TurboJ, a Java Bytecode-to-Native Compiler

Michael Weiss, François de Ferrière, Bertrand Delsart, Christian Fabre, Frederick Hirsch,
E. Andrew Johnson, Vania Joloboff, Fred Roy, Fridtjof Siebert, and Xavier Spengler

Open Group Research Institute

Abstract. TurboJ is an off-line Java compiler, translating Java bytecodes to native code. TurboJ operates in conjunction with a Java Virtual Machine (JVM); among the supported JVMs are those on HPUX, Linux, and Wind River's Tornado for Java (running under VxWorks). Interfacing with a standard JVM entails benefits not enjoyed by the alternate "standalone" approach; particularly important for embedded systems is the opportunity to reduce the memory footprint via mixed-mode execution.

In this paper, we discuss the characteristics of TurboJ as currently implemented, focusing on two aspects of TurboJ: its interactions with the JVM, and the optimizations it makes. We then briefly summarize TurboJ's current status, compare it with some other Java off-line compilers, and outline future plans.

1 Introduction

TurboJ is an off-line Java-to-native compiler, translating Java bytecodes to native code. At run-time, the TurboJ-generated code interfaces with a standard Java Virtual Machine (JVM), which provides memory and thread management, and class and library loading services. This combination supports seamless mixed mode execution, where some classes are compiled in advance, and others are interpreted or dynamically compiled by a Just-In-Time compiler (JIT). TurboJ currently works with JVMs for HPUX, Linux, Solaris, ScoUNIX, and Wind River's Tornado for Java under VxWorks.

We say a Java application is *closed* if all its class files are available at compile-time, so it does not dynamically load classes over the net. This is an important target area for TurboJ, and obviously for embedded systems. However, TurboJ does support open applications as well.

In this paper, we focus on two aspects of TurboJ: its support for mixed-mode execution of bytecode plus native binary, and the optimizations it makes. We will see that mixed-mode execution helps deal with the space constraints of embedded systems, and the optimizations of course help with the performance needs. However, we will not address hard real-time requirements.

2 TurboJ and the JVM

A basic choice confronts the designer of an off-line Java-to-native compiler: either generate a standalone executable that "does it all", providing all its own run-time services, or else generate native code that will interface with a JVM for certain services. Toba[9] and IBM's HPC-Java[6] are examples of the former approach; TurboJ takes the latter approach.

[Figure is shown at end of article]

Figure 1: TurboJ inputs and outputs

In principle, the standalone approach offers the maximal opportunity for optimization: every aspect of execution is under the control of the generated native code. But the JVM-interface approach has compensating advantages. First, the entire Java application need not be available at compile-time: dynamic class-loading is permitted. Next, bytecode is a fairly compact format compared to native code. Most JVM instructions use only 1 or 2 bytes. Moreover, they are sophisticated instructions that cannot be translated into a single native processor instruction as a rule. For example, if every JVM instruction expands on average to two 32-bits instruction, the code will blow up by a factor of 4. In fact, most Java compilers that we know of expand code size by a factor of 5 to 10. For embedded systems where memory is scarce, space-time trade-offs are inevitable, and the ability to support mixed-mode execution (compiled with interpreted) is a definite plus. (TowerJ[12] supports dynamic loading by translating bytecodes into native code on-the-fly. Note that this does not reduce the run-time memory footprint.) Finally, this approach leads to increased robustness: the JVM furnishes a "fallback" mode of execution. (We discuss this further in Section 5. TurboJ enjoys these advantages while still achieving substantial speed-ups (typically 4 or more times JIT performance).

TurboJ relies on the JVM for object management, thread management, class and library loading, and the execution of any classes left in bytecode format. Everything else (including class initialization and exception handling) is handled by TurboJ.

The input to TurboJ consists of one or more class files; the output consists of new class files (called *interludes*), and C files. (See figure 1.) The C files are compiled with cc and linked into one or more shared libraries. Using C as an intermediate format eases the task of porting TurboJ to different platforms, and leverages the optimization technology of the native C compiler. We refer to the final result as "Turbo-ized code" here, since "compiled code" would be ambiguous. Bytecodes are, after all, the output of the javac compiler.

The interface between TurboJ and the JVM may be divided into two parts. First, the JVM invokes Turbo-ized methods via the interludes. Second, Turbo-ized code requests services from the JVM via special entry points in the JVM.

