Inlining Java Native Calls At Runtime

Levon Stepanian
Dept. of Computer Science
University of Toronto
Toronto, Ontario
levon@cs.toronto.edu

Angela Demke Brown
Dept. of Computer Science
University of Toronto
Toronto, Ontario
demke@cs.toronto.edu

Allan Kielstra
IBM Toronto Software Lab
Toronto, Ontario
kielstra@ca.ibm.com

Gita Koblents
IBM Toronto Software Lab
Toronto, Ontario
gita@ca.ibm.com

Kevin Stoodley
IBM Toronto Software Lab
Toronto, Ontario
stoodley@ca.ibm.com

ABSTRACT

We introduce a strategy for inlining native functions into Java™ applications using a JIT compiler. We perform further optimizations to transform inlined *callbacks* into semantically equivalent lightweight operations. We show that this strategy can substantially reduce the overhead of performing JNI calls, while preserving the key safety and portability properties of the JNI. Our work leverages the ability to store statically-generated IL alongside native binaries, to facilitate native inlining at Java callsites at JIT compilation time. Preliminary results with our prototype implementation show speedups of up to 93X when inlining and callback transformation are combined.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*compilers, optimization, run-time environments*

General Terms

Languages, Performance, Design, Experimentation

Keywords

Java, JNI, inlining, JIT compilation, native code

1. INTRODUCTION

Currently, there is no single programming language that is universally suitable for all tasks, nor is one likely to emerge. There is an inherent tension between the support for low-level operations (such as bit manipulations) provided by languages such as C, and high-level features such as type safety and automatic garbage collection supported by languages such as Java. Rather than focusing on a one-size-fits-all approach to programming language design, support for *language interoperability* is the preferred approach. In addition to allowing programmers to choose the right tool for the job, interoperability allows the reuse of legacy applications and libraries that may have been written in a different language.

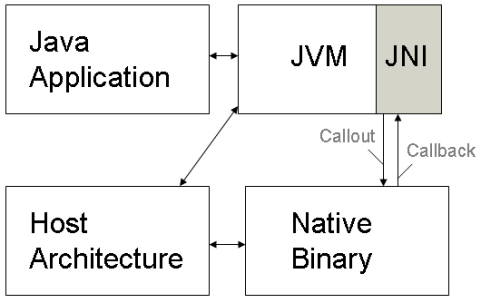

Figure 1: The Java Native Interface

Most high-level languages support interoperability by providing some mechanism for calling code written in a low-level language (such as C). These mechanisms typically impose both time and space overheads at each cross-language function invocation because arguments and results must be packaged carefully to bridge the boundary between the languages involved. In this paper, we focus on the Java programming language [16], because of its widespread adoption for general programming tasks.

Java's interoperability mechanism consists of the Java Native Interface (JNI) [22]. The JNI (Figure 1) is a platform-independent application programming interface that provides language interoperability at the function level. It permits Java code to call functions implemented in external languages (which we refer to as *callouts*), at the same time allowing external code to access and modify data and services from an executing Java virtual machine (which we refer to as a *callback*). Callouts provide Java applications with the ability to leverage legacy, high-performance and architecture-dependent native codes, whereas callbacks imply the availability of Java data types and related functionality in native code. Unfortunately, the JNI shares the same space and time overheads that afflict other interoperability

VEE'05, June 11-12, 2005, Chicago, Illinois, USA.

mechanisms. For example, at each callout to native code, extra data (beyond the actual arguments themselves) must be marshaled into the call to provide the native code with access to the JVM.

Our work addresses these overheads directly by providing a Just-in-Time (JIT) compiler optimization targeting native function calls. Specifically, we extend the JIT compiler's function inlining optimization to handle callouts to native functions, and provide a transformation mechanism to deal with inlined JNI callbacks. We use inlining to reduce the number of callouts, and then take advantage of the JVM context to transform JNI callbacks from overhead-laden operations to semantically equivalent, but dramatically less expensive operations. In addition, we believe that making native function callsites more transparent will expose them to further optimization opportunities in our JIT compiler.

Strategies for function inlining, whether for traditional statically-compiled code, for dynamic JIT compilation, or for polymorphic callsites, are well-researched areas of compiler optimization, and are not the focus of this paper. The main contribution of this work is to extend inlining to native functions during JIT compilation. For our purposes, we simply require support for general function inlining in the JIT compiler framework.

The rest of the paper is organized as follows. In Section 2, we motivate our work by showcasing the ubiquity of native functions and the JNI in Java programs. We also provide more details on the overheads currently affecting the usage of both, and introduce our approach. Section 3 then describes the JVM and JIT compiler framework that our work builds on, and Section 4 describes the design of our proposed solution. The current status of our prototype implementation and initial results are presented in Section 5. We compare our approach to improving JNI performance to prior strategies, and discuss other related work on language interoperability in Section 6. We then conclude in Sections 7 and 8 by summarizing various issues that need addressing as our work progresses.

2. BACKGROUND AND MOTIVATION

The Java programming language provides desirable high-level features, including platform independence, type safety, object orientation, and automatic memory management via garbage collection, which have led to its widespread adoption in many settings. The JNI, a required feature of all specification-conforming JVMs, greatly enhances the applicability of Java by allowing legacy, high-performance, or architecture-dependent native codes to be integrated with Java applications. We begin this section by highlighting some of the uses of the JNI, motivating the importance of this interface for a large number of applications. We then discuss the performance issues associated with use of the JNI, and introduce our strategy for addressing these issues.

2.1 JNI: The Pervasive API

JNI calls occur in many different classes of applications. They have been used in **I/O implementations**, including improving performance of object serialization for distributed computing [9], providing bindings to low-level parallel computing libraries [4, 14], as well as for high-speed network interfaces [35]. The JNI is also used to implement various **JVM frameworks**, such the Java 1.5 Class Libraries as well as the reflective, Java-based OpenJIT compiler from Ogawa et al. [28], and the Microsoft™ Marmot JVM Class Libraries [12].

Strong arguments in favor of implementing **numerical routines** in Java native functions are made by Bik and Gannon [6]; despite improvements in pure Java numerical libraries, interfaces to widely-used but platform-dependent optimized native linear algebra packages are still being developed [17]. The **graphical components** of Java-based user interface libraries, including the Standard Widgets Toolkit [27] and the Abstract Window Toolkit [31], as well as other Java-based multimedia APIs [24] rely on the JNI to make use of underlying architecture functionality that isn't provided in Java. Native codes are also used to recover other functionality unavailable in Java, including low-level hardware performance measuring tools [29], and accurate timers and system resource monitors [5].

The JNI provides much-needed access to low-level native code, but the overhead of this interface is significant.

