TJC : A Tcl to Java Compiler

Mo DeJong
Mo DeJong Consulting
mo@modejong.com

Abstract

A recent trend in corporate computing environments is the reimplementation of legacy systems on a Java based software platform. Significant long-term savings can be realized through use of a common software platform and tools, but the costs of software development and retraining of engineers remains high. If a legacy system can be ported to a Java environment without having to rewrite existing code, then most software development and retraining costs can be avoided. A large semiconductor manufacturer recently faced just such a challenge and decided to evaluate Jacl (Tcl interpreter written in Java) as a migration tool for a large legacy system implemented in Tcl. Jacl was found to be satisfactory in all areas except one, runtime execution speed. The native version of Tcl contains a runtime compiler and execution engine, while Jacl supports only interpreted execution. As a result, native Tcl executes code from 10 to 50 times faster than Jacl. This paper introduces TJC, a Tcl to Java compiler that converts Tcl procs into Java bytecode and closes this performance gap. In many cases, TJC compiled Tcl code executes more quickly than the same code running in native Tcl. This paper describes TJC's initial design and implementation, demonstrates code generated for simple examples, and describes optimizations added as the compiler has matured into production ready software. In addition, performance of TJC compiled code is compared to native Tcl, and to other scripting languages implemented in Java.

Keywords: Tcl, Java, JVM, Scripting, Compiler, Bytecode, Optimization

1 Design Goals

The TJC compiler was designed to optimize for runtime execution speed. Execution time required to compile Tcl procs, as well as runtime memory usage, were secondary considerations. The targeted execution environment was a server system running JDK 1.4, with sufficient memory and CPU resources to handle most tasks. Since compile time was a secondary concern, the compiler could make multiple passes and generate the fastest possible code for a specific usage. Compatibility with native Tcl was a design requirement and has been maintained in all areas except one. TJC will generate inlined logic for Tcl primitives like `set`, `if`, `for`, `lindex`, and others. The compiler assumes that the user will not redefine these built-in Tcl primitives at runtime.

Early on, three options were considered:

A) Duplicate native Tcl's bytecode compiler and execution engine

B) Duplicate native Tcl's bytecode compiler but emit Java bytecode

C) Design new Tcl compiler and emit Java source code

Option A was the most straightforward. Native Tcl contains a compiler that emits Tcl bytecode and a runtime execution engine that interprets Tcl bytecode instructions [1, 2]. Although porting native Tcl's compiler and execution engine to Java would take time, this approach involved few risks or unknowns. The problem with option A was that it was unclear if running a Tcl bytecode interpreter on top of the JVM would produce acceptable performance results.

Modern JVMs go to great lengths to execute Java bytecode efficiently, compiling to native machine code in many cases. Sun's Hotspot compiler is able to inline methods, predict branches, and dynamically recompile code based on actual usage. Unfortunately, these optimizations would be of little value to a Tcl bytecode interpreter loop implemented in Java. A significant percentage of an application's execution time could be spent just decoding Tcl bytecode instructions and then jumping into and out of code that implements specific instructions [3, 4, 5, 6, 7]. Option A was rejected on these grounds.

Option B was briefly considered as a solution to the Tcl bytecode execution issues described above. The existing compiler from native Tcl could be ported to Java, but it would be modified to emit Java bytecode equivalents for each Tcl bytecode instruction. This approach would avoid using CPU resources to decode Tcl bytecode instructions at runtime. The Hotspot compiler could then convert entire compiled command implementations to native code and aggressively inline utility methods. Java libraries that could significantly simplify this type of implementation are freely available [8, 9]. However, option B was rejected because it would be too complex to implement, debug, and modify.

Option C involves emitting Java source code and converting to Java bytecode using a Java compiler like `javac`. This approach would require quite a bit of implementation effort, as new compiler and runtime support modules would need to be implemented from scratch. This approach would also generate a relatively large number of Java bytecode instructions, as compared to the more compact Tcl bytecode instructions. Nonetheless, option C was chosen because it had a number of important advantages.

