# Toba: Java For Applications
# A *Way Ahead of Time* (WAT) Compiler

Todd A. Proebsting   Gregg Townsend   Patrick Bridges
John H. Hartman   Tim Newsham   Scott A. Watterson

The University of Arizona [1]

**Abstract**

Toba is a system for generating efficient standalone Java applications. Toba includes a Java-bytecode-to-C compiler, a garbage collector, a threads package, and Java API support. Toba-compiled Java applications execute 1.5–10 times faster than interpreted and Just-In-Time compiled applications.

## 1 Introduction

Java [GYT96] is an object-oriented language designed by Sun Microsystems that supports mobile code, i.e., executable code that runs on a variety of platforms. Although the language is interesting in its own right, Java's popularity stems from its promise of "write once, run anywhere." Mobile code proponents envision a future of location-independent code moving about the Internet and running on any platform.

Java's mobility is achieved by compiling its object classes into a distribution format called a *class file*. A class file contains information about the Java class, including *bytecodes*, an architecturally-neutral representation of the instructions associated with the class's methods. A class file can execute on any computer supporting the Java Virtual Machine (JVM). Java's code mobility, therefore, depends on both architecture-neutral class files and the implicit assumption that the JVM is supported on every client machine.

Most JVM implementations execute bytecodes through interpretation or *Just-In-Time* (JIT) compilation, which compiles the bytecodes into machine code at run time. Thus, Java's mobility comes at a price, exacted by the cost of interpreting or JIT-compiling the bytecodes *every time the program is executed*. These systems incur modest to severe performance penalties compared to more traditional systems that compile source code directly to machine code once. For example, a compiled C program runs 1.5-2 times faster than the equivalent JIT-compiled Java program, and 2-10 times faster than an interpreted Java program.

These performance penalties are especially bothersome in non-mobile applications that are run many times without change. To combat these inherent performance penalties we have developed a Java system that pre-compiles Java class files into machine code. Our system, Toba,[2] first translates Java class files into C code, then compiles the C into machine code. The resulting object files are linked with the Toba run-time system to create traditional executable files. To distinguish our technique from JIT compilation, we have (somewhat facetiously) coined the phrase *Way-Ahead-of-Time* (WAT) compiler to describe Toba. Toba compiles Java programs into machine code during program development, eliminating the need for interpretation or JIT compilation of bytecodes. Although we forfeit Java's architecture-neutral distribution, Toba-generated executables are 1.5-10 times faster than alternative JVM implementations.

Toba has several advantages over interpretation or JIT-compilation. First, because Toba runs way-ahead-of-time, rather than just-in-time, the resulting machine code can be more heavily optimized to yield more efficient executables. Second, because Toba creates a C-equivalent to the Java program, the standard C debugging and profiling tools can operate on Toba-generated executables. Third, because Toba executables include all class files used by the application, there is no possibility of an application suddenly ceasing to execute because of a change in available class files. For these reasons we believe that WAT-compilation is valuable for the development and distribution of efficient Java programs.

Toba consists of several components: a bytecode-to-C translator, a garbage collector, a threads package, a run-time library, and native routines implementing the Java API. Toba is a surprisingly small system: the translator is only 4000 lines of Java; the garbage collector is a modestly-altered version of the freely available Boehm-Demers-Weiser conservative collector [BW88]; the threads package is built on top of Solaris threads; the run-time library is only 5000 lines of C; and the API routines are simply translations of Sun's API class files. Except for dynamic linking and a windowing system, Toba provides a complete Java execution environment.

---

[1] Address: Todd A. Proebsting, Department of Computer Science, University of Arizona, Tucson, AZ 85721; Telephone: 520/621-4326; Email: todd@cs.arizona.edu.

[2] Lake Toba is a prominent feature on Sumatra, the island just west of Java.

## 2 The Java Virtual Machine

The Java Virtual Machine (JVM) defines a stack-based virtual machine that executes Java class files [LY97]. Each Java class compiles into a separate class file containing information describing the class's inheritance, fields, methods, etc., as well as nearly all of the compile-time type information. The Java bytecodes form the machine's instruction set, and combine simple arithmetic and control-flow operators with operators specific to the Java language's object model. Powerful object-level instructions include those to access static and instance variables, and those to invoke static, virtual, nonvirtual and interface functions. The JVM also includes an exception mechanism for handling abnormal conditions that arise during execution.

The JVM also provides facilities for managing objects and concurrency. The JVM implements a garbage-collected object allocation model, with facilities for initializing and finalizing objects. Concurrency is provided through a thread abstraction. Threads are pre-emptive and scheduled according to priority. A monitor facility provides mutual exclusion on critical sections as well as thread scheduling through wait/notify primitives. Monitors are recursive, allowing a single thread to acquire the same monitor lock multiple times without deadlocking.

