A Better way of running OO bytecodes? A Research Proposal.

Mathew Zaleski*

23rd February 2004

1 Introduction

There are good reasons why deploying Java applications as object-oriented machine-neutral bytecode with late binding between classes is attractive. For example, application code can be downloaded on demand. However, implementing a program to execute the bytecode quickly is complicated. Experience has shown that a straightforward implementation of a virtual machine based on a bytecode interpreter is very slow.

The performance challenges faced by Java can be loosely categorized as those related to bytecode interpretation and those related to Java's object-oriented, late-bound approach. It is reasonably well understood how to execute leaf methods quickly. It is less clear how to claw back the overhead imposed by late-binding object-oriented techniques.

Java just-in-time, or JIT, compilers translate the bytecode of selected methods to native code after they have been loaded. The interpreter then arranges to dispatch these methods instead of interpreting them. Even naively-generated native code runs about two or three times faster than the originating bytecode can be interpreted. As a result, realistic Java leaf methods containing loops and complex expressions perform acceptably well.

Two properties of object-oriented systems like Java, Smalltalk and Self conspire to complicate the life of the JVM implementor. First, callsites can be polymorphic. Second, linkage to methods is carried out at run-time, which is extremely late relative to languages like C++ and FORTRAN.

*Research Proposal for DCS PhD
$Revision:$$.$$$

This draft is scheduled to be presented May 7, 2003.
If you are not a collaborator or advisory committee member then please ask permission to read further.

contact author at matz@cs.toronto.edu

Polymorphism Polymorphic callsites are a consequence of how object-oriented languages invoke methods on objects. The type of object on which the method is invoked determines the actual destination. The idea is that each variety of object may override a method and so at runtime the system must dispatch the definition of the method corresponding to the type of object. In many cases it is not possible to deduce the exact type of the object at compile time. Since, in such situations, the object might turn out to one of multiple types we refer to the call as polymorphic. For example, consider an array of objects in Java (or Smalltalk or Self). Suppose we code a loop that iterates over the array and invokes the `toString` method on each element. Since the array can contain any type of object we cannot tell what type of object `toString` will be invoked on. There is no single destination of the callsite.

The dispatch of polymorphic methods in statically-bound systems, such as virtual functions in C++, is simpler since the link editor and loader can enumerate all possible destinations of each callsite. The construction of efficient virtual function dispatch tables that makes this possible has been heavily studied and is well understood [4].

Late binding in Java stems from the requirement to support mobile code. The binding between a polymorphic callsite and its callees must be carried out at run time because when the calling class is loaded we may have not yet loaded the classes defining the possible callees. Both a bytecode interpreter and code generated by a JIT compiler must be prepared to occasionally search through loaded classes to locate the target method at the time of the call. Caching techniques can be very effective in conjunction with a JIT. For instance, Hölzle's Polymorphic Inline Cache [5] reduces the cost of the dispatch after the initial dynamic look-up has been done.

The indirection inherent in a polymorphic callsite acts as a barrier to inter-procedural optimization. Given the tendency of modern object-oriented software to be factored into many small methods which are called throughout a program, even in its innermost loops, the optimization barriers can lead to poor native code quality. A typical example might be that common subexpression elimination cannot combine redundant memory accesses separated by a polymorphic callsite because it cannot prove that all callees do not kill the memory location. To achieve performance comparable to procedural compiled languages, inter-procedural optimization techniques must somehow be applied to regions laced with polymorphic callsites.

It turns out that modern programming practice results in little effective polymorphism. Effective polymorphism measures how many destinations a given callsite actually dispatches, on the average. Various studies have indicated that almost all callsites are effectively monomorphic, which is to say that they always dispatch the same method. This fact suggests that we should not compile a callsite until we have had a chance to verify if it is monomorphic and record its destination. Efficient dispatch code can then be generated that performs well when the callsite continues to be monomorphic.