2.1 Interludes and Mixed Mode Execution

The interlude interface is relatively simple. If the original class file `Foo.class` declares a method `bar()`, then the corresponding interlude file is also named `Foo.class` and contains a method `bar()` with the `native` attribute. Although TurboJ compiled methods are declared native, the interface is not quite identical to that of user-written native methods. Fortunately, the JVM internal data structures include an *invoker attribute* for each method of a class. If the class is an interlude class, class initialization causes a TurboJ initialization routine to be called, and this routine sets the invoker attribute to point at a special TurboJ invoker. Figure 2 illustrates the generated files for a simple case.

There is significant overhead in going from Java code to native or compiled code and vice versa, especially when arguments have to be reformatted (or marshalled, in ORB parlance). For example, the JVM ABI requires that 64 bit quantities be passed in two 32 bit slots. TurboJ attempts to avoid this reformatting overhead via mechanisms discussed in the next paragraph. When it cannot be avoided entirely, it may be ameliorated by other optimizations. (See Section 3.3.)

Interludes allow interpreted code to call Turbo-ized code. Turbo-ized code can also call interpreted code; the JVM provides an entry point to support this. Of course, if a Turbo-ized caller needs to invoke a (known) Turbo-ized callee, it uses the standard C calling convention, i.e., it makes a direct call. A Turbo-ized caller may not know whether the callee has been Turbo-ized. In this case it calls

Java Source

```
public class foo {
  public void bar(double f, int i) {
    phoo(f + i);
  }
  .....
}
```

Generated Interlude

Note: the interlude is generated as classfile (i.e., bytecodes); this is the equivalent as Java source.

```
public class foo {
  public void bar(double f, int i) native;
}
.....
```

Generated C file (simplified outline)

```
// declarations of static constants, e.g., offsets

void TurboJ_foo_bar (
      JHandle*  h_var_0, /* handle to "this" */
      double    d_var_1,
      int       var_2) {

  /* direct call */
  TurboJ_foo_phoo(h_var_0, d_var_1 + (double) var_2);

  /* but if a direct call was not possible, then something roughly
     like this: */
  double d_var_3 = d_var_1 + (double) var_2;
  /* reformat the argument */
  tmp.v = d_var_3;
  _args[1].i = tmp.a[0];
  _args[2].i = tmp.a[1];
  /* obtain info about method phoo from method table */
  method_info = method_table(...);
  callJavaMethod(_args[0].h, method_info, _args);
 }
```

Figure 2: Example: Generated Interlude and C File

a special C routine we call a *trampoline*. The trampoline queries the JVM to find out the status of the callee; the JVM has this information by virtue of loading the interlude. The trampoline then dispatches the call appropriately, marshalling arguments as needed, and incidentally caching the result of the query for future use.

The Turbo-ized version of Sun's HotJava browser showcases the mixed-mode capabilities of TurboJ. The source for HotJava is a collection of 539 class files, and is one of the applications we use to test TurboJ. At run-time, dynamically loaded applets run in interpreted (or JIT) mode under the Turbo-ized HotJava, while the HotJava classes run as native code. The two sorts of classes communicate without difficulty.

Finally, we note that both the HPUX and Solaris JVMs contain JIT compilers. TurboJ works with both of these, dynamically loaded classes being compiled by the JIT.

2.2 Object Management

TurboJ currently relies on the JVM for all object allocation and garbage collection. The need to maintain consistency during garbage collection chiefly motivated this design choice. By entrusting the JVM with all aspects of heap management, we insure a robust implementation with minimal demands on the JVM. Indeed, one of TurboJ's primary design goals was that it work with off-the-shelf JVMs.

The attendant inefficiencies are alleviated in several ways, discussed in Sections 3.2 and 3.3.

3 TurboJ Optimizations

We divide these into three categories: whole-program optimizations, per-class optimizations, and per-method optimizations.

3.1 Whole-Program Optimizations

TurboJ attempts to compute the *closure* of the set of input class files: i.e., it follows references to external classes, recursively adding them to the input set. In the ideal scenario of a closed application, TurboJ will have available the bytecodes for all referenced classes when it generates the interludes and native code. TurboJ analyses the structure of the application as a whole, computing the class hierarchy tree and the class reference graph. However, TurboJ is not dependent on having total information: if only some classes are available, it will compute what it can and make conservative assumptions about the rest.

TurboJ uses this global information to make a number of optimizations:

Virtual-to-nonvirtual conversion: Virtual dispatches are converted to non-virtual dispatches if the apparent callee is "effectively final", i.e., not overridden. Since this optimization makes assumptions about what classes may appear at run-time, it is under the control of a switch, on a per-class basis.