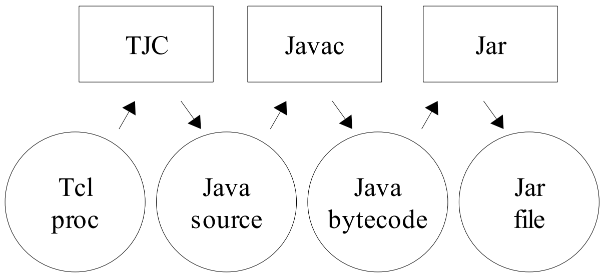
A new compiler could take advantage of optimizations that were not even considered in the native Tcl compiler. Java bytecode would be executed directly in the JVM, so there would be no runtime overhead associated with a Tcl instruction decode and execute loop. Option C would be easy to debug, since any Java source code debugger could be used to step through generated code. Each Tcl proc would be mapped to a Java class, each Tcl proc invocation would be mapped to a Java method invocation, and each Tcl variable frame would be mapped to a Java stack frame. In addition, Java profiling tools could be used to profile generated code in terms of time taken by each Tcl command.

Readability and transparency were important factors in choosing option C. Java source code emitted by the compiler would be human readable and regression testable. The importance of being able to easily understand compiler output cannot be understated. Many tricky problems were solved without the aide of a runtime or a debugger, simply by looking at the Tcl input and the Java output. Optimizations were similarly easy to visualize when working directly with emitted source code.

2 Dual Implementations

The TJC compiler supports two compilation modes. Batch mode is used to compile all the procs defined in a set of Tcl files into a single Jar file. Runtime mode is used to compile a specific Tcl proc into Java bytecode. Runtime mode consumes memory and CPU resources, so startup time could be affected. Batch mode does all compilation off-line, so only minimal CPU and memory resources are required at runtime. Batch mode is particularly useful for large libraries of Tcl code. Runtime mode is most useful when a small number of procs are to be compiled, or when the procs are not defined until runtime.

2.1 Batch Mode

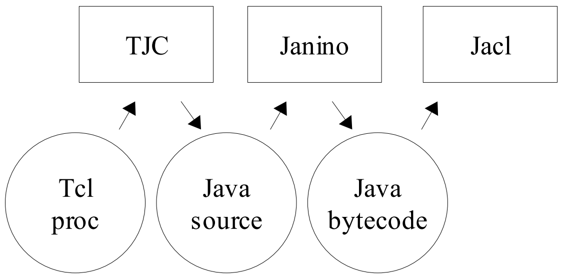


Batch mode is invoked from the command line via a script named `tjc`.

```
  $ tjc example.tjc
```

The invocation above will scan the Tcl files indicated in `example.tjc` and compile each statically defined proc. The compiler will generate a Java source file for each proc and then invoke `javac` to compile to Java bytecode. Finally, Java's `jar` tool is invoked to create a pair of Jar files. The `example.jar` file contains compiled Java class files. The `examplesrc.jar` file contains generated Java source files. A user would add both Jar files to the `CLASSPATH` and then start the Jacl shell.

2.2 Runtime Mode



In runtime mode, the compiler runs in a separate thread. Unlike native Tcl, procs are not automatically compiled when defined in Jacl. The user must request that a specific proc be compiled. First, the user loads the TJC package into Jacl, then the `TJC::compile` command is invoked to compile a proc.

```
  % proc hello {} {return "Hello World"}
  % package require TJC
  % TJC::compile hello
```

Once compilation is finished, the compiled implementation of `hello` replaces the original. Behind the scenes, TJC makes use of the Janino embedded Java compiler [10] to convert generated Java source into bytecode. Generated Java bytecode is then loaded into the current interpreter.

TJC is implemented entirely in Tcl code. Modules are organized as follows:

```
(Shared Modules)
  parse.tcl
  descend.tcl
  compileproc.tcl
  emitter.tcl

(Runtime Modules)
  tjcthread.tcl

(Batch Modules)
```

```
  nameproc.tcl
  parseproc.tcl
```

Since TJC is written in Tcl, one can use the compiler to compile itself and realize a significant performance improvement. TJC compiling itself in interpreted mode can take 25 minutes, but a compiled version of TJC can compile itself in 2.5 minutes.