Translation of Bytecode to native code The strategy pursued by state-of-the-art method-based JIT compilers is to wait until a called method is "hot" and then translate it to native code and optimize it. This is essentially a gamble that an investment in dynamic compile time will be recouped by future execution of the resulting optimized native code. As the method is optimized, it is desirable to inline methods it calls at selected callsites for which the destination is known. This results in efficient dispatch as well as the opportunity to expose caller and callee to optimization together. But what about the callsites that have not yet executed and hence whose destinations are still unknown? Only generic, potentially poorly performing code can be generated for these. Later, when more is known about these callsites, the only recourse is to compile the entire method again. This can be expensive because in order to recompile a relatively small part of a method we may have to recompile its caller(s) and all its inlined callees as well.

It is significant that some of the infra-structural complexity of the method-based approach is self-inflicted, in the sense that code is optimized at the same granularity as it is translated into native instructions, namely the method. It may be preferable to translate relatively fine-grained regions of bytecode (as they are recognized as hot) but optimize more coarse-grained portions of the overall control flow graph (CFG) as it emerges in the hot regions of the program.

We would like to find a way to organize a JIT so that it can translate relatively fine-grained regions of bytecode to machine instructions and yet optimize more coarse-grained regions of the program. A recent dynamic optimizer for binary code built by Bala, Duesterwald and Banerjia at HP labs identifies a possible approach to solving this problem. HP Dynamo [1, 2] starts out by doing a rather strange thing for a system intended to speed up the execution of programs: It interprets highly optimized binaries instead of running them. Dynamo starts with a highly optimized executable ready to run on native hardware. Then, counter-intuitively, it starts to interpret the executable rather than dispatch the binary. As interpretation continues, Dynamo heuristically identifies hot regions of the program, which are termed "traces". Dynamo dynamically discovers the loops in the program and does not consider the call graph. Dynamo then enters a mode called "trace-generation" in which native instructions are issued into a trace cache as the originating code is interpreted. The resulting code fragments that are added to the trace cache only include code for the exact path through the program that was traversed during trace generation. Traces link together and execution soon occurs almost exclusively from the trace cache. It turns out that for compute intensive benchmarks almost all time is spent executing traces and not interpreting the originating methods. In fact, less than one percent of the time is spent interpreting. Simple code straightening optimizations that improved branch prediction had sufficient impact that Dynamo actually ran most SPEC benchmarks faster than the underlying executables. Dynamo obtained performance gains even with small trace caches of only a few hundred kilobytes.

My own detailed review of Dynamo, including a discussion of some of the problems with the approach such as the *return guard problem* appears in [10].

In addition, my hypothesis that the performance gains were at least partially due to instruction cache prefetch effects is introduced. A discussion of how polymorphism can be accommodated is given by [9].

The techniques pioneered by Dynamo are relevant to a Java JIT because they show how we might organize inter-procedural optimization at the same time as compile only code we have seen before. Inter-procedural optimization is carried out as traces extend across calls. The fact that all code within a trace has been executed before means that at least one destination of each callsite is known. The Dynamo results prompt us to recognize that the unit of translation need not relate to the structure of the source code or even the call graph of the originating program.

I propose to sidestep some of the difficulty experienced by method-oriented JIT compilers by adopting a different unit of compilation. Rather than compiling entire methods, I suggest that an enhanced Java interpreter should trace-generate, like Dynamo, fragments of native code for hot paths through the bytecode of a Java program. When dispatched, the generated code will function only for the path followed when it was generated. If execution must diverge from this path then the fragment will exit and normal Java interpretation will resume. As these forays away from compiled code themselves become hot they will be trace generated and linked in to our code cache as well.

I anticipate that the main contribution of this project will concern how optimization should be organized in such a trace-based JIT. Any reasonably simple code generation technique to lower the bytecode of a trace to native instructions is bound to leave redundant code in each trace. One open question is: "How can this code be exposed to classical optimizations such as common subexpression elimination and loop invariant code motion?" The issue is that any given trace does not contain much of the loop structure of the program so optimizing each individually is unlikely to help[1].

My suggestion is to delay global optimization until enough traces have been linked together to expose the loop structure of the program. At that time the optimizer will have a reasonably broad view of the overall CFG of the program. Potentially many traces will be combined into a *combined compilation unit* (CCU) and optimized. Since every callsite has been executed previously, we need expose only the effectively monomorphic destination of each to the optimizer and represent other possibilities as an exceptional return. How to represent these exceptional returns so that reasonable optimization is possible, and yet we can correctly return to the interpreter, is an open question.