## 3 Toba's Run-Time Data Structures

Java's rich object model requires run-time data structures to describe each object's type and methods. We developed our data structures with both performance and simplicity in mind. They differ in many respects from those of Sun's implementation of Java. For instance, Sun's implementation requires that all object references go through a *handle*, which represents an extra level of indirection, an added inefficiency, and an extra complication. Toba accesses objects directly. The differences are invisible to Java programmers but important to authors of native methods.

### 3.1 Naming

Toba attempts to preserve Java names in the C it produces, although this isn't always possible. Java names may draw from thousands of different Unicode characters whereas C names are limited to just 63 ASCII characters. Furthermore, some legal Java names such as `enum` and `setjmp` have special meaning in C. When a Java name cannot be used directly as a C name, Toba discards non-C characters, adds a hash-code suffix, and additionally adds a prefix character if the resulting name begins with a digit or other illegal character.

Java method names always require hash-code suffixes. Toba translates each Java method into a C function, and these functions share a global namespace. Because Java methods may be overloaded among and within classes, a hash-code suffix is added to distinguish the methods. The suffix encodes the class name, the method name, and the method signature.

### 3.2 Data Layout

Java includes eight primitive types: byte, short, int, long, boolean, char, float, and double. Each translates into a primitive C type. (Note that Java's "char" type represents a 16-bit Unicode value.)

All other Java types are *reference* types that subclass the root class, `java.lang.Object`. All reference types are translated into a C pointer type. Each reference points to an object instance, and all instances of a particular class contain a class-pointer to a common class structure. Java has two different kinds of objects: array objects and ordinary objects. The Toba structure for ordinary objects appears in Figure 1. An ordinary object's class descriptor includes the instance size and a flag that indicates it is not an array. The Toba structure for array objects appears in Figure 2. An array's class descriptor includes the element size and its flag indicates that it represents an array. Array instances contain both a length field and a vector of elements.

Each per-class run-time structure has three parts: general information that is needed for all classes (e.g., superclass information), a method table that contains pointers to virtual functions, and a table of class variables. Figure 3 summarizes run-time class-level information common to all classes.

The method table is simply a vector of function pointers and unique method identifiers. The method identifiers are used when invoking interface functions, which must be found at run-time. The structure of the method table is typical of statically-bound object-oriented languages like Oberon-2 [MW91] and C++ [Str86]. Method tables include inherited methods as well as functions defined by the class itself.

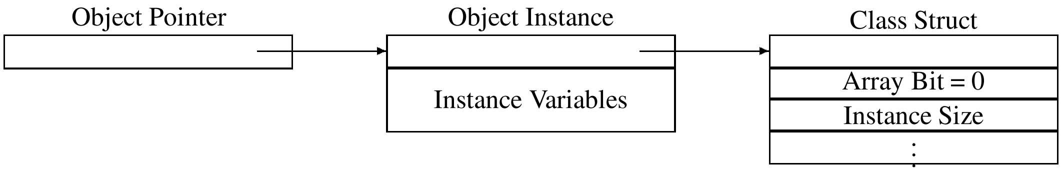


Figure 1: Ordinary Object Structure

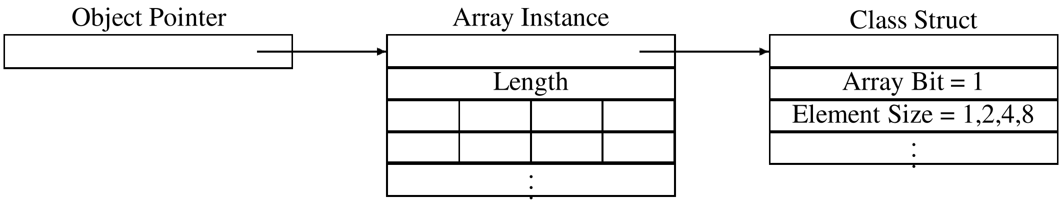


Figure 2: Array Object Structure

| | |
|---|---|
| initialization flag | Determines if the class has been initialized |
| other flags | Miscellaneous flags including the Array Bit |
| class name | Pointer to instance of class `java.lang.String` |
| class instance | Instance of class `java.lang.Class` |
| superclasses | Pointer to vector of superclasses for checking subclass relationship |
| interfaces | Pointer to vector of interfaces |
| referenced classes | Pointer to vector of referenced classes |
| array class | Pointer to array class of current class |
| element class | Pointer to element class, if array class |
| initializer | Pointer to class initializer function |
| constructor | Pointer to default instance initializer function |
| finalizer | Pointer to instance finalizer function |