Direct calls: Often TurboJ can determine the target of a call at compile-time. If the callee is slated for Turbo-ization, then the C calling convention is used, instead of going through the JVM.

Direct offsets: TurboJ can determine many field offsets at compile-time, and uses them in lieu of run-time resolution.

Inlining: TurboJ will inline calls when appropriate.

TurboJ also supports a distinction between "inside" and "outside" classes. An outside class is one whose class file is available for inspection, but which is not considered part of the application; i.e., TurboJ should not generate any code for it. System classes are typically treated as outside classes.

The "inside/outside" distinction can be employed to achieve separate compilation. The core system classes themselves have been Turbo-ized, resulting in a shared library and core interludes. If an application is Turbo-ized using these system interludes as outside classes (instead of the original system classes), then TurboJ will be able to make direct calls from the application into the system classes. This technique works in general for any pre-Turbo-ized collection of utility classes.

3.2 Per-Class Optimizations

In Java bytecodes, field references are kept in symbolic form; these are converted into offsets as part of *constant pool resolution*. For example, the JVM spec for the `getfield` instruction says that the operands of the instruction are used to construct an index into the constant pool of the class file; the indicated constant pool item is then "resolved, determining both the field width and the field offset"[7].

The JVM spec also describes the `getfield_quick` instruction, a optimized version of `getfield` which contains the field offset instead the index into the constant pool. The JVM can replace `getfield`s with `getfield_quick`s when the class is loaded, insuring that field offsets are resolved only once. So the most obvious optimization is already performed by the interpreter.

TurboJ improves on this in two ways. The first was noted above: field offsets can often be computed when the code is Turbo-ized. For the rest, TurboJ generates initialization routines, both for the entire class and for each method in the class. (A method initialization routine are called the first time a method is invoked.) These routines do as much constant pool resolution as is possible "up-front"; although care must be taken to preserve the late-binding semantics of the JVM spec, in practice almost all resolution can be done when the class is initialized.

Performing up-front resolution reaps the same "resolve once" benefit as the `getfield_quick` optimization. But in addition, by collecting the initialization code in one spot, it becomes subject to common subexpression analysis by TurboJ, as well as the optimizations of the native C compiler.

Similar remarks apply to constant pool resolution for method names.

3.3 Per-Method Optimizations

To avoid the inefficient simulation of the Java-stack in the compiled C code, the bytecode instructions are first transformed into expression trees. The trees are then translated into nested C-expressions. TurboJ endeavors to create fewer larger trees instead of many small trees. A C compiler will generally do a good job of register allocation on code of this sort. Explicit assignments to local variables that simulate the Java-stack can be avoided in most cases.

The data and control flow optimizations should not simply be left to the C compiler. TurboJ possesses additional information which enables optimizations the C compiler cannot perform. In other words, we can take advantage of Java's tighter semantics vis-a-vis pointers. It is often possible to prove that an object's field has not changed between two accesses; the C compiler has to assume the worst. We give an example in the next section.

To exploit this additional potential, TurboJ runs an extra pass for common subexpression detection and removal. As usual in compilation, exposing more semantics enables better optimization. For example, a field access is split into three independent expressions: check for a null object pointer; dereference the pointer; access the field. In several consecutive accesses to different fields of the same object, the common null check and dereferencing need be done only once.

With surprisingly little effort, this scheme for common subexpression removal can be extended to hoist loop invariant expressions out of loops. Care has to be taken for exceptions, which still have to be checked during the execution of the loop. The generated code for the hoisted code is done conditionally only when no exception is thrown.

We discovered early on that Java bytecodes make heavy use of StringBuffer class methods, even when this system class is not explicitly mentioned in the Java source code. For example, a string concatentation expression such as `lastName + "," + firstName` results in two StringBuffer appends. TurboJ recognizes such sequences and translates them into more efficient C code.

Finally, TurboJ performs certain optimizations with regard to exception handling, and miscellaneous peephole optimizations as it emits the actual code.

3.4 Example: Optimization of Field Accesses

We said above that TurboJ benefits from the fact that Java has tighter semantics than C, and as a consequence can often perform optimizations that the C compiler cannot. We give an example in this section.

Field accesses are implemented using pointers to arrays; these are indexed by a value that is usually constant. The C compiler cannot generally prove that these accesses are not aliased. Different fields in different objects cannot reference the same memory location, and this is obvious to TurboJ, but not to the C compiler in the generated code.

