`perljvm`: Using B to Facilitate a Perl Port To the Java Virtual Machine

Bradley M. Kuhn
Department of Electrical and Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45219
bkuhn@ebb.org
http://www.ebb.org/bkuhn

Abstract

perljvm is poised to be a functional JVM port of Perl. In designing *perljvm*, various ports of other languages to the JVM were studied and classified. Using this data, it was determined that Perl's `B` (back-end) compiler modules provided the best path to accomplish a JVM port of Perl. Since the `B` modules provide access to to the internal representation (IR) used by the perl back-end, *perljvm* concerns itself only with translating that IR into JVM assembler, using the Jasmin assembler syntax. A JVM assembler code emitter module was developed to minimize *perljvm*'s dependence on Jasmin. To support Perl's native data types on the JVM, equivalents for native Perl data structures must be designed and built in Java. Once *perljvm* is stable and most common Perl features are supported, work can begin on using the JVM as an object model to integrate Perl and Java objects seamlessly. At that time, integration with the JPL will be feasible.

Introduction

Perl is one of the most ported languages available for modern computing. However, Perl has yet to be ported to the Java Virtual Machine (JVM) architecture. Such a port of Perl is useful for the following reasons:

- Many other very high level languages, including Ada, Eiffel, Java, Python, Scheme, and Tcl have working JVM ports. Perl must keep pace with its peers.
- Hardware systems with embedded JVM processors, although rare today, continue to be the focus of many research projects [7, 13, 16]. If and when such systems gain popularity, Perl must be ready to support such systems. Indeed, such systems may be Perl's best shot at embedded computing.
- Software systems (such as web browsers and the JEmacs project [4]) with embedded JVMs are common. Programmers who target applications for these systems have so far been an untapped user base for Perl.
- Perl has yet to be ported to a system without a C compiler. Upon completion, a Perl port to the JVM will provide a useful case study of porting Perl to architectures that lack a C compiler. Such a case study will remain useful even if a C compiler is later ported the JVM.

Thus, a Perl port to the JVM is useful on many different fronts. Even considering the worst case scenario, where the JVM architecture ceases to be used in the field, the Perl port the JVM is still useful as a case study.

The rest of this paper is organized as follows: First, the Java-Perl Lingo (JPL) is discussed and considered. Next, various different approaches used to port other languages to the JVM in the past are discussed. Each approach is analyzed for its appropriateness for a Perl port to the JVM, and advantages and disadvantages for each approach are discussed. Next, the actual approach taken by *perljvm*, a hybrid of previous approaches, is considered. Following that, details concerning the actual implementation of *perljvm*, including various hurdles that were faced, are presented. Finally, possible future directions for *perljvm* are speculated.

What the JPL Taught Us

The Java-Perl Lingo (JPL) is a system that is part of the core *perl* distribution that eases the integration of Java and Perl. As Larry Wall frequently points out, it shows that Java's and Perl's semantics are interoperable [17]. The JPL is useful to programmers who have access to both *perl* and a Java environment and want to write some Java methods in Perl, instantiate Perl objects in Java, and/or instantiate Java objects in Perl.

This magic is accomplished through the Java Native Interface (JNI). This package allows a programmer to create Java classes and methods in C. The developers of JPL used JNI to interface *perl* (which is written in C) with the Java environment, providing a number of wonderful features.

However, such a solution can never be developed into a full Perl port to the JVM. The JNI is usually not available on JVMs that are embedded in hardware or other software programs. Thus, while the JPL is a powerful tool for those who use Java and Perl on a system where Perl is already fully supported, it will never allow Perl to run on embedded JVMs.

In addition, JPL has some overhead. The running process of a JPL program must have an active instance of *libperl.so* to run the Perl code as well as an active instance of a JVM to run the Java code. A native Perl port to the JVM would run much faster, and would enable Perl to take advantage of advances in the state of the art of JVM development.