3 Generated Code Examples

This section presents some simple Tcl procs and the Java code generated by TJC. Since generated code can be verbose, only the first example will include the full Java class output.

3.1 Hello World

Input:

```
proc hello {} {
  return "Hello World!"
}
```

Output:

```
// TJC implementation of hello
import tcl.lang.*;

public class HelloCmd
    extends TJC.CompiledCommand {
  public void cmdProc(
    Interp interp,
    TclObject[] objv)
      throws TclException
{
  if (!initCmd) { initCmd(interp); }
  CallFrame callFrame =
    TJC.pushLocalCallFrame(
      interp, wcmd.ns);
  try {
  if (objv.length != 1) {
    throw new TclNumArgsException(
      interp, 1, objv, "");
  }
  { // Invoke: return "Hello World!"
    interp.resetResult();
    interp.setResult(const0);
    if ( true ) { return; }
  } // End Invoke: return
  } catch (TclException te) {
    TJC.checkTclException(interp,
      te, "hello");
  } finally {
    TJC.popLocalCallFrame(interp,
      callFrame);
  }
}

  TclObject const0;

  protected void initConstants(
    Interp interp)
      throws TclException
  {
    const0 = TclString.newInstance(
      "Hello World!");
    const0.preserve();
    const0.preserve();
  }
} // end class HelloCmd
```

In this generated code, the `HelloCmd` extends `TJC.CompiledCommand` and implements the `Command` interface. Jacl will invoke the `cmdProc()` method when the `hello` command is called in the interpreter. Each class generated by TJC includes code to push a call frame and then pop it off the stack when the method is finished. The proc `hello` invokes just one Tcl command, it is translated into three Java statements in the emitted code. Note the statement `interp.setResult(const0)`, it sets the interpreter result to the constant string `"Hello World!"`. Finally, the code returns and the method completes normally.

3.2 Command Invocation

This example shows how a Tcl command would be invoked from a compiled proc. The output in this example includes just the generated code for the command invocation. The generated class defines three constant strings, one for each command argument.

Input:

```
proc foolen {} {
  string length "foo"
}
```

Output:

```
// Snippet of FoolenCmd.java

{ // Invoke: string length "foo"
  TclObject[] objv0 =
    TJC.grabObjv(interp, 3);
  try {
    TclObject tmp1;
    // Arg 0 constant: string
    tmp1 = const0;
    tmp1.preserve();
    objv0[0] = tmp1;
    // Arg 1 constant: length
    tmp1 = const1;
    tmp1.preserve();
    objv0[1] = tmp1;
    // Arg 2 constant: "foo"
    tmp1 = const2;
    tmp1.preserve();
    objv0[2] = tmp1;
    TJC.invoke(interp, null, objv0, 0);
  } finally {
    TJC.releaseObjvElems(interp,
      objv0, 3);
  }
} // End Invoke: string
```

The code above invokes the `string` Tcl command. The `TJC.invoke()` method calls the `cmdProc()` method in the `StringCmd` class, passing an array of `TclObject` arguments. Most of the code in this invocation is needed to populate the argument array and maintain Tcl's reference counting rules for `TclObject` arguments.

3.3 Inlined String Command

Most Tcl command invocations are implemented like the previous example, but TJC is able to emit inlined code for a number of built-in Tcl commands. This example shows an inlined call to Tcl's `string` command. Again, the output in this example includes just the generated code for the command invocation. The generated class defines just one constant, the string `"foo"`.

`Input:`

```
proc foolen {} {
  string length "foo"
}
```

`Output:`

```
// Snippet of FoolenCmd.java

{ // Invoke: string length "foo"
  int tmp0 = const0.toString().length();
  interp.setResult(tmp0);
} // End Invoke: string
```

This inlined `string` command is significantly less complex when compared to invoking the `string` command at runtime. The inlined code avoids allocating an array, populating the array, incrementing and decrementing ref counts, and array cleanup and release. The inlined code is efficient and is easily optimized by the Hotspot compiler.