We illustrate this with the kernel of a bubblesort algorithm (taken from the CaffeineMark loop benchmark). Figure 3 gives the Java source for the main nested loops.

```
for(i=0; i<this.count; i++) {
  for(j=1; j<this.count; j++) {
    if (this.array[j-1] >= this.array[j]) {
      swap = this.array[j-1];
      this.array[j-1] = this.array[j];
      this.array[j] = swap;
    }
  }
}
```

Figure 3: Nested Loop Example, Java Source

We have loop-invariant reads to the fields `this.count` and `this.array` and read and write accesses to the elements of an array. Figure 4 gives a straightforward translation into C, ignoring runtime checks and other details.

```
for(i=0; i<this[count_offset]; i++) {
  for(j=1; i<this[count_offset]; j++) {
    if (this[array_offset][j-1] >= this[array_offset][j]) {
      swap = this[array_offset][j-1];
      this[array_offset][j-1] = this[array_offset][j];
      this[array_offset][j] = swap;
    }
  }
}
```

Figure 4: Naive translation into C

The C compiler would have severe difficulties in proving that the pointers `this` and `this[array_offset]` are not identical. Thus it would not be able to move the field accesses out of the loop body. But Java semantics states that an object and an array can never occupy the same memory location. TurboJ can thus safely perform two sorts of optimizations: the loop invariants `this[count_offset]` and `this[array_offset]` are hoisted out of the loop, and the common subexpressions `this[array_offset][j-1]` and `this[array_offset][j]` are each computed only once. The resulting code looks like figure 5, again ignoring runtime checks and other details.

The results on this benchmark are 50% better with these optimizations than without. Specifically, on an HP-UX 11 using HP's C compiler, the Caffeine loop score was 15122 with the optimizations and 10315 without.

```
loopinv_1 = this[count_offset];
loopinv_2 = this[array_offset];
for(i=0; i<loopinv_1; i++) {
  for(j=1; i<loopinv_1; j++) {
    cse_1 = loopinv_2[j-1];
    cse_2 = loopinv_2[j];
    if (cse_1 >= cse_2) {
      swap = cse_1;
      loopinv_2[j-1] = cse_2;
      loopinv_2[j] = swap;
    }
  }
}
```

Figure 5: Translation into C with Optimizations

4 Current Status

TurboJ supports the full JVM 1.1 instruction set. All required checks (null checks, bound checks, etc.) are enforced. It passes the JCK conformance suite, and has been used to process a number of substantial applications, e.g., HotJava, and the Java WorkShop.

The most up-to-date benchmark results may be found at our Website[13]. We mention here just a couple of results.

On an HP-UX B.10.10 A 9000/829, using the vendor's compiler with optimization level +O2 and the JVM HP-UX Java B.01.12.01, the overall embedded CaffeineMark 3.0 score is as follows (note that higher is better for CaffeineMark):

Caffeine	interpreted	JIT	TurboJ	TurboJ + JIT
	64	381	2084	2242

On an HP K6, 200 Mhz, 32 MB RAM, on VxWorks 5.3.1 with Tornado for Java 1.1 Beta, the results are:

Caffeine	interpreted	TurboJ
	149	3966

5 Related Work

We consider here the following off-line Java-to-native compilers: Harissa[8], jc1[1], Toba[9], JCC[10], jc1[1], TowerJ[12], and IBM's High Performance Compiler for Java[6]. Our information in all cases is taken from the cited references, the most recent we are aware of.

Harissa is a fairly mature research prototype from IRISA/INRIA. Like TurboJ, Harissa allows the mixing of compiled code and interpreted bytecode. Instead of interfacing to a JVM, however, Harissa achieves this by linking in its own version of the JVM. As a consequence, some functions of the Java virtual machine were missing as of December 1997.

Jc1 comes from Cygnus, and is intended for embedded applications. Like TurboJ, jc1 interfaces to a JVM, but unlike TurboJ, it requires a specific JVM, namely Kaffee[15]. As of July 1997, many bytecodes were not supported, nor was exception handling.

Harissa, jc1, and TurboJ are, to our knowledge, the only bytecode-to-native compilers that support mixed-mode execution. The remaining compilers all use some form of the standalone executable approach. Several of them (according to the cited references) do not support applets or graphics, or the JVM 1.1 spec, or differ from the standard in minor ways. While we expect that these defects will be remedied in time, they do illustrate a point made in the introduction: having a standard JVM around furnishes a graceful fallback mode of execution.

TowerJ is the only compiler (among the “standalone” group) that supports dynamic loading of bytecode; it does this by translating the bytecode into native code on-the-fly. We have noted the memory drawbacks of this approach in Section 2.