The research I propose to carry out is to modify a production IBM Java interpreter and JIT. The interpreter will be modified to detect and trace generate bytecode traces. The mechanisms of the JIT will be employed to locally optimize a trace prior to linking it in a trace cache. The underlying runtime system will have to be extended to support trace execution. As traces are linked into the trace cache a special-purpose CFG should be updated. A heuristic will run as each new fragment is inserted to determine whether the overall CFG has evolved

[1] Dynamo isn't of much assistance here as it started with highly optimized native code.

such that global optimization is warranted and which set of traces should be subjected to it.

My thesis statement is that: **A trace based jit will identify regions of a program to translate into native code more effectively than method-based approaches with the result that fewer bytecodes need to be interpreted. Applying global optimization to code assembled from the fragment cache will reduce the volume of code optimized and hence reduce the overhead of this costly operation. Since the fragment cache contains only callsites that have been dispatched previously the inherent indirection of their polymorphism can be hidden from the optimizer to some extent. This will improve our ability to optimize across callsites and hence should improve code quality. Overall Java program performance will be improved.**

Structure of this document In the rest of this document I shall describe my proposed research, methodology, contribution and current status.

2 Background

A survey of the background to this research appears in a separate document prepared for my depth oral [10].

3 Proposed Research

My goal is to show that by extending a Java interpreter with a dynamic, optimizing, trace-oriented JIT we demonstrate an effective new way of structuring a JVM. In this context "effective" means that the compile time and compiler footprint will be at least comparable to a typical procedure-based JIT compiler. Obviously my intention is that the resulting execution time and trace-cache footprint will be better or similar to current Java JIT systems employing comparable levels of optimization.

We need to be careful to distinguish which tasks address open research questions and which tasks solve unfamiliar compiler engineering problems. Each will be discussed in more detail below.

3.1 Research Questions

The primary research tasks are to:

- Design the trace selection scheme;
- Identify a trace-generation-time heuristic to combine traces into a new single-entry combined compilation unit (CCU) for optimization;

- Devise an intermediate representation for trace exits such that an optimizer is able produce good code for the CCU and also generate correct code for trace exits themselves;

- Devise new optimizations to improve traces, for instance an optimization to collapse return guards in the CCU.

More detailed descriptions of these follow.

Trace Selection Most of my effort to date has been expended on this aspect of the project. Initial results indicate that even a straight-forward trace-selection heuristic works fairly well for Java bytecode. Nevertheless a couple of issues, described in the Jootch document [9], could benefit from further investigation:

- How can code replication, caused by the trace selection heuristic, be managed? Is it, in fact, a problem, or is it a feature in the sense that it may improve branch prediction and/or i-cache prefetch and perhaps create downstream scheduling opportunities?

- Is there still a "return guard problem" once we are executing the CCU code? If so, should it be addressed by modifying the trace selection heuristic, replicating traces in response to a hot return guard, or a special-purpose transformation run as part of CCU optimization?

Trace combination or ccu formation How exactly should trace optimization be delayed until sufficient traces can be combined and optimized together as one control flow graph? Is a single-entry CCU realistic? How will trace exits from the resulting CCU be handled? A simple experiment is described in [8].

Trace exit representation The whole point behind assembling traces into a combined compilation unit was to sidestep the barriers to optimization represented by polymorphic callsites. Care must be taken lest trace exits result in similar barriers. We must invent a representation that describes trace exits in such a way that that they do not form barriers to data flow analysis and hence hamstring optimization. We will first attempt to express the trace exits as exceptional returns from the CCU.

CCU optimization I hope that it will be straightforward to run many classical optimizations to improve the CCU. In addition, adaptations of classical algorithms may be able to deal with specific issues left over from the traces. For instance, a modified form of store motion may be able to move stores to local variables into trace exit glue code.

Special-purpose optimizations that are specific to our trace-based approach might have great impact. Two examples follow: First, many trace exits assert a predicate about the code. For instance, a call guard will assert the type of an object. Optimizations that propagate this information may be quite effective.

For instance, subsequent trace exits asserting the same predicate could be eliminated. Second, return guards have shown up as a problem in the DynamoRIO project. Once traces have been combined into a CCU it is possible that calls and return guards could be matched and thus eliminated.