2.2 JNI Performance Issues

The JNI's strength lies in decoupling native codes from a specific JVM implementation by providing relatively opaque access to JVM internals, data, and services. The cost of this property is lost efficiency, namely large runtime overheads during callouts to native functions, and even larger ones during callbacks to access Java code and data.

2.2.1 Cost of Callouts to Native Code

Generally speaking, a native library containing the function being invoked must be loaded on or before the first *invoke* Bytecode is expected, the containing class and function must be resolved (resolution may require multiple passes over the exported functions in a native library), and related JVM data structures must be updated. These are one-time costs that can be amortized if a particular native function is invoked repeatedly.

During each individual native function invocation, the JVM must also perform work to set up the native stack and possibly registers in order to pass arguments, and must perform null-checking on reference arguments before possibly adding a layer of indirection to them. Upon returning from native code, the return value must be pushed onto the Java stack and the native stack must be restored. In addition, exception status must be checked and local reference cleanup performed on return. JVMs with JIT compilers may reduce these overheads by generating specialized code segments to perform some of the required work at native call sites, as is done in the Intel™ Open Runtime Platform [10] and in our JIT compiler (see Section 3.1).

Sunderam and Kurzinyec [33] studied the performance of different types of native calls using different JVM implementations. Slowdowns when using native functions ranged from a factor of 1.05 to a factor of 16 in the worst case. Only one case achieved a speedup during a native call. Similar results are produced in overhead-measuring experiments performed by Murray et al. [25]. These high overheads are cause for concern, especially since one of the motivations for using the JNI is to allow access to high-performance native code.

2.2.2 Cost of JNI Callbacks

Although callouts are reasonably expensive, the more significant source of interoperability overhead occurs during the

invocation of JNI callbacks. Because of the JNI's platform-independent design, JNI functions are only callable through a reference to the *JNI Environment* variable (the `JNIEnv` pointer). This *JNI Environment* is a table of function pointers, each referring to the implementation of a specific callback function. A callback thus pays a performance penalty because two levels of indirection[1] are used - one to obtain the appropriate function pointer through the `JNIEnv` pointer, and one to invoke the function using that pointer. Other, more specific callback overheads depend on the JNI function being called:

- **String and array parameters** - To make use of string and array data originating from a Java application, native code must first acquire native-side access to them. Unfortunately, this requires expensive runtime copy operations whose purpose is to leave the JVM's copy of the data untouched. The JNI, however, also provides callbacks that claim to increase the chances of receiving direct references to underlying JVM data, but this is left to the JVM's discretion and also places certain restrictions on the programmer's freedom. Because JVMs may implement these callbacks in any way they choose, there is no guarantee that better performance will actually result from their use. Sunderam and Kurzinyec [33] demonstrate that the achieved performance varies widely across different JVM implementations.

- **Fields and methods** - Using Java data types, modifying object data, calling methods and accessing JVM services from native code are also performed via callbacks. For example, modifying a field of an object or calling a static method first requires retrieving a handle to it. This retrieval is commonly implemented as a traversal on the JVM's reflective data structures, in addition to expensive string-based signature comparison operations at runtime [8]. Results in [33] highlight these overheads: field accesses in Java are orders of magnitude quicker than those via the JNI.

- **Other JNI functions** - The JNI also provides callbacks that instantiate objects, manage references, handle exceptions, support synchronization and reflection as well as those that provide the ability to embed JVMs inside native code. These functions share costs similar to those of field and method access callbacks, but have their own unique set of additional overheads. We exclude analyzing them in this paper.

We also note that callbacks may block if a JVM is in the midst of performing a blocking task (such as garbage collection) when the callback occurs.

2.3 Addressing JNI Overheads

Given the extensive use of the JNI in existing applications, we believe the performance penalties must be addressed directly, rather than by introducing changes to the interface, or introducing a new interoperability mechanism for Java. Further, since high-performance JVMs include JIT compilers, it is appropriate to leverage JIT optimizations to reduce the overheads incurred as a result of using the JNI for interoperability. Instead of simply generating efficient code to perform the extra work required at JNI invocation points, however, we aim to eliminate much of this extra work entirely. We consider other strategies for dealing with JNI overhead in more detail in Section 6.

Our approach is to extend the JIT compiler's function inlining optimization to handle native function calls. Once native code has been inlined at its callsite in a Java program, it is no longer necessary to set up and tear down a native stack, or perform other expensive operations to pass arguments. More importantly, the callbacks, designed to gain access to internal JVM state, can now be transformed into compile-time constants or lightweight Bytecodes that preserve the semantics of the original source program. Our prototype implementation shows significant performance increases from inlining native code and transforming JNI callbacks for simple microbenchmarks. Although the implementation is not complete, we expect these benefits to translate into performance improvements in real applications that make extensive use of the JNI. In addition, we believe that native function inlining will expose native code to other optimization opportunities, and reduce the need for the JIT compiler to make conservative assumptions when optimizing Java code.

Our strategy for native function inlining has been prototyped and evaluated in the context of a high-performance production JVM and JIT compiler from IBM. Before describing the details of our strategy, we thus first describe this framework.

3. JIT COMPILER FRAMEWORK

In this section, we introduce the VM and JIT compiler technologies from IBM that form the basis of our implementation, and present the inlining strategy employed by the JIT compiler.

The **IBM® TR JIT compiler** is a high-quality, high-performance optimizing compiler, conceived and developed at the IBM Toronto Software Lab. Designed with a high level of configurability in mind, it supports multiple JVMs and class library implementations, targets many architectures, can achieve various memory footprint goals and has a wide range of optimizations and optimization strategies.

The control flow for the TR JIT consists of phases for intermediate language (IL) generation, optimization and code generation as depicted in Figure 2. When compiling a method, the **IL Generator** walks the method's Bytecodes, and generates tree-based JIT compiler IL (referred to as TR-IL) that also encodes the control flow graph. The **Optimization** phase is a pipeline through which the TR-IL flows and may be modified and reordered by architecture-independent/dependent, speculative and profile-based adaptive optimizations. The **Code Generation** phase lowers the TR-IL to a specific machine instruction set, performs register allocation and schedules the instructions before emitting a final binary encoding. Auxiliary data structures which include stack maps and exception tables are also generated at this point.

The TR JIT compiler is currently used by the IBM J9 Java virtual machine[2]. J9 is a clean-room JVM implementation targeting numerous different processors and operating systems, supporting ahead-of-time compilation, and method hot-swapping as well as a host of other features.

[1]Only one level of indirection is required for C++ natives.