Figure 3: Fields of Class Descriptors

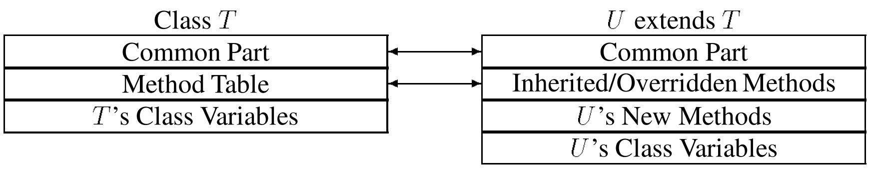


Figure 4: Class/Subclass Structures

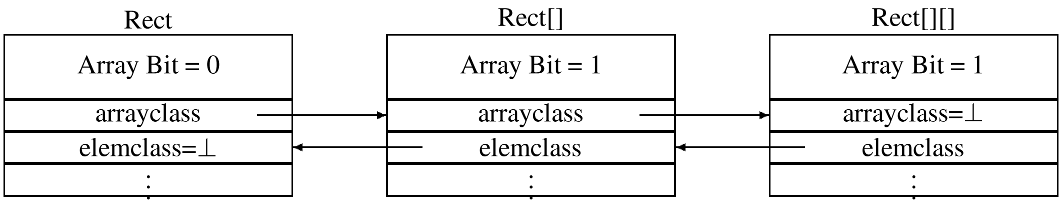


Figure 5: Array Class Descriptors

| Java | Reference Type | Toba-generated Reference |
|---|---|---|
| `r.width` | instance variable | `r->width` |
| `r.flip()` | virtual method | `r->class->M.flip_r_b79qV(r)` |
| `r.clear()` | interface method | `findinterface(r,298564082)(r)` |
| `rect.clearAll()` | static method | `clearAll_b7zk4()` |
| `rect.nrects` | class variable | `cl_rect.V.nrects` |

Table 1: Toba-generated References (Omitting C Casts)

Class variables exist on a per-class basis, not a per instance basis. Toba-generated programs reference class variables as externals stored in the class structure. Figure 4 shows the class/subclass relationship of class descriptors.

Class descriptors for arrays require special handling. An array of class $X$ ("$X$[ ]") may be declared by any arbitrary class that imports $X$. Similarly, an array of arrays of arrays of $X$, $X$[ ][ ][ ], may be declared by any class that imports $X$. Descriptors for these array classes must be unique—all instances of $X$[ ][ ][ ] must share the same class descriptor. Therefore, these array class descriptors must be able to be built at run-time. (It is possible to build them at link-time, but we chose to avoid this complication.) Figure 5 illustrates the simple relationship between the descriptor of a class and the descriptor of an array of that class.

### 3.3 Referencing Values and Methods

Toba constructs efficient value and method references in C. Assume, for instance, that `r` is an instance of class `rect`. Table 1 summarizes the way Toba references objects and methods in C. Toba-generated C accesses the instance variable `width` as `r->width`. A virtual function call requires an indirection through the method table and requires passing the instance as the first argument. Note that method names include hash suffixes. An interface call utilizes a table-lookup of the appropriate method based on its unique identifier (e.g., 298564082). Static methods and class variables do not require an instance variable. A static method invocation is a simple C function call. Class variables are accessed via the class's run-time descriptor.

## 4 Code Translation

Toba translates one class file at a time into a C file and a header file. To translate a class file, Toba requires the class files for all of the class's superclasses. To compile a class's resulting C file, header files are necessary from itself, its superclasses, and all imported classes.

### 4.1 Code Translation

Within class files, methods are encoded in the JVM's byte-coded instruction set. Toba translates each method into a C function. Toba assumes that the class file is valid and verifiable, although it does nothing to confirm this assumption.

The JVM instruction set is stack-based. During execution, (verifiable) bytecode maintains a stack invariant that is critical for translation into efficient C (or native) code: regardless of the previous execution path, at any given point in the program, the stack is always in a consistent state (i.e., the same number and types of values are on the stack). For instance, if along one path to a given program point, $P$, the stack is empty just prior to executing $P$, then along *all* paths the stack will be empty just prior to executing $P$. This invariant means that the depth of the stack and the types of its contents at any point in the program are fixed. A simple traversal of the bytecode can determine this information at compile time. Using this information, the Toba translator is able to turn all stack accesses into references to simple local variables—one per stack location. This eliminates the need for an explicit stack or stack pointer.