3.2 Compiler Engineering

Our primary focus is to shed light on the research questions described above. Thus, we need to choose a software infrastructure that reduces the amount of engineering work that must be done. In an ideal world we could initially use the infrastructure in a relatively unmodified state and later make trace-related enhancements to enhance performance.

3.2.1 Experimental Infrastructure

Mr Kevin Stoodley of IBM manages the team designing and building the IBM VisualAge JVM and JIT product. Our initial conversations indicate that their system should serve my purposes reasonably well. Intellectual property considerations have been handled by arranging an IBM CAS fellowship through Professor Demke Brown.

It was tempting to extend the Java simulator I have already built to compile and execute native code traces. However, the creation of a new back end (code generation and optimization) infrastructure has little to do with the research at hand. A few other infrastructure possibilities have also been considered and rejected. After very little investigative work I have rejected the Microsoft .NET compilation infrastructure because I fear little or no access to the designers and developers would be possible. I reject the GNU Java compiler infrastructure because it has not been integrated with an interpreter. The same can be said of the IBM Jikes RVM infrastructure.

3.2.2 Engineering Tasks

A lot of engineering work will be required to design a scheme to support trace exits and the other novel features of our proposed approach. However, we will attempt to adapt our implementation to the features offered by the IBM JIT infrastructure as best we can. The engineering of a prototype doesn't have to be complete or totally general – though it should at least make clear which engineering issues remain to be addressed. Some of the obvious engineering tasks are:

- Integrate the trace-selection heuristic and trace-generation scheme into the IBM threaded interpreter;
- Adapt the method-based JIT to compile traces;
- Adapt the runtime from the method-based JIT to dispatch the traces and manage a trace cache;

- Adapt the IBM optimizer to process a CCU.
- Adapt the IBM implementation of "high speed debug"[2] functionality to recreate local variables and operand stack suitable for trace exits.

4 Expected Contribution

Naturally I hope to demonstrate that a trace-based JIT has performance benefits. The benefits may come from a reduction of interpretation overhead, a savings of compile and optimization (by omitting cold bytecode) and perhaps from an improvement in inter-procedural optimization by optimizing dynamically combined sets of traces. There may be micro-architectural benefits like those (better branch prediction and/or better use of i-cache prefetch bandwidth) thought to have benefited Dynamo.

One risk with this style of research is that its success may be judged only by the relative performance of the resulting prototype. A fact of life is that vendors have invested much effort maximizing the performance of their method-based JIT compilers on the common benchmarks. This suggests that the chance of actually outperforming them on the SPEC benchmarks is slim. Nevertheless, research in the dynamic compilation field has traditionally included some aggressive experimentation. Ideas like my concept of assembling compilation units from run-time identified traces should be attempted to guide future design. My investigation of how dynamically discovered traces can be combined into compilation units before optimization will contribute important knowledge to the field whether my precise scheme improves performance or not.

5 Methodology

Using the IBM software base it should be possible, hopefully in not much more than a year, to build a prototype to test these ideas. The general idea is that I will modify the IBM threaded interpreter to select bytecode traces and the IBM JIT to compile and optimize them. Once this general infrastructure is in place we will undertake the core of our research agenda, which is to investigate how to combine traces into a combined compilation unit (CCU) and how the combined compilation unit should be optimized.

Our goal will be to run the SPEC JVM benchmarks on our prototype and compare our results to the unmodified IBM system. The IBM project includes an elaborate performance measurement function that is currently used to report the performance of the IBM product to the SPEC [3] consortium. In addition the performance team reports the resource consumption of the IBM JIT product relative to competitors and other IBM Java systems. I intend to use these same measurement tools to evaluate my prototype.