[2]performance results for a TR JIT enabled J9 virtual machine can be found on www.spec.org

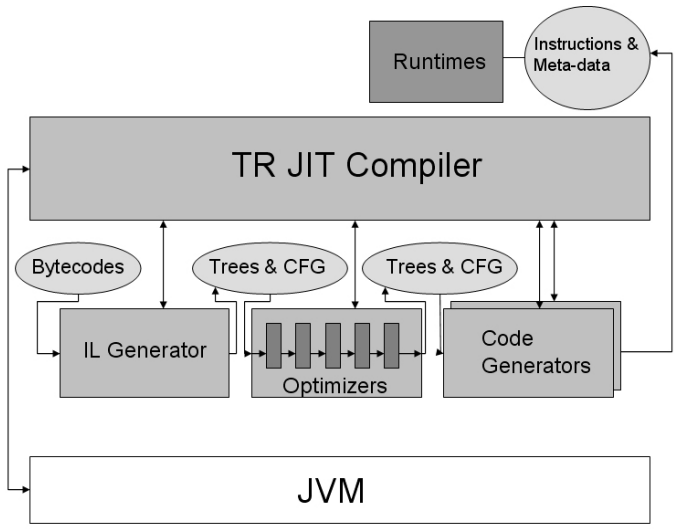

Figure 2: Architecture of the TR JIT compiler

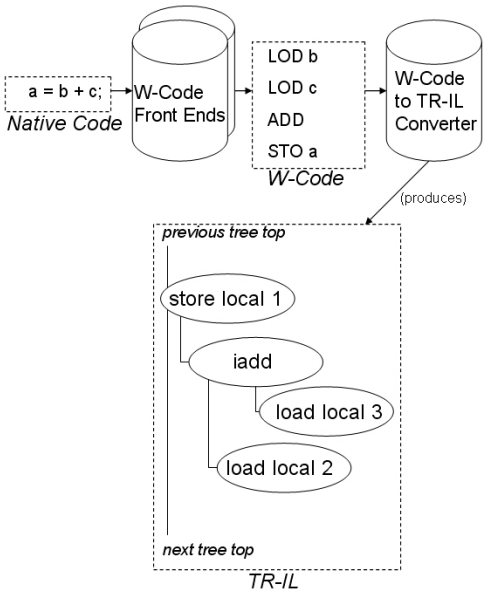

Figure 3: The IL conversion process

3.1 Java Function Inlining in the TR JIT

The TR inlining optimization reduces the overhead of function invocations by inlining entire functions at their callsites. The primary purpose of this inlining, however, is to expose more IL to the optimizer and to eliminate pessimistic assumptions that must be made about the behavior of opaque calls. Like most inlining strategies, it uses a variety of heuristics when determining if a given function call should be inlined. Once the decision has been made to inline a function, the inliner generates TR-IL for the callee, and completes the process by performing all the required transformations on both the caller and callee functions.

The TR JIT currently handles *invoke* Bytecodes that have a native target by generating code that transfers the native call setup and tear-down work to J9, or by using a proprietary mechanism known as *Direct2JNI* that is based on compile-time signature parsing to produce compiled glue code tailored to perform the native call to each unique native target. Direct2JNI supports inlining of the tailored native dispatch into the Java caller for non-virtual methods, but is restricted to synthetic glue methods for other types of methods. In this way Direct2JNI is able to support and therefore optimize call sites where some receiver types have native implementations while others have pure Java implementations using a common dispatch sequence.

Our work extends the inlining strategy in the TR JIT to native function calls. Our focus has been on providing this novel functionality, rather than exploring new heuristics that might be more suitable for native code. We thus use the existing heuristics to decide when the call should be inlined.

4. NATIVE FUNCTION INLINING

Our design for native function inlining consists of three major components. First, we introduce a conversion engine that translates native code into TR-IL. Second, we introduce an enhancement to the inliner, allowing it to synthesize calls to opaque native functions (i.e., calls to other native functions that occur in the inlined code, but that cannot themselves be recursively inlined for a variety of reasons). Third, we introduce a callback transformation mechanism that replaces expensive callbacks with compile-time constants and cheaper Bytecode equivalents, while preserving the semantics of the original source language. We now describe each of these components in detail.

4.1 The W-Code Conversion Engine

The first phase of our native inlining strategy requires the conversion of native code into the same IL used by the TR JIT compiler. To do this efficiently, we exploit the ability to store IL alongside native executable code in the same binary object file or library. This strategy of retaining the IL generated by a traditional static compiler to support future optimizations is similar to strategies used to support link-time cross-module inlining optimizations in several commercial compilers, include those from HP [2] and IBM [20]. It is also reminiscent of the strategy for supporting "life-long" program optimization used in LLVM [21].

In our case, the original IL is *W-Code*, a mature stack-based Bytecode-like representation generated by IBM compiler front-ends for C, C++, FORTRAN, COBOL, PL/1 and other programming languages. Because W-Code is designed to support a large number of languages, aliasing is made explicit in the IL itself. Aliasing is also explicit in TR-IL, making it possible to preserve alias information from the W-Code of native functions when they are converted to TR-IL. As depicted in Figure 3, the W-Code to TR-IL conversion engine operates by iterating through the W-Codes of a native function, and generating TR-IL for each encountered instruction. Once W-Codes have been processed, TR can treat the generated internal TR-IL as if it were derived from Java Bytecodes. In the following sections, we discuss situations where care must be taken to provide appropriate linkage and preserve the semantics of the original language with respect to global variables and external functions.

In our implementation, the IL conversion step occurs at

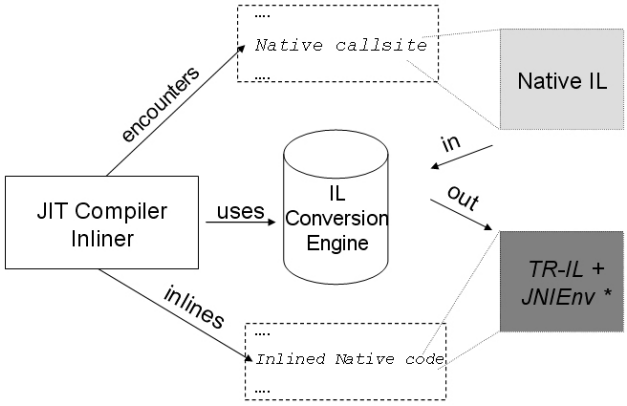

Figure 4: Inlining native codes

runtime, when the inliner decides to inline a particular native function. In principle, the conversion could also be done offline, storing a representation of the JIT compiler's IL along with the native executable. TR-IL, however, is an in-memory IL and is not suitable for efficient serialization to disk. In contrast, W-Code (like Bytecode) is a suitable disk format by design, and the conversion to TR-IL is a single-pass, lightweight operation. Storing the more compact representation and converting at runtime is in keeping with the *slim binaries* strategy proposed by Franz and Kistler [13].

