that have no representation in Java code. Also, the bytecodes contain no information about the internal structure of a method, e.g., information about loop structures is no longer available.

Modern virtual machines perform aggressive optimizations at run time. When bytecodes are translated to optimized machine code, the original method granularity is dissolved by standard optimizations like method inlining. Recent research projects, like partial method compilation and trace compilation, change the structure of the resulting machine code even more radically. The current method structure of the bytecodes is of more a burden than a help for optimizations. Because arbitrary control flow is allowed and no high-level structural information like loop information is present, every compiler has to regenerate this information in the first compiler phases.

We propose a new bytecode format that is based on *basic block granularity*. A block consists of bytecodes that are always executed sequentially together (as long as no exception is thrown by a bytecode). The blocks are connected using control flow information, which should include high-level semantics like the loop structure. This simplifies an optimizing just-in-time compiler in two ways: the compiler does not need to analyze and then discard rarely executed parts of a method, and the compiler does not need to re-introduce a structure for the frequently executed parts.

In the Java language, local variables can be declared in scopes smaller than the whole method, but in the current bytecode format local variables are at the method level. In the proposed format, local variables are defined at the block level. This requires an efficient parameter passing and calling convention between blocks. Variables that are alive in multiple blocks should be managed so that the compiler can easily identify all positions where a value is assigned and where it is used. We need to evaluate if static single assignment (SSA) form is beneficial for these purposes.

The primary focus of the new format is to simplify just-in-time compilers that translate bytecodes to machine code. However, the new format should also be suitable for interpretation. This leads to interesting research problems to find a suitable tradeoff: Efficient interpretation requires a format where local variables and temporary values can be stored in a compact data block and accessed efficiently at run time. In contrast, the just-in-time compiler usually wants to treat long-living values differently to short-living ones, so one variable area might not be optimal here.

Optimistic dynamic optimizations in the just-in-time compiler require the ability to optimize based on currently loaded code, and to revert the optimization decisions later on when preconditions change. *Deoptimization* [4], i.e., the ability to switch back from optimized machine code to unoptimized interpretation, has shown to be a simple and convenient support for such optimizations. We claim that the bytecodes must be structured in a way to allow efficient deoptimization. In summary, the following question must be answered regarding the interpreter: Should one interpreter stack frame be created per method—which in some sense re-introduces method boundaries—or should there be one interpreter stack frame per block, per loop, or some other granularity—which certainly increases the interpretation overhead.

While Java bytecodes are not tied to the Java source language, only bytecodes and concepts required for Java are currently present. This makes it difficult to compile dynamic languages, like Python and Ruby, and functional languages, like Scheme, to Java bytecodes. Concepts like dynamic method invocations, tail calls, and closures are not available in Java bytecodes and must be simulated, e.g., by splitting code up into synthetic methods and interfaces. The block granularity of the new bytecode format should simplify this. No synthetic method structure is necessary, an ex-

tended set of annotations for blocks is sufficient. For example, a block that ends with a method call can be annotated with a flag that this is a tail call.

2. Related Ideas

Several other recent research projects are related to this topic and should be used for inspiration of the new format:

- Profile information: Advanced compiler optimizations require the collection of profile information at run time. How is profiling affected, and possibly simplified, by the new bytecode format? The efficient collection of profile information and the suitability for optimistic and feedback directed optimizations [2] is a crucial design goal.
- Bytecode formats based on static single assignment (SSA) form: A prominent example is the format used by the Low-Level Virtual Machine (LLVM) [6] project.
- Inherently safe bytecode formats (see for example [1]): Bytecodes from untrusted sources must be verified prior to execution. By designing the format so that it is easy to verify, or so that it is even impossible to encode malicious programs, is a major goal.
- Client-server solutions for embedded systems where reduced or even block level granularity is used to transfer code from a sever to a client. Only code actually executed is transferred (see for example [11]).
- Stack-based vs. register-based bytecodes: Studies in the last year compared stack-based and register-based bytecodes (see for example [8]). This information can help to decide which format to use for the new bytecode format.
- Region based compilation, partial method compilation, and partial method inlining (see for example [9, 12]): It is necessary to evaluate how these ideas are simplified by the new bytecode structure.
- Trace compilation [3]: Trace compilers record and optimize actual execution paths through an application. Not having a method structure should simplify them.

3. Results

A successful prototype implementation of the idea must show that such a bytecode format is well defined, safe, efficiently executable, and generally useable. We believe that the following is necessary for evaluation (assuming that the system is based on Java, but it can also be based on the .NET infrastructure):

- A definition of the new bytecode format.
- A modified version of an existing VM that executes the bytecodes. Possible candidates are either research VMs (Jikes RVM [5], Maxine VM [7]), or even better a production-quality VM like the Java HotSpot™ VM [10].
- A compiler that generates the new bytecodes for Java applications, either directly from Java source code or by translating Java bytecodes.
- A compiler that generates the new bytecodes from a dynamic language, like Python or Ruby. This should show that the new format is better suitable for dynamic languages than current Java bytecodes.

References

[1] W. Amme, N. Dalton, J. von Ronne, and M. Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–147. ACM Press, 2001.

[2] 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):449–466, 2005.

[3] A. Gal, C. W. Probst, and M. Franz. HotpathVM: An effective JIT compiler for resource-constrained devices. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 144–153. ACM Press, 2006.

[4] U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 32–43. ACM Press, 1992.

[5] *Jikes RVM*, 2009. http://jikesrvm.org/.

[6] *The LLVM Compiler Infrastructure*, 2009. http://llvm.org/.

[7] *Maxine Research Virtual Machine*, 2009. http://research.sun.com/projects/maxine/.

[8] Y. Shi, D. Gregg, A. Beatty, and M. A. Ertl. Virtual machine showdown: stack versus registers. In *Proceedings of the International Conference on Virtual Execution Environments*, pages 153–163. ACM Press, 2005.

[9] T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Transactions on Programming Languages and Systems*, 28(1):134–174, 2006.

[10] Sun Microsystems, Inc. *The Java HotSpot Performance Engine Architecture*, 2006. http://java.sun.com/products/hotspot/whitepaper.html.

[11] G. Wagner, A. Gal, and M. Franz. SlimVM: Optimistic partial program loading for connected embedded Java virtual machines. In *Proceedings of the International Symposium on Principles and Practice of Programming in Java*, pages 117–126. ACM Press, 2008.

[12] J. Whaley. Partial method compilation using dynamic profile information. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, pages 166–179. ACM Press, 2001.