Most Java constructs translate simply into bytecode for this stack machine. For instance, the middle column of Figure 6 gives the bytecode for `a=b+c;` assuming that `a`, `b`, and `c` are the first, second and third local variables of the enclosing method. The `iload` and `istore` instructions refer to loads and stores of local variables. Toba creates a C local variable for each JVM local variable.

| Java | Bytecode | Generated C |
|---|---|---|
| `a = b + c;` | `iload_2` | `i1 = iv2;` |
| | `iload_3` | `i2 = iv3;` |
| | `iadd` | `i1 = i1 + i2;` |
| | `istore_1` | `iv1 = i1;` |

Figure 6: Translating `a = b + c;` into C

Figure 6 gives a simple translation of the previous Java statement into C. In the example, `i1` and `i2` refer to the first and second elements of the stack, and `iv1`, `iv2` and `iv3` refer to the first three JVM local variables. Once the stack depths are known, Toba generates naive code. Toba relies on an optimizing C compiler to do copy propagation and register allocation to eliminate useless copies and local variables.

Generating code for each method follows the following outline:

1. Read the bytecode instructions from the class file
2. Compute the stack state at every instruction
3. Note instructions that are exception range entry points and assign labels to them
4. Note jump target instructions and assign labels to them
5. Generate C function header
6. Generate C code for each instruction

Computing stack states requires visiting all instructions. After computing stack state, Toba translates bytecode instructions one at a time.

The Java bytecode supports both direct (conditional and unconditional) branches, as well as indirect jumps. Toba computes all potential targets of direct and indirect jumps, as well as exception handling blocks, in a control-flow analysis. (Verifiable bytecodes are guaranteed to be easy to analyze accurately.) Toba emits a C label before the executable code for each target instruction. To handle indirect jumps and exception handling, a giant `switch` statement wraps each method's generated C code, with each indirect target having its own `case` arm. Thus, indirect jumps translate into C code that sets a program counter variable, jumps to the top of the `switch`, and then dispatches on that variable to the appropriate chunk of code. Unconditional direct jumps become `goto`'s; conditional direct jumps become `if (...) goto L`*n* statements. As an optimization, Toba omits the `switch` wrapper in the absence of exception handling blocks and indirect jumps.

Figure 7 shows a simple Java method along with its translation into bytecode and then into C. The naive code generation algorithm has produced several more assignments than would a human coder, but modern C compilers are good at removing these.

## 4.2 Exception Handling

The Java Virtual Machine supports exception handling in a manner similar to Ada [Bar84] or C++ [Str86]. Exceptions are *thrown*, either implicitly or explicitly, and are *caught* by the closest matching exception handler. Exceptions that cannot be caught in a procedure require the JVM to unwind the call stack and re-throw the exception in the caller's environment. Re-throwing continues until the exception is caught.

Exception dispatching is based on the execution-time program counter of the JVM. Toba simulates the program counter by assigning values to a local `pc` variable. It is not necessary to set `pc` for every JVM instruction, but only when entering or leaving an exception range (taking into account that jumps can enter the middle of a range).

Toba uses C's `setjmp` and `longjmp` routines to control the call-stack unwinding. For each C function that may catch an exception, Toba creates a small prologue that calls `setjmp` to initialize a per-thread `jmpbuf`. The prologue saves the previous `jmpbuf` value in a local structure; epilogue code restores the old value before the function returns. Toba translates exception throwing into `longjmp` calls that use the `jmpbuf`. Such calls transfer control to the prologue of the nearest function that might handle the exception. This prologue code simply checks a table to determine if, given the type of the exception and the currently active program counter, this procedure can handle the exception.

```
class d {
   static int div(int i, int j) {
      i = i / j;
      return i;
   }
}
```

```
Method int div(int,int)
   0 iload_0
   1 iload_1
   2 idiv
   3 istore_0
   4 iload_0
   5 ireturn
```

Figure 7: Simple Java Program and Bytecode

```
Int div_ii_3WIeN(Int p1, Int p2)
{                                int div(int, int)
Int i0, i1, i2;                  integer stack
Int iv0, iv1;                    integer variables

      iv0 = p1;                  init variables from params
      iv1 = p2;

L0:   i1 = iv0;                  iload_0
      i2 = iv1;                  iload_1
      if (!i2)                   idiv
         throwDivisionByZeroException();
      i1 = i1 / i2;
      iv0 = i1;                  istore_0
      i1 = iv0;                  iload_0
      return i1;                 ireturn
}
```

Figure 8: Sample Toba Output

If so, the target label is set to the appropriate handler and execution transfers to the `switch` statement that dispatches indirect jumps. Otherwise, the prologue restores the previous `jmpbuf`, and immediately executes a `longjmp` with this `jmpbuf`.