4.2 Inlining and Synthesizing Native Calls

The native inlining proceeds as follows: TR uses the conversion engine described in Section 4.1 to load and generate TR-IL for a Java-callable native function, and then maps parameters to arguments, generates temporaries as needed, merges caller and callee IL and control flow graphs, and materializes the `JNIEnv` pointer for use by inlined JNI callbacks. This inlining process is recursive and considers non-Java callable native functions as potential inlineable candidates as well. Figure 4 depicts this process.

4.2.1 Synthesized Native Functions

The IL for a native method cannot, in non-trivial cases, proceed directly through the rest of JIT compiler processing. This is because it may contain calls to non Java-callable natives for which the JIT compiler cannot render an IL. Such "opaque" calls occur in two situations: (1) calls through function pointers, and (2) calls to functions in binaries where W-Code is unavailable. Opaque native calls are replaced with calls to synthesized methods similar to those produced by Direct2JNI, whose purpose is to call the native function after having set up the proper linkages and context to make the call. The effect is to bridge the Java application to the previously-buried native function. This situation is depicted in Figure 5. In some cases (e.g., when the opaque function is a well-known library function) the call to the function can be effected without going through an entire intermediate native method.

Another concern is data shared between an inlined native function and any other native function (i.e., a synthesized native function, another function in the same library, or a function defined somewhere else). Shared data in this context can be any one of external or static data, as well as addresses of variables (automatics or parameters) that may be passed by an inlined native to one of its synthesized calls. In these cases, TR is careful to ensure that correct linkage is used and the inlined native code is able to read and write to the same portion of shared memory as non-inlined functions. Because this address resolution is performed at JIT compile-time, and the original native function is now inlined rather than called explicitly, additional care must be taken to ensure that the dynamic loading of new libraries is handled correctly.

Finally, the inlined native methods clearly execute in a Java context. Therefore, the code must be conditioned to interact with all appropriate VM requirements. In particular, instructions to perform handshaking with VM components such as garbage collection are inserted before and after inlined native code.

The inliner recursively inlines functions called by a Java-callable native method until it either encounters a call to an opaque function or internal code growth limits are reached. Having reached a termination point, the inliner examines the resulting IL for synthesis requirements, and after satisfying them, continues with normal JIT compiler processing.

Note that inlining a single original native function call may require the synthesis of multiple calls to opaque natives, as depicted in Figure 5. Inlining, however, creates the opportunity to remove the much-higher overhead of callbacks, and reduces the need for conservative assumptions about the behavior of native code in the JIT optimizer. We thus expect that it will often be profitable to synthesize multiple callouts to opaque natives, provided callbacks can be transformed into cheaper operations as we discuss in the following section.

4.3 Transforming JNI Callbacks

The native inlining process is augmented by JNI callback transformations; inlined native code executes in the JVM context, thus there is no need for the `JNIEnv` pointer and the JNI function pointer table to obtain access to internal JVM services and data. Once the native inlining technique has converted W-Codes to TR-IL, it pattern-matches the generated IL, looking for JNI callbacks. This pattern-matching step is necessary because it is impossible to distinguish a function call that performs a JNI callback from any other function call based solely on the IL. Whenever possible, these callbacks are transformed into compile-time constants, and new semantically equivalent TR-IL that represents faster, more direct access to JVM services and data. This process proceeds by using a mechanism, depicted in Figure 6, to iterate over each TR-IL instruction and perform the required transformation.

4.3.1 Identifying Inlined JNI Callbacks

Since our technique is based on transforming TR-IL and understanding the semantics of JNI callbacks, we require a preliminary step that renders each callback defined by the JNI API in terms of TR-IL. This step allows the compiler to understand the expected "shape" of each JNI callback. The "shape" encodes how each callback uses the `JNIEnv` pointer and any other arguments. This representative shape is what uniquely determines a callback. TR uses the constructed shapes for subsequent analysis (and to avoid any attempt to recursively inline a callback). This shape-building step can be performed in three ways: dynamically performed at the

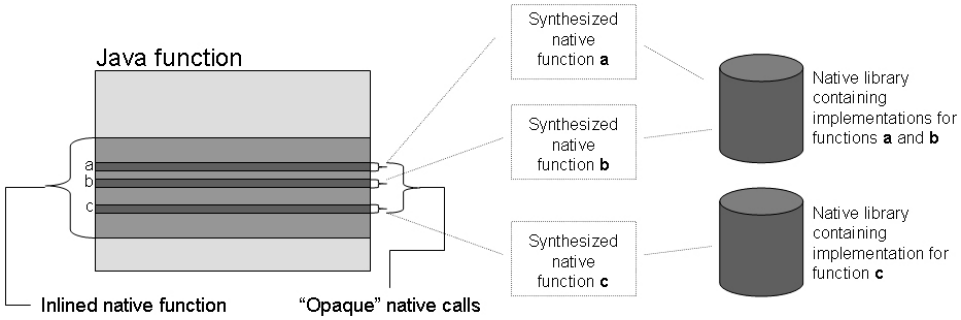

Figure 5: Synthesizing native calls

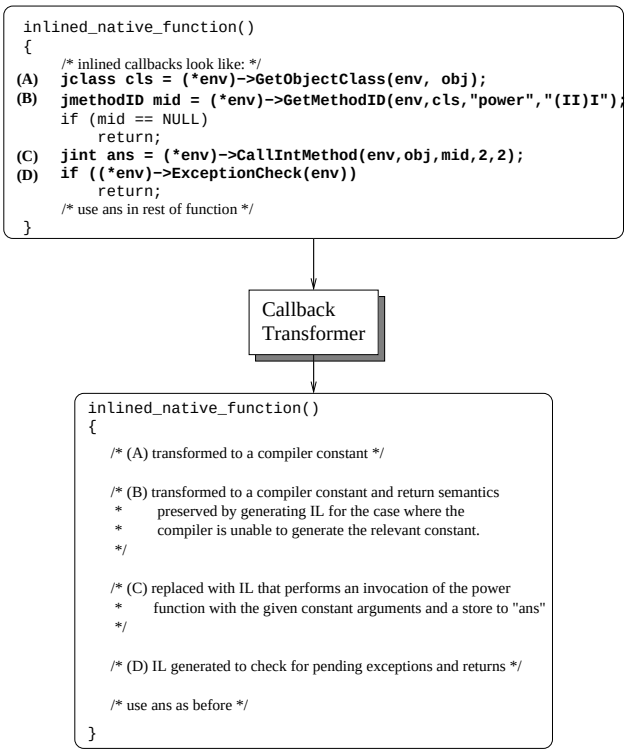

Figure 6: Transforming JNI callbacks

start of a Java program execution, performed as part of the process of building the JIT compiler itself, or statically performed by encoding each callback's shape in the compiler. The only requirement is that the representative IL that is used must be correct for both the current version of the JIT compiler (where the IL definition may change from time to time) and for the JVM being targeted (because each JVM is free to define how the JNI specification is actually implemented). Taken together, these representative pieces of IL are known as the list of shapes that inlined callbacks will be pattern-matched against for identification.