3.4 Inlined If Command

This example shows how a Tcl `if` command is converted to inlined Java code and how a simple expression is evaluated.

`Input:`

```
proc iftrue {} {
  if {0 == 1} {
    # no-op
  }
}
```

`Output:`

```
// Snippet of IftrueCmd.java

{ // Invoke: if {0 == 1} ...
  // Binary operator: 0 == 1
  ExprValue tmp0 = new ExprValue();
  tmp0.setIntValue(0);
  ExprValue tmp1 = new ExprValue();
  tmp1.setIntValue(1);
  TJC.exprBinaryOperator(interp,
    TJC.EXPR_OP_EQUAL, tmp0, tmp1);
  // End Binary operator: ==
  boolean tmp2 =
    ( tmp0.getIntValue() != 0 );
  if ( tmp2 ) {
    interp.resetResult();
  } else {
    interp.resetResult();
  }
} // End Invoke: if
```

The `ExprValue` class manages the details of maintaining the expression result type and its value. The expression result is converted to a Java `boolean` and one of the branches of the `if` statement is taken. If this example had included Tcl commands in the `if` block, instead of a comment, then these commands would appear before the first `interp.resetResult()` call.

4 Optimizations

This section describes some of the most important optimizations implemented in TJC. These examples are simplified and make use pseudo-code, see [11] for detailed examples that include complete Java source code.

4.1 Shared Constants

Assume the following Tcl proc is defined.

```
proc hello {} {
  return "Hello World!"
}
```

If this proc was interpreted in Jacl, the `return` command would be parsed from a string:

```
"return \"Hello World!\""
```

Jacl would parse these word elements into an array and then lookup and invoke the `return` command. Pseudo-code for this command might look like:

```
TclObject[] objv = new TclObject[2];
objv[0] = TclString.newInstance(
  "return");
objv[1] = TclString.newInstance(
  "Hello World!");
TJC.invoke(interp, objv);
```

The first and most obvious optimization to apply here is to avoid allocating two new `TclObject` values each time the command is invoked. TJC creates a pool of shared constants and then uses these constants each time a value is accessed inside a compiled proc. Pseudo-code might look like:

```
TclObject[] objv = new TclObject[2];
objv[0] = const0;
objv[1] = const1;
TJC.invoke(interp, objv);
```

This example assumes that the constants have already been initialized, another method would be emitted to do that.

```
void initConstants() {
  const0 = TclString.newInstance(
    "return");
  const1 = TclString.newInstance(
    "Hello World!");
}
```

4.2 Cached Command Lookup

The next optimization that could be applied to the `hello` proc would be to cache a reference to the command, instead of looking it up by name for each invocation. TJC implements this optimization by passing a cached command reference when invoking a command. This reference would be defined as an instance variable in the generated class, so that it would be saved from one invocation to the next.

```
class HelloCmd {
  Command ccmd0 = null;

  ...
}
```

Then, method invocation code might look like:

```
TclObject[] objv = new TclObject[2];
objv[0] = const0;
objv[1] = const1;
if ( ccmd0 == null ) {
  ccmd0 = TJC.lookup("return");
}
TJC.invoke(interp, objv, ccmd0);
```

4.3 Inlined Commands

Tcl's `catch`, `expr`, `for`, `foreach`, `if`, `switch`, and `while` commands are special cases since these commands can contain other commands. When one of these commands is found, TJC will inline the command and any contained commands. TJC also includes inline support for other built-in Tcl commands, these are `append`, `break`, `continue`, `global`, `incr`, `lappend`, `lindex`, `list`, `llength`, `return`, and `set`.