Finally, JPL and *perljvm* really have different scopes. The JPL seeks to maximize flexibility of transition between Java code and Perl code. Even though *perljvm* would seek to eventually provide that, that is only a small part of the picture. The goal of *perljvm* is to run nearly any arbitrary Perl program natively in the Java environment by generating a valid JVM class file that is the equivalent to the given Perl program.

Possible Approaches to a JVM Port

Traditionally, there have been four ways in which languages are ported to the JVM. They are as follows:

1. Implementation of a language interpreter in Java.
2. Compilation from the source language to Java source.
3. Direct compilation from the source language to JVM bytecode.
4. Mapping of language structures and idioms onto the JVM.

In the subsequent subsections, each of these approaches is considered in detail.

Implementation of a Language Interpreter in Java

For most languages, implementation of a language interpreter in Java is perhaps the most straight-forward method of porting that language to the JVM. This approach was used by the Tcl [11] and Python [9]

ports. Since there are number of different compilers that can convert Java source into JVM bytecode, this interpreter written in Java can run easily on the JVM.

When a program from the source language needs to be run on the JVM, this new interpreter must take as input the source program, as well as the input that the source program is expecting. An `eval` construct using these two input sets is then invoked, and in that manner, the source program is run.

This approach has a number of advantages. First, if the source language has a well-written specification, or is a language with few constructs, based around a single paradigm (e.g., an object-oriented paradigm), then implementing an interpreter for the language is often a simple matter of implementing the specification. Design issues are often already decided by the specification or by the paradigm, greatly easing the burden on the implementor.

A second advantage is that real-time, on-the-fly code evaluation (i.e., `eval($string)`) is always available. The Java program that implements the interpreter simply needs to instantiate a new instance of the interpreter, and feed it `$string` as input.

However, this approach has two disadvantages, one of which is particularly problematic for a Perl port. The first disadvantage is speed. Since hardware devices that have JVMs on a chip are still in the realm of research labs, not the mass-market, most JVM implementations are done in software. The JVM bytecodes are interpreted by this software. Thus as Per Bothner notes, "if your interpreter for language X is written in Java, which is in turn interpreted by a Java VM, then you get double interpretation overhead". Such a situation is unacceptable for Perl, which has always prided itself on speed.

Another disadvantage that might be acceptable for some languages, but is completely unacceptable for Perl is code divergence. If a language has a well-defined specification that describes precisely the syntax and semantics of the language, code divergence is not an issue. An implementation must adhere to the specification. However, it has often been noted in the Perl community that "the specification is the implementation". The community cannot tolerate divergent implementations. Indeed, much work in the mid-1990s was done to stop the divergence of the Microsoft and Unix-like Perl implementations. It would surely be a tragedy if a Perl port to the JVM went down the road of code divergence.

Thus, the only way to safely achieve an acceptable JVM port of Perl using this approach is to compile *perl*, the existing C implementation of Perl, with a C compiler targeted to JVM. Experimental compilers of this nature do exist [6], but they are far from ready for production. In addition, even such a port of Perl would undoubtedly be slower than any of the other approaches. Indeed, given the relatively large size of *perl*, such a port would most likely be completely inappropriate for JVM implementations embedded in small hardware devices or other software programs.

Therefore, simply waiting for a C compiler to be targeted to the JVM is not the best approach for porting Perl to the JVM. Other methods must be investigated and attempted.

Compilation from the Source Language to Java Source

Compilation of the a source language into Java source code is a possible approach for porting that source language to the JVM. As was mentioned in above, compilers that target Java source to the JVM are widely available. Thus, if, for every program in the source language, an equivalent program in Java could be constructed automatically, then the source language would be effectively ported to the JVM.

The only real advantage to this approach is that the porter need not be concerned with the inner workings of the JVM. This minor advantage does not outweigh the two grave disadvantages. First, the port becomes immediately susceptible to changes in the Java language and its accompanying class libraries, which are more subject to change than the JVM specification. Second, the Java source language is not as expressive as JVM bytecodes. Although Java source is very close to JVM bytecodes, there are constructs (such as `goto`) that exist on the JVM but do not exist in Java [3].