4.3.2 Native Inlining and JNI Callback Identification

When a native call is inlined, care is taken to record uses of the `JNIEnv` pointer within the IL. Recursive inlining is expected and the `JNIEnv` pointer may be passed to recursively inlined functions. However, before recursive inlining, the IL representing the callsite is examined as follows:

1. If the `JNIEnv` pointer is used in the same position in the IL as it appears in any of the constructed shapes, the inliner proceeds to Step 2. Otherwise, inlining continues normally.

2. For each constructed shape in which the `JNIEnv` pointer appears in the same position as it does in the IL for an inlined callsite under consideration, the inliner attempts to match the entire shape to the IL for the callsite. A match occurs if the shape and the IL share the same number and compatible types of arguments. If there is a match, the callsite is not eligible for inlining but may be eligible for transformation. Otherwise, inlining continues normally.

As part of this process, the inliner records the callsites that it has determined do correspond to JNI callbacks and stores them in a list in the order of their appearance in the IL. If one call uses the result of another call, the innermost call will appear in the list before the use of its result.

Theoretically, an optimization such as the one described herein should follow uses of data defined in terms of the `JNIEnv` pointer to track all possible callsites that may correspond to JNI function calls. However, it is harmless to perform this optimization on some callsites and decline to perform it on others. Furthermore, most JNI code that we have observed does not tend to copy the `JNIEnv` pointer to other places. Therefore, this optimization can be quite effective without following all possible references to the `JNIEnv` pointer; it can catch a sufficient number to be useful and causes no harm by ignoring the rest.

4.3.3 JNI use/def Analysis

Once callbacks have been identified, a pass of **JNI use/def** analysis is performed. The idea is that objects that are passed to inlined natives are tracked to the points where they are passed as arguments to callbacks. In general, each argument is represented by a set of possible objects as dictated by the control flow in the native method. The use/def analysis computes all possible classes, fields, and methods reaching uses in callbacks. When the analysis cannot conclusively determine the class that an object *must* be an instance of, it produces sufficient information to allow the transformation phase to consider all possible classes that the object *may* be an instance of. It is possible, however, that the analysis is

unable to compute even conditional results if, for example, arguments to a callback are fetched from storage.

During JNI use/def analysis, the results of *GetObjectClass*, *GetSuperClass*, *FindClass*, *GetMethodID* and *GetFieldID* are treated as definitions and their uses are tracked. The results of these methods can normally be transformed into compile-time constants which are used instead of the usual constant pool indices. The use/def analysis also tracks the string arguments to *FindClass*, *GetMethodID*, and *GetFieldID* and by doing so, TR may positively resolve some of these calls while a more naive implementation would be unable to do so.

4.3.4 Transformations Using Use/Def Results

Once JNI use/def analysis is complete, the procedure continues by iterating over the list of identified callbacks and attempting to transform them to compile-time constant values or new TR-IL that is semantically equivalent and much less expensive. Depending on the type of callback, the following transformation outcomes are possible:

- If all of the possible definitions reaching a *GetObjectClass* are of the same class, the call is replaced by an appropriate constant.
- If all possible classes reaching a *GetFieldID* or *GetMethodID* are compatible and the string arguments can be uniquely determined, the call is replaced by an appropriate constant.
- If all possible field ids reaching a *Get<type>Field* or a *Put<type>Field* are the same and all possible objects reaching the call are of compatible class types, the call is replaced by a new, simpler sequence of TR-IL. More generally, if the offset of the data member from the beginning of the object is the same for all possible types that can reach the call, then the same code can be used for all the objects, allowing the callback to be "strength reduced". Note that this form of IL defers throwing exceptions in accordance with the Java rules for executing native methods.
- Similar transformations are performed for the various *Call<type>Method* callbacks by replacing the existing IL with new IL that makes direct calls to the function.

Any of the identified callbacks that are not handled by the steps above are treated as an ordinary call to an appropriate VM service routine via a synthesized function. For any of these callbacks, if the use/def analysis produces known but inconclusive information, conditional logic may be inserted along with the appropriate IL that represents the semantics of the callback being transformed. When the transformed callback is executed, appropriate behavior can be selected based on actual values. Runtime safety checks on arguments to callbacks can be performed as part of the callback transformation process.

5. CURRENT STATUS & RESULTS

In this section we report on the current implementation status of the design described in Section 4, and describe encouraging preliminary results.

5.1 Implementation Status

We have completed the W-Code conversion engine, and have added support to the TR inliner for inlining calls between native functions. We thus have a fully-functional Java JIT compiler that can be substituted as a back-end for the various W-Code generating static front-ends. The correctness of our implementation has been verified by successfully compiling all of C benchmarks from SpecCPU2000 [32], as well as standard C conformance tests. We have also compiled these benchmark programs with native-side inlining enabled and have observed the expected performance increases.

Two significant changes made to the TR JIT to support W-Code-based languages include extending its data type set to include unsigned types (since it was originally designed as Java JIT compiler), as well as modifying some of the alias-analysis-dependent optimizations since aliasing in Java is much simpler than in C. As noted in Section 4.1, alias information for the native code is explicit in the W-Code IL, and is preserved during the transformation to TR-IL.

We have extended the TR inliner to include support for inlining a restricted set of callouts to native functions. The mechanism for synthesizing calls to opaque native functions described in Section 4.2 has yet to be implemented, but we expect to be able to leverage the existing Direct2JNI functionality for this purpose. At present, however, we can recursively inline native functions defined in the same module and have verified the feasibility of transforming callbacks into cheaper direct references to JVM internals, by applying the transformation operations to known callback functions. We have not yet built the generalized shape-matching or the use/def analysis required to automatically detect and transform all callbacks. Non-transformable inlined callbacks, however, are successfully handled via calls to synthesized functions, and we have been successful in linking globally-declared data in native code to Java code.

For the purposes of generating a proof-of-concept prototype, we have further restricted the features of Java that we consider. We are currently ignoring string parameter access, reference creation, exception handling, the Reflection-related functions, monitors and the Invocation API. Although these are important features that must be handled in a full implementation, we believe this is a matter of engineering, and one that will not substantially alter the applicability of our native function inlining optimization.