4.4 Compiled Local Variables

Optimizing local variable access in a compiled proc is critical to efficient execution. Consider the following procedure.

```
proc setme {} {
  set i 0
  set j $i
}
```

The code above might be mapped to operations like:

```
setVar("i", const0);
setVar("j", getVar("i"));
```

In the pseudo-code above, the local variable `i` is accessed twice and `j` is accessed once. In interpreted mode, Jacl stores local variables in a hashtable. While this approach is flexible, it can quickly become a performance problem because of the sheer number of hashtable searches.

Compiled local variables avoid a hashtable search on each access by saving variables in an array. Each local variable name is associated with an integer array index. Pseudo-code to allocate such an array might look like:

```
Var[] compiledLocals = new Var[2];
compiledLocals[0] = new Var("i");
compiledLocals[1] = new Var("j");
```

Then, logic for the `setme` proc might look like:

```
setVar(compiledLocals[0], const0);
setVar(compiledLocals[1],
  getVar(compiledLocals[0]));
```

4.5 Omit Unused Results

A Tcl proc can return either an empty result or a specific value. If a proc does not return a value, then the result of the last command in the proc is returned as the result. Consider the following:

```
proc setme {} {
  set i 0
  set j 1
  set k 2
}
```

The result of executing this proc is 2. Pseudo-code might look like:

```
TclObject tmp;
tmp = setVar("i", const0);
setResult(tmp);
tmp = setVar("j", const1);
setResult(tmp);
tmp = setVar("k", const2);
setResult(tmp);
```

The result of this command can never be 0 or 1, so the compiler need only emit the third call to `setResult()`. TJC implements specific logic to detect when the result of a command is not used. The compiler can then omit pointless result set operations. With this optimization enabled, emitted code for the example above might look like:

```
setVar("i", const0);
setVar("j", const1);
TclObject tmp = setVar("k", const2);
setResult(tmp);
```

4.6 Expr Operations

The `expr` command and expression arguments to the `for`, `if`, and `while` commands are particularly important because expression evaluation can take up a large percentage of the total execution time. So, optimizing expression evaluation can have a significant performance impact. Consider the following example:

```
expr {!$v}
```

This expression consists of a unary not operator and a variable operand. Pseudo-code to evaluate this expression might look like:

```
{ // Invoke: expr {!$v}
  ExprValue tmp = new ExprValue();
  tmp.setValue( getVar("v") );
  TJC.exprUnaryOperator(interp,
    TJC.EXPR_OP_UNARY_NOT, tmp);
  TJC.exprSetResult(interp, tmp);
} // End Invoke: expr
```

Tcl's expression evaluation logic is tricky, the variable operand could contain an int, double, or string value. TJC handles each of these input types in the `exprUnaryOperator()` method. The logic above is less than optimal, since a new `ExprValue` object is allocated each time the expression is evaluated. TJC addresses this issue by allocating temporary `ExprValue` objects at the beginning of a compiled proc. Pseudo-code might look like:

```
(At the beginning of the method)

ExprValue tmp = new ExprValue();

...

{ // Invoke: expr {!$v}
  tmp.setValue( getVar("v") );
  TJC.exprUnaryOperator(interp,
    TJC.EXPR_OP_UNARY_NOT, tmp);
  TJC.exprSetResult(interp, tmp);
} // End Invoke: expr
```

This change might not seem like a big deal, but it can have a huge impact on performance. Consider the following loop, with an `expr` command in the body.

```
for {set v 0} {$v < 1000} {incr v} {
    expr {!$v}
}
```

Allocating an `ExprValue` at the beginning of the command moves the allocation outside of the loop. Instead of allocating and garbage collecting 1000 temporary `ExprValue` objects, a single object is allocated and reused. Allocating and garbage collecting lots of temporary objects is a serious performance problem, even for a modern JVM. This allocation change alone resulted in a 5x performance improvement for some examples.