[2] This is the name used by Sun's Hotspot compiler. See [6, 7]

Possible Activity	Effort months
Instrument existing method-based IBM/OTI production interpreter and JIT to get instruction cache hit ratio and other summary numbers to compare at high level with the results of the Jootch trace selection experiment. Create or obtain native code listing for Trace2 program to compare with hand-coded traces of the trace combination experiment.	1-2
Invent a scheme to support trace generation in the IBM threaded interpreter.	2-4
Extend IBM/OTI production interpreter and JIT to generate and execute unoptimized native code traces.	2-4
Explore how to combine the traces in the trace cache into a CCU. Invent a heuristic that indicates when this should be done. (Like trace links closing loops.) This might be done in the Jootch infrastructure then ported to the IBM JIT.	3-6
Investigate trace exit representation with respect to CCU optimization.	2-4
Investigate return-guard optimization for CCU.	1-3
Write Dissertation	6-??
Total.	17-21

Table 1: Major tasks of the research and guesses as to their best and worst case cost

6 Current Status

6.1 Work to date

Up to this point I have concentrated on two aspects of this research. In the interests of keeping this document reasonably short these are described in much more detail in separate documents.

The first experiment involved the construction of Jootch, a Java simulation of a trace-oriented JVM. The report appears in [9]. The goal was to investigate how the SPECL heuristic could be adapted to polymorphic Java bytecode. The project also served to familiarize me with the operation of a Java interpreter and JVM. Jootch made it clear that traces could be used to select the "hot" bytecode of an executing Java program. I found that trace selection and generation produced traces that accounted for 97% of the bytecode executed by the Java SPEC JVM benchmarks compress, jess and raytrace. The trace exit behavior of the benchmarks was examined in detail and many possible improvements to trace selection were discussed.

The second experiment investigates how traces selected by Jootch can be combined into a combined compilation unit (CCU) and optimized. A small program was created that is structured as a doubly-nested loop obscured by a polymorphic method invocation. Then, bytecode traces as selected by Jootch are combined by hand into a flowgraph and hand-coded in C. The C is optimized by `gcc`. The resulting code is examined and the performance of the hand-coded C compared to Sun's Hotspot JIT. I am encouraged by the results primarily because the gcc optimizer did a good job optimizing the CCU. The fact that the performance of the hand-coded C was better than the JIT might have been due to many factors and probably shouldn't be taken too seriously at this point. The report for this experiment appears in [8].

6.2 Estimated Duration of key tasks

Estimates (rough guesses, really) of how much effort some of the tasks listed above require appear in Table 1. It's only realistic to assume that the estimates in Table 1 are optimistic. Consequently we should expect that the research we are proposing will require in the vicinity of two more years to complete. As usual with a research agenda it should be anticipated that some really difficult and time consuming tasks aren't on the list because we don't know we will have to do them yet.

6.3 Next Steps

My IBM CAS fellowship has recently begun. I have just acquired access to the sources and am in the process of setting up the development environment. Coding is about to begin.

References

[1] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Transparent dynamic optimization: The design and implementation of Dynamo. Technical report, Hewlet Packard, 1999. http://www.hpl.hp.com/techreports/1999/HPL-1999-78.html.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: a transparent dynamic optimization system. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 2000. http://citeseer.nj.nec.com/bala00dynamo.html.

[3] Standard Performance Evaluation Corporation. SPEC jvm98 benchmarks. http://www.spec.org/osg/jvm98.

[4] Karel Driesen. *Efficient Polymorphic Calls.* Klumer Academic Publishers, 2001.

[5] Urs Hölzle. *Adaptive Optimization For Self:Reconciling High Performance With Exploratory Programming.* PhD thesis, Stanford University, 1994.

[6] Sun Microsystems. The Java hotspot virtual machine, v1.4.1, technical white paper. http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html, 2002.

[7] M. Paleczny, C. Click, and C. Vick. The Java hotspot server compiler. In *2001 USENIX Java Virtual Machine Symposium*, 2001. http://www.usenix.org/events/jvm01/full_papers/paleczny/paleczny.pdf .

[8] Mathew Zaleski. Feasibility of combining and optimizing bytecode traces. http://www.cs.toronto.edu/~matz/combiningTraces.pdf, April 2003.

[9] Mathew Zaleski. Jootch, a simulation of bytecode trace selection and creation in a JVM. http://www.cs.toronto.edu/~matz/traceCreation.pdf, April 2003.

[10] Mathew Zaleski. Trace-based dynamic compilation for object-oriented programming systems. http://www.cs.toronto.edu/~matz/depth.pdf, April 2003.