5.2 Evaluation Methodology

Native inlining is an optimization that interacts with the performance dynamics of our JIT compiler, as well as with the running Java program making native function calls via the JNI. As with any JIT optimization, the runtime cost of performing the inlining and transformation must be balanced against the expected benefit of removing overhead and exposing more IL to the JIT optimizer. Ultimately, we believe the true power of this approach lies in the ability to treat native and Java code together during JIT compilation, particularly since we have the opportunity to eliminate pessimistic assumptions that the optimizer must make in the presence of opaque calls. In this paper, however, we focus on the cost of converting native functions from W-Code IL into TR-IL, on the benefit of eliminating callout/return overhead and transforming heavyweight callback operations into substantially cheaper operations. We also evaluate the runtime

Benchmark	Total W-Code Opcodes	Total Time (ms)	Time per opcode (μs)
bzip2	15383	78.277	5.09
crafty	84693	466.952	5.51
gap	336466	1797.185	5.34
gcc	133506	663.246	4.97
gzip	25469	139.263	5.47
mcf	5615	25.431	4.53
parser	48411	256.472	5.30
perlbmk	279196	1596.122	5.72
twolf	105027	547.702	5.21
vortex	193413	1091.121	5.64
vpr	56756	310.426	5.46

Table 1: Cost of W-Code to TR-IL conversion for SPECint 2000 C benchmarks

savings due to our optimization for a microbenchmark that performs data transfers between Java and C similar to those required for JDBC™ [36].

We evaluate critical aspects of our proposed system using microbenchmarks. All our timing measurements are performed on an IBM 7038-6M2 with eight 1.4 GHz POWER4™ CPUs. In each case, we measure the time to perform 100 million calls and divide to obtain the reported per-call execution time. Unless otherwise stated, no additional optimization was performed on the inlined native IL to highlight the impact of removing callout and callback overheads.

To validate the applicability of native inlining and callback transformations on real-world code, we profiled a run of SPEC JAppServer2004 using IBM Websphere® Application Server 6.0. We found that 4.07% of all function calls made during the run were native calls to 71 unique native functions, accounting for roughly 23% of the running time. Of these, 19 unique native functions were called at least 5000 times, and out of those, six were called at least 50,000 times. A single native function, *Object.hashCode()*, was called more than 300,000 times. This suggests that the runtime cost of inlining can be amortized over a large number of uses for important native functions. If the native function is well-understood by the compiler, semantic expansion [37] or a related inlining technique could be used to create a special-case version. This approach, however, is less general than our technique.

5.3 Microbenchmark Results

Using a series of small test cases, we evaluate our prototype in terms of the cost of IL conversion, and the benefits of native inlining and callback transformation.

5.3.1 Cost of IL Conversion

To evaluate the cost of converting from W-Code to TR-IL for C functions, we measured the time to convert the SPECint 2000 C benchmarks (eon is omitted because it is in C++). Table 1 shows the results. We report the total number of opcodes converted, the total time for the conversion, and the average time per opcode for each benchmark. Overall, we find that the cost per opcode converted is small (averaging just 5.3 microseconds), and reasonably similar across benchmarks. These results are encouraging, as they suggest a simple heuristic should be able to estimate the cost of converting a given native function at runtime based on its size in W-Code opcodes. Further, note that the cost of conversion only needs to be paid once, when the function is inlined, whereas the benefits of removing callout overhead will be obtained on every subsequent use of the inlined code.

Microbenchmark Test	Without Native Inlining (ns)	With Native Inlining (ns)	Speedup (X)
instance			
0 args	423	0	∞
1 args	458	0	∞
3 args	490	0	∞
5 args	579	0	∞
hash	535	97	5.5
static			
0 args	128	0	∞
1 args	137	0	∞
3 args	138	0	∞
5 args	143	0	∞
hash	176	96	1.8

Table 2: Microbenchmark runtimes and improvements with native inlining

5.3.2 Impact of Inlining Callouts

We implemented a series of microbenchmark tests that would show the overheads involved with performing callouts. The results are shown in Table 2. Instance and static natives were implemented with varying numbers of parameters, and then JIT-compiled with and without the native inlining optimization. For all static versions without native inlining, *Direct2JNI* was used to create compiled glue code for the native call, as described in Section 3.1. The benefits of Direct2JNI can be observed by contrasting the runtimes of the static tests against the instance ones. In all but *0 args* and *hash*, the native method bodies simply return one of the passed arguments. These tests show the incremental cost of passing arguments to native functions. In most cases, the resulting code after inlining performs so little work that it is below the resolution of the timers (reported as 0 in Table 2).

For native functions that contain real code, the benefits of inlining alone will clearly depend on amount of time spent executing native code in the function. Examples could easily be constructed showing speedups that range from effectively infinite (as for the *0 args* function call) to effectively 0 (for very long-running native functions). To see the benefits that might occur in one realistic use of native functions, we wrote a native hash function[3]. Inlining our instance hash function gives a speedup of 5.5, while a static version gives a speedup of 1.8.

These results show that for small native functions, removing the overhead of callouts can be a significant benefit. The primary motivation for inlining native code, however, is to create the opportunity to transform the much more expensive callbacks. We now consider the effect of these transformations.

5.3.3 Impact of Transforming Callbacks

We also implemented a series of microbenchmark tests

[3]we based this hash function on Wang's 32-bit mix function at http://www.concentric.net/ Ttwang/tech/inthash.htm

Microbenchmark Test	Without Native Inlining (ns)	With Native Inlining (ns)	Speedup (X)
instance			
GIntField	2560	0	∞
SIntField	2310	0	∞
CVoidMethod	2630	204	12.9
static			
GStaticIntField	2190	0	∞
SStaticIntField	2140	0	∞
CStaticVoidMethod	2520	214	11.8
GArrayLength	5640	60	93.4

Table 3: Microbenchmark runtimes and improvements with native inlining and callback transformations

Array Length	Without Native Inlining (ns)	With Native Inlining (ns)	Speedup (X)
1	586	2.4	244.1
10	597	20.7	28.9
100	1010	85.5	11.8
1000	4540	600	7.6

Table 4: Moving data from Java to C - runtimes and improvements with native inlining and callback transformations

that would show the overheads involved with performing callbacks in native code, and the benefits of transforming these callbacks in inlined natives. Instance and static native functions consisting of a series of callbacks were JIT-compiled with and without the native inlining and transformation optimization. For example, the CVoidMethod native code calls *GetObjectClass* (which is transformed to a compile-time constant), *GetMethodID*, (also a compile-time constant) and finally, *CallVoidMethod* (which is transformed to a non-JNI-dependent virtual function call using the constants from the previous two transformations). The GArrayLength test creates an array of characters using *NewCharArray* and returns its length using *GetArrayLength*. The native code in this last example is transformed into the equivalent TR-IL. The results are shown in Table 3. Because callbacks are more expensive than callouts, we see that the benefit of transforming them is correspondingly greater, with a *minimum* achieved speedup of nearly 12X in our test cases.