TJC is able to use compile time type information to further optimize compiled expressions. Consider the `exprSetResult()` method, it sets the interp result based on the type of the passed in `ExprValue`. TJC knows that the result of a unary not operator is always an integer type, so int type logic from `exprSetResult()` can be inlined.

```
{ // Invoke: expr {!$v}
  tmp.setValue( getVar("v") );
  TJC.exprUnaryOperator(interp,
    TJC.EXPR_OP_UNARY_NOT, tmp);
  setResult( tmp.getIntValue() != 0 );
} // End Invoke: expr
```

In addition, TJC is able to inline logic from `exprUnaryOperator()`, to optimize the common case where the `TclObject` operand contains an integer value.

```
{ // Invoke: expr {!$v}
  TclObject otmp = getVar("v");
  if ( otmp.isIntType() ) {
    tmp.setIntValue(
      otmp.ivalue == 0 );
  } else {
    tmp.setValue( otmp );
    TJC.exprUnaryOperator(interp,
      TJC.EXPR_OP_UNARY_NOT, tmp);
  }
  setResult( tmp.getIntValue() != 0 );
} // End Invoke: expr
```

Applying these expression optimizations results in a significant improvement in runtime performance. The Hotspot compiler, particularly in the server configuration, does a very good job of optimizing the code above. In some cases, the code above runs only slightly slower than Java code using typed local variables.

5 Regression Testing

TJC has been successful in large part due to the effectiveness of the regression test suite designed and implemented along with the compiler. The regression test suite is called `tjcruntime`, it is available via CVS. Every Tcl language feature is extensively tested by the suite. Native Tcl includes a regression test suite, but it could not be used directly since TJC supports multiple compilation options that need to be tested individually. Instead, many tests from the native Tcl test suite were incorporated into `tjcruntime`.

When run, the `tjcruntime` test suite executes about 17,000 tests. There are 1,900 individual tests, each is compiled with 9 different option configurations.

6 TJC vs. Native Tcl

In this section a few simple Tcl procs are presented. Execution time for an interpreted version of each proc is compared to a TJC compiled version and to the execution time in native Tcl. These examples are very simple and are by no means a complete comparison. For a detailed, feature by feature comparison of TJC and native Tcl, see [12].

6.1 Int Sum Loop

```
proc isum { num } {
  set sum 0
  for {set i 0} {$i < $num} {incr i} {
    incr sum $i
  }
  return $sum
}

% isum 1000
499500
```

6.2 List Sum Loop

```
proc lsum { elems } {
  set sum 0
  foreach elem $elems {
    incr sum $elem
  }
  return $sum
}

set elems [list]
for {set i 0} {$i < 1000} {incr i} {
    lappend elems $i
}

% lsum $elems
499500
```

6.3 Call Command

```
proc caller {} {
  for {set i 0} {$i < 1000} {incr i} {
    callme "abcdefghijklmnop"
  }
}

proc callme { str } {
  return [string index $str 0]
}
```

6.4 Timing Results

Timing results are given in uSecs. JDK 1.4.2 (-server) and Tcl 8.4.12 (compiled with gcc -O2) were run under WinXP. Timing results are gathered after the Hotspot compiler has optimized the code.

	isum	**lsum**	**caller**
Jacl	7600	3750	26500
TJC/Jacl	125	96	400
Native Tcl	220	181	1250

7 Java Scripting Languages

In this section, timing results from some other scripting languages implemented in Java are compared to plain Java and to TJC. The `isum`, `lsum`, and `caller` examples from the previous section were coded in each language. Only the timing results (in uSecs) are presented here, see [13] for full source code.

	isum	**lsum**	**caller**
Java	6	12	25
TJC/Jacl	125	96	400
Jython	120	100	390
Pnuts	146	880	550
Groovy	2100	1250	2650
Beanshell	4087	9100	11200

Both Groovy and Beanshell lack the ability to compile to bytecode at runtime. Groovy (JSR-06) includes a batch mode compiler, but it generated classes that did not pass the bytecode verifier. An attempt to evaluate Jruby 0.9.0 was made, but it did not work with JDK 1.4. Both Jython and Pnuts contain a bytecode compiler and both turned in performance numbers comparable to TJC.

8 Future Directions

Performance of TJC compiled code is good, but could be improved significantly. Many additional optimizations have been identified, but they require time and funding to implement.