With these disadvantages and only one minor advantage, it is not surprising that there has yet to be any language that has been successfully ported to the JVM using this method.

Direct compilation from the Source Language to JVM Bytecode

Another approach for porting a language to the JVM is to provide a compiler that targets the source language to the JVM. This can be done either by writing a compiler from scratch (as was done for Scheme [3] and Java), or by retargeting an existing compiler to the JVM (as was done for Eiffel).

The former method will cause code divergence, which is appropriate for Scheme and Java, since these languages have detailed written specifications. As has been established, such code divergence is not reasonable for Perl. The later method of retargeting a compiler to the JVM is reasonable for Perl, yet there is a risk.

As we have seen with the JPL, a useful feature of a JVM port is to permit the source language and Java to communicate smoothly via the common architecture of the JVM. When the Eiffel compiler was retargeted to the JVM, the port was not "aware" of the JVM. Thus, it has been a difficult road to use Eiffel's JVM port to integrate Eiffel and Java, since the compiler treats JVM bytecode as just another assembler syntax—not as a rich object architecture. Plans to modify the Eiffel port to support integration with Java exist; however, the design Perl's port to the JVM must not inherently contain this limitation.

Mapping Language Idioms onto the JVM

The final approach for porting a language to the JVM is to map each language idiom onto the Java architecture. If all the language features have semantic equivalents in Java or on the JVM, a mapping can be done to allow the language to run on the JVM. ADA's port to the JVM relied heavily on this approach [8].

This method, of course, must usually be combined with some Java or JVM code generation to be completely successful. However, it is worth categorizing this approach separately, since when used with a retargeted compiler, the problem that the initial Eiffel port encountered is avoided.

The downside to this approach is that each language idiom *must* have an equivalent in Java or on the JVM. If more than a few such idioms do not have equivalents, then a programmer must construct such idioms. This is not unreasonable, but it does take time and effort.

perljvm: A Hybrid Approach

Given the varied approaches for porting a language to the JVM, choosing the right approach for Perl was a difficult task. More than any other language, Perl has always prided itself on minimizing the downsides for programmers. Since each approach to this port could have drawbacks, rigidly choosing just one approach seemed unnecessarily problematic.

The best solution for Perl was a hybrid approach that attempted to minimize the disadvantages of all these approaches while maximizing the advantages. Given the history of Perl culture, such a solution seemed quite appropriate. Perl itself is "happily derivative" and traces its etymological roots back to many different languages. Thus, it is not surprising that the best approach to a JVM port of Perl would draw on all the experiences of porting other languages to that architecture.

Thus, *perljvm* was born, a Perl to JVM compiler that draws on the various approaches of porting other languages to the JVM, leveraging the various strengths of the different approaches.

Leveraging of the Existing *perl* Implementation

It is a common misconception that *perl* is an interpreter for Perl. The misconception arises from the fact that *perl* has two components within the same actual binary. First, *perl* has a front-end compiler which

includes a lexer and parser that analyzes a Perl program and produces an intermediate representation (IR) of the program, in the form of a syntax tree [2].

Second, *perl* has a back-end, which includes an implementation of the native Perl data types (such as scalar, array, and hash), as well as the Perl Virtual Machine (PVM). The PVM can take the IR generated by *perl*'s front-end, along with the data type implementations, and evaluate the IR (i.e., execute the code given by the Perl programmer). Thus, *perl* is not an interpreter; it is actually a compiler and a virtual machine for Perl.