The final set of microbenchmark tests that we present deal with passing integer array data from Java to native code, an operation that is commonplace in native codes used by JDBC drivers. In these experiments, a single callback is used to obtain the entire array, similar to the "coarse-grained" strategy of JNIbench [1]. Table 4 displays the speedups obtained by transforming inlined *GetIntArrayRegion* callbacks for each of the varying array lengths, which vary from a factor of 244 for a single element array, to a factor of 7.6 for a 1000-element array. As expected, these speedups decrease for larger array sizes because the overhead in performing the callout and callback shrinks relative to the actual work done in transferring the array.

6. RELATED WORK

This section describes previous research on language interoperability, as well as work on optimizing Java native functions and the codes they call.

Examples of **language interoperability frameworks** that operate across languages, processes and machine boundaries include CORBA [30], Remote Procedure Calls [7] and the Component Object Model [23]. These frameworks use interface definition languages (IDLs) to specify common types, and depend on proxy stubs to help clients translate between machine architectures, execution models and programming languages. A more recent advance in language interoperability is Microsoft .NET [15]. In order for .NET programs to be interoperable, programs must adhere to the Common Language Specification (CLS), a subset of the Common Language Runtime (CLR) environment. The CLS most noticeably sets restrictions on the data types that can be used, confines interoperability to .NET-labeled languages, and excludes C or C++.

Programmer-based optimizations put the onus on the application programmer to practice efficient coding techniques when writing native code that uses the JNI. A former IBM developerWorks® article [19] advised batching native calls and passing as much data as possible per native call, as well as a number of other recommendations to amortize overhead. Although this article is no longer available through developerWorks, the recommended JNI programming practices are still valid. Similarly, the JNI specification [22] provides a set of "critical" functions that may return direct references to JVM data and objects and suggests ways to avoid making JNI callbacks, including caching field and method IDs during static initialization of classes. By reducing the overhead of the JNI automatically, our approach obviates these programming practices, removing the added burden from the application programmer.

Restricting functionality that can be offered in native code is another approach to reduce overhead and minimize the dependence on JNI callbacks. For example, the Intel ORP [10] supports a "direct call" mechanism that bypasses the construction of special wrapper codes that would otherwise be found preceding native call frames on the call stack. The speedup that results from not having to perform maintenance work in the wrappers comes at the expense of not being able to unwind the stack. Therefore, direct calls can only be used for methods that are guaranteed not to require garbage collection, exception handling, synchronization or any type of security support. Bacon [3] has implemented a JVM-specific JNI "trap-door", simplifying reference management for garbage collection, based on the observation that his native code only accesses parameters and never performs any JNI callbacks. While these strategies can improve performance in certain circumstances, they are not a general solution and cannot be used for most existing JNI code.

Proprietary native interfaces that are used by various VMs take advantage of knowing the internals of the VM, and therefore help mitigate the overheads of native calls by tightly coupling them to the VM. Examples include the PERC Native Interface for the PERC VM [26], the Jcc optimizing compiler [34], Microsoft's now-supplanted Raw Native Interface (RNI) [11], and according to [25], the original but now deprecated Native Method Interface (NMI). All of these approaches closely couple the interface with the specific virtual machine, and thus seriously restrict porta-

bility. The JNI, in contrast, is a cleaner and more portable solution because all JVM internals are represented in an opaque manner and can only be accessed via JNI callbacks. This, ironically, lies at the heart of JNI-related overheads. Our work is independent of the JVM being used and our technique can be utilized by those who want to support it.

A different approach involves extending the JVM to support features for which native functions are commonly used. One example is incorporating **unmanaged memory** into the JVM. The provision of high-speed access to unmanaged memory can be used to implement shared memory segments, memory-mapped files, communication and I/O buffers, and even memory-mapped hardware devices. Jaguar [35] implements Bytecode-to-assembly code mappings in a JIT compiler to generate inlined assembly code for limited sequences of machine code. The main use of the mappings is to map object fields to memory outside the managed Java heap. The benefit of this approach is the near-C performance obtained for various latency and bandwidth simulations. However, there are a number of limitations with this approach, including the inability to recognize and map long, complex sequences of Bytecodes, as well as the inability to apply mappings to virtual method invocations, since code mappings can't handle runtime class loading. Buffers in the Java new I/O libraries [18] are also allocated outside the garbage-collected heap, and can be accessed by the JVM without having to perform any time-consuming copy operations. Our technique is orthogonal to the work on unmanaged memory.

Optimizations that target native functions and the JNI specifically include IBM's enhancements to the Java 2 Platform mentioned in [19] and inlining of helper code that sets up JNI stack frames as mentioned in [10]. IBM also reuses existing Java stack frames to reduce the native stack frame setup overhead. Andrews' [1] suggestions include native memory mirroring as well as provisioning the JNI for lightweight calls. No known implementations of native function inlining exist, but have been referred to by Andrews [1] and by Liang [22] as a powerful yet difficult-to-implement optimization. Our solution demonstrates that native function inlining is feasible with a JIT compiler, and that the benefits of removing overhead alone may make it worthwhile. Further, it enables more aggressive optimizations, similar to traditional inlining techniques.

7. FUTURE WORK

There are some issues that make our proof of concept harder to generalize. Native callsite polymorphism, including overriding and overloading of natives, as well as synchronized natives are outside the scope of our immediate work. Other engineering issues we currently ignore include inlining native code located in modules external to the one containing the original native callout.

The pattern matching basis of our callback transformation algorithm may be too strict to handle all types of native code, and we currently only analyze a small subset of JNI callbacks. We are working on completing the implementation of the design described in Section 4, which will allow us to explore questions of when the native inlining strategy should be performed, and when the existing native call should be left alone. We also plan to evaluate the impact of exposing native code to the TR optimizer, and study the end-to-end effect on realistic benchmarks.

Although a JIT compiler is unlikely to be able to compete with a static native code optimizer, the W-Code IL stored alongside our native binaries is the output of a sophisticated interprocedural optimizer and loop transformer. This provides the TR JIT with some of the benefits of static analysis that could not be contained in the compile-time budget of a dynamic compiler. We also expect that some of the runtime information unavailable to static optimizers will help further improve the quality of the inlined native code. Currently, we aren't overly concerned with modifying the TR JIT inliner's heuristics, except for some fine-tuning that recognizes a number of differences between Java and non-Java functions, including their sizes, variable length parameter lists and parameters whose addresses are taken. Studying suitable heuristics for native inlining is a subject for future work.

8. CONCLUSION

We have shown that native function inlining can greatly reduce the overheads normally associated with Java native function calls and callbacks. One key component of our strategy is an IL conversion engine that allows code written in low-level languages to be expressed in the same IL used by the TR JIT for Java code. This facilitates the inlining of native function calls defined in native libraries that include the IL of a static optimizing compiler, and the further transformation of callbacks. One of the key benefits of our strategy is that the full JNI API is maintained while giving the performance characteristics of direct access to Java objects from C. Our goal for this technique is both to improve the performance of applications that make extensive use of the JNI, and to remove the need for programming practices designed to circumvent the currently-heavy penalty of using the JNI.