For example, all `expr` operators should be inlined, but only the unary operators are inlined in the current implementation. Command invocations could be accelerated by saving argument arrays on the stack. Invocations that set a variable to the result of an inlined command could be optimized by emitting logic to skip setting the interp result. Many scripts would run more quickly if the `regexp` and `regsub` commands were inlined, since regular expression patterns could be compiled ahead of time.

It may also be possible to significantly reduce the size of emitted Java bytecode by using code templates that cover the most common Tcl command invocations. Additional research is needed to determine if these and other optimizations would make TJC compiled code execute more efficiently.

9 Conclusions

The TJC compiler project has been an unqualified success. The compiler is functionally complete, it supports both batch and runtime compilation, and emits Java source code that is easily understood. Comparing execution times for interpreted Jacl to TJC shows that compiling results in an order of magnitude improvement. TJC outperforms native Tcl in many cases, sometimes by a wide margin. Timing results show that TJC tied both Jython and Pnuts as the fastest scripting languages implemented in Java.

The TJC project is an effective example of how corporations can benefit from open source software. In this case, porting the infrastructure from a legacy system to the Java environment was significantly less expensive than reimplementing thousands of scripts. TJC has been thoroughly tested and contains no known defects. The compiler is currently being evaluated for deployment in productions systems.

10 Availability

Jacl 1.4.0 includes the TJC compiler, runtime modules, and complete documentation. Jacl can be downloaded from:

http://tcljava.sourceforge.net

Both Jacl and TJC are licensed under BSD like licenses. For details, see the `license.terms` and `license.amd` files included in Jacl.

11 Thanks

Many thanks go to the Austin based Advanced Process Control Framework team at Advanced Micro Devices, Inc. The TJC project would not have been possible without their vision, funding, and patience. Thanks also go to New Iron Systems for making it possible for this project to move forward.

Many thanks go to One Moon Scientific for funding TJC's runtime compiler implementation. The runtime compiler work was made possible by NIH grant R01GM073053-02. Thanks also go to the original Tcl compiler team at Sun Labs and later Scriptics. Many good ideas that first appeared in the Tcl compiler were subsequently implemented in TJC.

12 References

[1] An On-the-fly Bytecode Compiler for Tcl : Lewis : http://www.usenix.org/publications/library/proceedings/tcl96/full_papers/lewis/index.html

[2] Tcl bytecode optimization : some experiences : Kenny, Sofer, Hobbs : http://aspn.activestate.com/ASPN/Tcl/TclConferencePapers2002/Tcl2002papers/kenny-bytecode/paperKBK.html

[3] Catenation and specialization for Tcl virtual machine performance : Vitale, Abdelrahman : http://portal.acm.org/citation.cfm?id=1059591&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618

[4] Alternative dispatch techniques for the Tcl VM interpreter : Vitale, Zaleski : http://www.cs.toronto.edu/~bv/tcl2005/tcl2005-vitale-zaleski.pdf

[5] The Behavior of Efficient Virtual Machine Interpreters on Modern Architectures : Ertl, Gregg : http://citeseer.ist.psu.edu/ertl01behavior.html

[6] The Structure and Performance of Efficient Interpreters : Ertl, Gregg : http://www.jilp.org/vol5/v5paper12.pdf

[7] Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences : Gagnon, Hendren : http://www.sable.mcgill.ca/publications/papers/2003-2/sable-paper-2003-2.pdf

[8] BCEL: The Byte Code Engineering Library : http://jakarta.apache.org/bcel/

[9] Serp : An open source framework for manipulating Java bytecode : http://serp.sourceforge.net/

[10] Janino : An embedded Java compiler : http://www.janino.net/

[11] TJC generated code examples : http://tcljava.cvs.sourceforge.net/*checkout*/tcljava/tcljava/docs/TJC/example2.html

[12] Comparing Tcl to TJC/Jacl : http://www.modejong.com/tcljava/tjc_results_6_20_2006/index.html

[13] Scripting Examples : http://www.modejong.com/tcljava/tjc_scripting_examples.zip