When seen in this fashion, the similarity between the *perl* environment and the Java environment are striking. However, there are some key differences, which are as follows:

- The JVM has a detailed written specification. The PVM is documented primarily only in the source code for *perl*.
- The JVM has very simple native data types, and relies on Java class libraries to provide more complex types. The PVM has a number of complex native data types (e.g., hash, scalar and array).
- JVM has fewer operation codes (opcodes) than the PVM. Indeed, the PVM has a separate opcode for nearly every Perl function in the *perlfunc(1)* manpage.
- Java compilers and JVMs are usually implemented separately. The PVM and the Perl compiler are tightly coupled inside *perl*.

Thus, it is not possible to simply "map" the PVM onto the JVM in any simple way. However, thanks to the `B` modules, *perljvm* can utilize *perl*'s existing front-end *and* much of the back-end to do the work of the port.

Perl's `B` modules allow the programmer implement their own back-ends separate from *perl*'s back-end. A module that is uses `B` has the opportunity to examine and manipulate the IR that was generated by *perl*'s front-end. In addition, `B` can be used to examine the internal data structures used by *perl*'s back-end.

On the command line, the user interfaces to these back-ends via the `O` module. The `O` module acts primarily as a wrapper, allowing the corresponding `B` module to be invoked. Thus, instead of running the "default" *perl* back-end, a completely different back-end, written in Perl, can be chosen. (Please see [5] and [10] for a more complete discussion of `B` and `O`.)

Taking advantage of this feature, the core of *perljvm* is implemented using `B`. Since all the facilities of *perl*'s front-end are provided, there is no need for *perljvm* to have its own lexer or parser for Perl. In addition, *perljvm* can use `B` to examine the IR, which is roughly "PVM code", plus references to Perl's native data structures.

Using Jasmin Assembler

Thanks to the `B` module, *perljvm* does not need to parse Perl, find syntax errors, generate an IR, nor do any front-end compiler work. That is already done by *perl*, and is completely accessible via the `B` module. With all front-end issues already solved, the next challenge is the creation of valid JVM class files.

The JVM file format is quite complex. Directly generating such a file from a `B` module would be tricky. There is no standard for assembler syntax for the JVM, so there are no tools in the standard Java environment to easily generate JVM class files. Early in the project, this was a focus of much attention.

However, Brian Jepson proposed that instead of generating the JVM class file directly, *perljvm* should instead generate output using the Jasmin assembler. Jasmin assembler is a syntax for writing JVM class files that is similar to assembler formats used for non-virtual architectures [15]. This solution greatly reduced the problem scope of *perljvm*. Mr. Jepson discusses this idea extensively in [10].

However, there was the concern that the Jasmin assembler format is not standardized; it is simply one possible format for JVM bytecode assembler. Indeed, other formats do exist and are in use. Therefore, it was imperative that *perljvm* rely on one particular assembler syntax as little as possible.

To alleviate this problem, the concept of JVM "bytecode emitters" was introduced. First, a virtual base class called `B::JVM::Emit` was created. All code that must emit Java bytecode uses the interface provided by `B::JVM::Emit`, and all subclasses of `B::JVM::Emit` must provide implementations of `B::JVM::Emit`'s interface specific to a given assembler syntax.

As an example, consider the following code. It creates a JVM class called `Foo`, with one static public method, `main`, whose body has a single JVM `dup` instruction.

```
my $emit = new B::JVM::Jasmin::Emit("Foo");

$emit->methodStart("main([Ljava/lang/String;)V", "static public");

$emit->dup("main([Ljava/lang/String;)V");
$emit->methodEnd("main([Ljava/lang/String;)V");
```

If a standard assembler format for the JVM is ever created, one needs only implement `B::JVM::StandardAssembler::E` as a subclass of `B::JVM::Emit`, and change the first line in the example above to:

```
my $emit = new B::JVM::StandardAssembler::Emit("Foo");
```

Assuming that `B::JVM::StandardAssembler::Emit` is implemented properly, the rest of the code will function properly, generating the `Foo` class.