9. ACKNOWLEDGMENTS

This research is supported by grants from the IBM Centre for Advanced Studies, CITO, and NSERC. It would also not have been possible without the support of the TR JIT compiler team at the IBM Toronto Software Lab.

10. REFERENCES

[1] Jack Andrews. Interfacing Java with native code: Performance limits. ITtoolbox for Java Technologies Knowledge Base web site, Peer Publishing section. `http://java.ittoolbox.com/documents/document.asp?i=780#`, 2000. Also available at `http://www.str.com.au/jnibench`.

[2] Andrew Ayers, Stuart de Jong, John Peyton, and Richard Schooler. Scalable cross-module optimization. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 301–312. ACM Press, 1998.

[3] David F. Bacon. JaLA: A Java package for linear algebra. Presented at the Computer Science Division, University of California, Berkeley, 1998. IBM T.J. Watson Research Center.

[4] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Xinying Li. mpiJava: A Java interface to MPI. In *Proceedings of the First UK Workshop, Java for High Performance Network Computing at EuroPar*, Southampton, UK, September 1998.

[5] Paolo Bellavista, Antonio Corradi, and Cesare Stefanelli. How to Monitor and Control Resource Usage in Mobile Agent Systems. In *Proceedings of the Third IEEE International Symposium on Distributed Objects and Applications*, pages 65–75, Rome, Italy, September 17–20 2001.

[6] Aart J. C. Bik and Dennis Gannon. A Note on Native Level 1 BLAS in Java. *Concurrency: Practice and Experience*, 9(11):1091–1099, 1997.

[7] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[8] Per Bothner. Java/C++ integration - writing native Java methods in natural C++. `http://gcc.gnu.org/java/papers/native++.html`, November 1997.

[9] Fabian Breg and Constantine D. Polychronopoulos. Java virtual machine support for object serialization. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, pages 173–180, Palo Alto, California, June 2–4 2001.

[10] Michal Cierniak, Marsha Eng, Neal Glew, Brian Lewis, and James Stichnoth. The Open Runtime Platform: A Flexible High-Performance Managed Runtime Environment. *Intel Technology Journal*, 7(1), February 2003.

[11] Bruce Eckel. *Thinking in Java*. Prentice-Hall, 1st edition, 1998.

[12] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgard, and David Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSN-TR-99-33, Microsoft Inc., June 16 1999.

[13] Michael Franz and Thomas Kistler. Slim binaries. *Communications of the ACM*, 40(12):87–94, December 1997.

[14] Vladimira Getov, Susan Flynn Hummel, and Sava Mintchev. High-performance parallel programming in Java: exploiting native libraries. *Concurrency: Practice and Experience*, 10(11–13):863–872, 1998.

[15] Andrew D. Gordon and Don Syme. Typing a multi-language intermediate code. *ACM SIGPLAN Notices*, 36(3):248–260, March 2001.

[16] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[17] Bjørn-Ove Heimsund. Native Numerical Interface (NNI). `http://www.math.uib.no/~bjornoh/mtj/nni/`, November 2004.

[18] Ron Hitchens. *Java NIO*. O'Reilly and Associates, Inc., August 2002.

[19] IBM Corporation. IBM rewrites the book on Java performance. `http://www.developer.ibm.com/java/j2/j2perfpaper.html`.

[20] IBM Corporation. XL FORTRAN: Eight ways to boost performance. White Paper, 2000.

[21] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization*, pages 75–87, San Jose, California, March 20–24 2004.

[22] Sheng Liang. *The Java Native Interface. Programmer's Guide and Specification*. Addison-Wesley, 1999.

[23] Microsoft Inc. Com: Component object model technologies. `http://www.microsoft.com/com/default.mspx`.

[24] Michael Lazar Milvich. JavaCAVE: A 3D immersive environment in Java. Master's thesis, Montana State University, July 13 2004.

[25] Paul M. Murray, Todd Smith, Suresh Srinivas, and Mattias Jacob. Performance issues for multi-language Java applications. In *Proceedings of the 15 International Parallel and Distributed Processing Symposium 2000 Workshops*, volume 1800 of *Lecture Notes in Computer Science*, pages 544–551, Cancun, Mexico, May 1–5 2000. Springer.

[26] NewMonics Inc. Best practices for native code integration with perc. `http://www.newmonics.com/perceval/native_whitepaper.shtml`, February 26 2003.

[27] Steve Northover. SWT: The Standard Widget Toolkit, Part 1: Implementation Strategy for Java™ Natives. `http://www.eclipse.org/articles/Article-SWT-Design-1/SWT-Design-1.html`, March 2001.

[28] Hirotaka Ogawa, Kouya Shimura, Satoshi Matsuoka, Fuyuhiko Maruyama, Yukihiko Sohda, and Yasunori Kimura. OpenJIT: An Open-Ended, Reflective JIT Compiler Framework For Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 362–387, Sophia Antipolis and Cannes, France, June 12–16 2000. Springer.

[29] Vladimir Roubtsov. Profiling cpu usage from within a Java application. `http://www.javaworld.com/javaworld/javaqa/2002-11/01-qa-1108-cpu.html`, November 2002.

[30] Todd Scallan. a corba primer. `http://www.omg.org/news/whitepapers/seguecorba.pdf`, June 3 2002.

[31] Davanum Srinivas. Java tip 86: Support native rendering in jdk 1.3. `http://www.javaworld.com/javaworld/javatips/jw-javatip86.html`.

[32] Standard Performance Evaluation Corporation. SPEC CPU2000 V1.2. `http://www.spec.org/cpu2000`.

[33] Vaidy Sunderam and Dawid Kurzyniec. Efficient cooperation between Java and native codes – JNI performance benchmark. In *Proceedings of the 2001 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, June 25–28 2001.

[34] Ronald Veldema. Jcc, a native Java compiler. Master's thesis, Vrije Universiteit, Amsterdam, August 1998.

[35] Matt Welsh and David Culler. Jaguar: Enabling efficient communication and I/O in Java. *Concurrency: Practice and Experience*, 12(7):519–538, May 2000.

[36] Seth White, Maydene Fisher, Rick Cattell, Graham Hamilton, and Mark Hapner. *JDBC™ API Tutorial and Reference: Universal Data Access for the Java™ 2 Platform (2nd Edition)*. Pearson Education, June 1999.

[37] Peng Wu, Samuel P. Midkiff, Jose E. Moreira, and Manish Gupta. Efficient support for complex numbers in java. In *Proceedings of the ACM 1999 Java Grande Conference*, pages 109–118, San Francisco, California, June 1999.