We are not sure to what extent these compilers can deal with incomplete information at compile-time. Harissa does support separate compilation.

6 Conclusions and Future Work

One of the pleasant surprises of our TurboJ work has been the degree to which good results can be obtained relatively cheaply. We believe this is largely due to the way the design of TurboJ piggybacks on pre-existing technologies. Our use of the JVM has been discussed sufficiently above. The other instance of this phenomenon is our use of the native C compiler, with its well-developed register allocation and code selection algorithms. This can be exploited however only if the generated code has a suitable shape. Hummel et al.[5] have noted that the single biggest penalty of a naive translation of bytecode to C comes from the inefficient use of registers. They remedy this by doing their own register allocation on the bytecodes; we solve it by converting the bytecodes into C expressions with significant nesting depth.

As another example, both our class hierarchy analysis and local optimization algorithms are not yet that sophisticated, but they harvest (we believe) a major portion of the benefit from more elaborate approaches.

Our future plans include several fairly standard optimizations. We will add local optimizations based on SSA-form[4, 11, 14]. We will also upgrade our class hierarchy analysis[3]. We believe there are synergistic possibilities in the combination of these technologies; for example, the inheritance tree can serve (with slight modification) as the lattice in the Wegman-Zadeck constant propagation algorithm[14]. We also intend to make further optimizations with regard to exception handling and synchronization.

As present, TurboJ makes use of no special features of the JVM. There are clear opportunities to be had in making this connection tighter: compiler-assisted garbage collection is an obvious example. However, there are also drawbacks. Adapting TurboJ to work with Tornado for Java (under VxWorks) involved relatively few changes, since the key issues of garbage collection and thread management are delegated to the JVM.

Object inlining[2] is one JVM-neutral optimization that can reduce the overhead of object management. Budimlic and Kennedy implemented this as a bytecode-to-bytecode optimization; this lead to certain protection difficulties that they had to work around. We note that these problems simply don't arise for a bytecode-to-native compiler.

References

1. Bothner, P. Compiling Java for embedded systems. http://www.cygnus.com/news/whitepapers/compiling.html.
2. Budimlic, Z., and Kennedy, K. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience* (June 1997). http://www.npac.syr.edu/projects/javaforcse/cpande/rice/JavaPaper.ps.
3. Chambers, C., Dean, J., and Grove, D. Whole-program optimization of object-oriented languages. Tech. Rep. 96-06-02, University of Washington, June 1996. http://www.cs.washington.edu/research/projects/cecil/cecil/www/Papers/whole-program.html.
4. Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct. 1991), 451–490. http://www.acm.org/pubs/articles/journals/toplas/1991-13-4/p451-cytron/p451-cytron.pdf.
5. Hummel, J., Azevedo, A., Kolson, D., and Nicolau, A. Annotating the Java bytecodes in support of optimization. In *Workshop on Java for Science and Engineering Computation, PPoPP97* (June 1997). http://www.npac.syr.edu/users/gcf/03/javaforcse/acmspecissue/finalps/1_hummel.ps.
6. IBM high performance compiler for Java: An optimizing native code compiler for Java applications. http://www.alphaworks.ibm.com/graphics.nsf/system/graphics/HPCJ/$file/highpcj.html.
7. Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* Addison-Wesley, 1996.
8. Muller, G., Moura, B., Bellard, F., and Consel, C. Harissa: a flexible and efficient Java environment mixing bytecode and compiled code. In *COOTS97* (1997). available from http://www.irisa.fr/compose/ harissa/harissa.html.
9. Proebsting, T. A., Townsend, G., Bridges, P., Hartman, J. H., Newsham, T., and Watterston, S. A. Toba: Java for applications. a *way ahead of time* (wat) compiler. available from http://www.cs.arizona.edu/sumatra/toba/.

10. Shaylor, N. JCC – a Java to C converter. http://www.geocities.com/CapeCanaveral/Hangar/4040/jcc.html.
11. Simpson, L. T. *Value-Driven Redundancy Elimination*. PhD thesis, Rice University, Apr. 1996. http://www.cs.rice.edu/ lts/thesis.ps.gz.
12. TowerJ release 2.0: A high performance compiler for server-side Java. http://www.twr.com/java/towerj2.html.
13. Turboj home page. http://www.opengroup.org/openitsol/turboj.
14. Wegman, M. N., and Zadeck, F. K. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13*, 2 (Apr. 1991), 181–210.
15. Wilkinson, T. http://www.kaffe.org/.