Data Type Support

With full access to the *perl* front-end, the `B` modules to manipulate the IR, and a code emitter object for JVM bytecodes, most of the components for a Perl to JVM compiler are in place. However, recall that the IR generated by *perl* assumes that a PVM and implementations of Perl's native data types are available. To successfully port Perl to the JVM, the data types that Perl considers native must be available on the JVM.

One approach would be to "map" all of Perl's data types into equivalent data types already available for the JVM. Unfortunately, in most cases, this approach is not possible, since Perl's native data types are so unique. For example, at first glance, it might seem feasible to map Perl's hash into an object of type `java.util.Hashtable`. However, Java's hash tables do not understand the concept of `tie`. Similarly, scalars cannot be mapped onto `java.lang.String`, since scalars act like numbers when they are supposed to, and Java strings do not. The uniqueness and flexibility of Perl's data types, loved by Perl programmers everywhere, become the headache of the programmer who wants to port Perl to an architecture whose data types are not so unique and flexible.

Thus, for each data type that *perl*'s back-end considers "native", an equivalent class for it must exist on the JVM. Since the Java language easily compiles to the JVM in an idiomatic way, these classes are implemented in Java. Each class provides an interface that Perl expects for the data type, and since the implementation is in Java, the data type can run on the JVM.

As an example, consider the following portion of the class `SvBase`, which is an implementation of *perl*'s `SvNULL` [1]:

```
class SvBase implements Cloneable {
  boolean defined;

  SvBase() {
      defined = false;
  }
```

```
  boolean isDefined() {
      return defined;
  }

  void undef() {
      defined = false;
  }
  // [...]
}
```

Thus, using this class, the Perl program:

```
defined $bar;
```

could be compiled to the Jasmin equivalent:

```
.class public main
.super java/lang/Object
.method static public main([Ljava/lang/String;)V
.var 0 is foo LSvBase
new SvBase
dup
astore_0
dup
invokespecial SvBase/<init>()V
invokevirtual SvBase/defined()Z
```

The Java equivalent of that is as follows (*perljvm* does not actually translate to Java; the following code is provided for didactic purposes only):

```
class main {
  static public void main(String argv[]) {
    SvBase bar = new SvBase();
    bar.defined();
  }
}
```

This example gives the flavor of how the native Perl data types are implemented in Java to provide access to Perl data types on the JVM. An in-depth discussion of these Java classes will be available in [12].

Putting It All Together

The final step to achieve the the JVM port is to support the opcodes in the IR. In this area, the `B` module is most useful. *Perljvm* descends the syntax tree provided by the IR, in a depth first fashion. At each opcode, *perljvm* processes it using the emitter to generating Jasmin code. The emitted Jasmin code utilizes the various data type classes to perform the task the opcode would have performed had it been run on the PVM.

As an example, consider the follow code:

```
sub B::SVOP::JVMJasmin {
  my $op = shift;
```

```
    my $name = $op->name();
    # ...
    my $curMethod = # ....
    # ...
    if ($name eq "gvsv") {
      my $stashName = $op->gv->STASH->NAME();
      my $gvName    = $op->gv->NAME();

      $emit->getstatic($curMethod, "Stash/DEF_STASH", "LStash;");

      $emit->ldc($curMethod, cstring $stashName);
      $emit->invokevirtual($curMethod,
                           "findNamespace(Ljava/lang/String;)LStash;");

      $emit->ldc($curMethod, cstring $gvName);
      $emit->invokevirtual($curMethod,
                "Stash/findGV(Ljava/lang/String;)Linternals/GV;");
      $emit->invokevirtual($curMethod, "GV/getScalar()LScalar;");

    }
    # ...
  }
```

In this code segment, we see part of the subroutine, `B::SVOP::JVMJasmin`. The name indicates that it is the code for handling opcodes of type `SVOP` for the JVM port using Jasmin. `SVOP` is a name provided and required by the `B` module. The `JVMJasmin` portion of the name is provided by the user of `B`, and is usually given on the command line when using the **O** module [5, 10].

The first argument when opcode subroutines are invoked is the opcode object itself. Usually, as in this case, the `name()` method is called to decide how to handle the particular opcode.

In this example, the code for handling the `SVOP` named `gvsv` is shown. The `gvsv` opcode is used when a dynamically scoped variable is mentioned. This opcode must find the actual data of the variable by searching for it in the name space. To generate the equivalent Jasmin code for this opcode, the three Java classes `Stash`, `GV`, and `Scalar` must be used. These are equivalents to stashes, GVs and SVs in the *perl* core [1].

If the variable being looked for happens to be in the `$main::foo`, then the code above generates Jasmin assembler that looks something like this:

```
getstatic Stash/DEF_STASH  LStash;
ldc "main"
invokevirtual findNamespace(Ljava/lang/String;)LStash;
ldc "foo"
invokevirtual Stash/findGV(Ljava/lang/String;)Linternals/GV;
invokevirtual GV/getScalar()LScalar;
```

The Java equivalent of that is as follows (*perljvm* does not actually translate to Java; the following code is provided for didactic purposes only):

```
Stash.DEF_STASH.findNamespace("main").findGV("foo").getScalar();
```

If you compare this to the process described in [1] of how a stashes work inside *perl*, it is easy to see that this is equivalent code for a `gvsv` opcode (given that the `Stash` and `GV` Java classes do their jobs correctly!).

Thus, a programmer wishing to add support for new opcodes in *perljvm* goes through the following procedure:

1. Analyze the opcode in the *perl* back-end, and see if it uses any native data types that do not have any equivalent Java classes yet.

2. If such Java classes are needed, write and test them.

3. Add a method `B::OP_TYPE::JVMJasmin`, and write code to emit equivalent JVM assembler for the new opcode, utilizing the Java classes as necessary.

It should be noted that the approach taken for *perljvm* is not without trade-offs. Since *perljvm* provides no actual interpreter nor compiler for Perl on the JVM itself, constructs such as `eval($string)` will not be supported. For these cases, it is most reasonable to wait until C itself is retargeted completely to the JVM [6]. Once *perl* itself is available on the JVM, *perljvm* can invoke it only as a last resort, but continue to support the rest of Perl natively on the JVM for better performance.

Future Directions

The first goal of *perljvm* is to continue adding support for more opcodes and more of Perl's native data types. At the time of writing, scalars and related operations were supported robustly and completely, and support for arrays was beginning to take shape. However, the subset of Perl that *perljvm* supports continues to grow quickly, and assistance has been solicited from other developers in the Perl community. By the time of the The Perl Conference 4.0, there will hopefully be much more of Perl supported on the JVM.

Once a usable subset of Perl is supported, *perljvm* should be integrated with the JPL. Users should have the option to have either native JVM code generated from JPL programs (via *perljvm*), or have the JNI-based Perl/Java mix generated by the existing JPL implementation. This would allow users to use some existing JPL code on embedded JVMs, and on systems with working JVMs but without *libperl.so* around.

Integration with Per Bothner's Kawa system [3], and eventual JEmacs [4] support for Perl could be quite useful. Kawa is designed to give some support services on the JVM for non-Scheme languages, and such services might make *perljvm*'s job a little bit easier. In addition, if *perljvm* can work well with Kawa, then JEmacs can allow scripting in Perl.

As a generalization of Kawa integration, it might prove interesting to experiment with using the JVM as an object model to integrate Perl and Java objects seamlessly. Python already supports seamless communication with Java via Jpython, and it is hoped that similar support can be created for Perl. In addition, since so many languages have JVM ports, it might be possible to use the JVM to allow different languages to communicate, using the JVM as generalized object model. More work must be done to determine if such a powerful use of the JVM is feasible.

Finally, *perljvm* appears to be the first use of `B` that is not heavily tied to the *perl* core itself (as the `B::CC` is). Care has been taken to carefully document what each opcode does, and what it needs to know about Perl's native data types. As the project progresses, it may turn out that *perljvm* is useful as an independent documentation and "reverse engineering" of the PVM and Perl's native data types. This information might be useful in future Perl efforts, such as Topaz.

Availability

`Perljvm` is currently available as part of `B::JVM::Jasmin` on CPAN, and at *http://www.ebb.org/perljvm*. It is copyrighted by Bradley M. Kuhn, and is licensed under the same license as *perl* itself.

Acknowledgements

Mr. Kuhn would like to thank USENIX for a student scholarship and stipend to research `perljvm`.

Brian Jepson released an early prototype of `B::JVM::Jasmin`, the core module used by `perljvm`. Although the current `B::JVM::Jasmin` shares no code with Mr. Jepson's prototype, Mr. Jepson was the first to introduce the idea of using Jasmin to facilitate a port of Perl to the JVM. His help is greatly appreciated.

Finally, Mr. Kuhn thanks Matthew T. O'Connor, who has assisted in the implementation of *perljvm* since the earliest versions.

References

[1]

Aas, Gisle. "Perl Guts Illustrated, Version 0.09". [Online] Available at *http://gisle.aas.no/perl/illguts/*. November 1999.

[2]

Aho, Alfred V., et al. *Compilers: Principles, Techniques, and Tools* Addison-Wesley, Reading, Massachusetts, USA, first edition, March 1988.

[3]

Bothner, Per. "Kawa — Compiling Dynamic Languages to the Java VM". *USENIX 1998 Annual Technical Conference: Invited Talks and FREENIX Track*, pages 225–272. New Orleans, Louisiana, USA. June 1998.

[4]

Bothner, Per. "JEmacs–The Java/Scheme-based Emacs". *USENIX 2000 Annual Technical Conference: FREENIX Track*. To Appear. San Diego, CA, USA. June 2000.

[5]

Beattie, Malcolm. "The Perl Compiler". *The Perl Journal*. Volume 1, issue 2 (Summer 1996), pages 34−36.

[6]

Cifuentes, Cristina, et al. "UQBT — A Resourceable and Retargetable Binary Translator". [Online] Available at *http://archive.csee.uq.edu.au/~csmweb/uqbt.html#gcc-jvm*, March 2000.

[7]

Cladingboel, Chris. "Hardware Compilation and the Java Virtual Machine". [Online] Available at *http://www.wadham.ox.ac.uk/~chris/project*, July 1998.

[8]

Comar, Cyrille, et al. "Targeting GNAT to the Java Virtual Machine". In *Proceedings of the conference on TRI-Ada '97*, pages 149−161. 1997.

[9]

Hugunin, Jim. "JPython". [Online] Available at *http://www.jpython.org*, March 2000.

[10]

Jepson, Brian. "Taking Perl to the Java Virtual Machine". *The Perl Journal*. Volume 4, issue 4 (Winter 1999), pages 53−59.

[11]

Johnson, Ray. "Tcl and Java Integration". Technical Report, Sun Microsystems Laboratory, February 1998. [Online] Available at *http://www.scriptics.com/products/java/tcljava.pdf*.

[12]
Kuhn, Bradley M. "An Implementation of Native Perl Data Types in Java." In Preparation.

[13]
Sun Microsystems, Inc. "The K Virtual Machine (KVM) White Paper". [Online] Available at *http://java.sun.com/products/kvm/wp*. January 2000.

[14]
Lindholm, Tim and Yellin, Frank. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Massachusetts, USA, first edition, September 1996.

[15]
Meyer, Jon and Downing, Troy. *Java Virtual Machine*. O'Reilly and Associates, Sebastopol, CA, USA, first edition, March 1997.

[16]
Tremblay, Marc and O'Connor, Michael. "picoJava: A Hardware Implementation of the Java Virtual Machine". [Online] Available at *http://infopad.eecs.berkeley.edu/HotChips8/4.3*. October 1996.

[17]
Wall, Larry. Personal Communication. August 1998.