### 4.3 Class Initialization

Each Java class may define an initialization routine to be run exactly once. Any of the following events can trigger initialization:

- The first creation of an instance of a class.
- The first invocation of any of a class's static methods.
- The first read or write of any class (not instance) variable.

In the worst case, each of these operations includes checks to determine if the class initializer must be run. Calls to allocation routines check a per-class initialization flag. Static methods include checks in their prologue code—no checking is done by the caller. Static-variable accesses include checks of the initialization flag.

Often, these checks are not needed. Toba omits the checks for classes that have no initialization routine.

## 5 Garbage Collection

Toba's garbage collector is based on the freely-available Boehm-Demers-Weiser (BDW) conservative garbage collector [BW88]. A conservative collector treats every register and word of allocated memory as a potential pointer and traces all memory reached from these pointers. Therefore, the BDW collector does not need type information for the memory it manages. This frees Toba and native routine developers from concerns about memory management.

Our modifications to the BDW collector are relatively minor, affecting about 30 lines of code. First, the BDW collector is a mark-and-sweep collector that requires all threads to be stopped during collection. This proved to be expensive in Toba's thread package (Solaris threads), so we optimized the "stop the world" functionality for the single-threaded case.

Second, the behavior of finalizers and cyclic data structures in the JVM are slightly different from those supported by the BDW collector. The Java language specification (page 231-234 , [GJS96]), allows object finalizers to make previously unreachable objects reachable again, thereby "resurrecting" the objects. Although the BDW collector supported finalization and resurrection of objects, it did not collect cyclic data structures containing finalizable objects. We therefore made another minor modification to the BDW collector to add this functionality.

# 6 Threads and Synchronization

The JVM defines a priority-based, preemptive thread model that includes synchronization facilities. Toba implements Java threads using Solaris threads, and uses Solaris locks to protect internal critical sections. The biggest problem we encountered when implementing Java threads is that Java allows threads to both suspend each other and to cause other threads to receive an asynchronous exception, such as thread termination. Toba uses UNIX's signal mechanism to handle these asynchronous events, causing the receiving thread to either suspend itself or throw an exception, as appropriate. The problem is that this may cause a thread to block (or even die) in the middle of a critical section, leaving the critical section locked. To eliminate this possibility Toba uses a limited form of roll-forward [MDP96] to allow a thread interrupted by a signal to exit the critical section before handling the signal. Note that this problem also exists with critical sections in the Java code itself; the Java literature does not offer much of a solution other than recommending limited use of these asynchronous thread operations.

Java threads synchronize via monitors. Each object and class has a monitor associated with it, and only one thread at a time may hold the lock associated with a monitor. Condition variables are also provided to allow thread scheduling; the standard wait, notify, and broadcast operations are supported.

An unusual feature of Java monitors is that they are recursive, i.e. the same thread may enter a monitor recursively without deadlock. This implies that Toba cannot implement Java monitors using lock and unlock primitives directly; instead monitors are a more complex data structure containing a lock, a reference count, and the identity of the thread holding the lock. If a thread enters a monitor whose lock it already holds, the reference count is incremented. Similarly, when the monitor is exited the reference count is decremented and the lock only released when zero is reached. If a thread leaves the monitor to wait on a condition, the lock is released and the reference count cleared; when the thread subsequently re-enters the monitor the lock is re-acquired, and the reference count is restored.

To reduce synchronization overhead, Toba has an optimized monitor implementation for single-threaded applications. Entering and exiting monitors only affects their reference count; the monitor locks are not used. Should another thread be created, the original thread first locks all monitors that have a positive reference count, thus ensuring mutual exclusion now that there is more than one thread.

# 7 Performance Results

## 7.1 Methodology

We tested Toba's performance using using both *application* benchmarks and *micro*-benchmarks. The application benchmarks test the overall system performance, while the micro-benchmarks isolate the performance of individual language features (e.g., exception handling, thread switching, etc.).

We compared Toba's performance to two other systems: Sun's interpreter (JDK version 1.0.2), and the Guava JIT compiler (version 1.0 beta 1), by Softway Pty, Ltd. We compared against the Sun interpreter because it is the reference implementation of Java, and against the Guava JIT compiler because it is the only other compilation system for SPARCs of which we are aware. We ran benchmarks on a Sun SPARCStation-20 with 128 MB of memory and two Model 61 SuperSPARC processors. C code was compiled using the Sun's commercial C compiler with full optimization (-xO4 -xcg92).

The Guava JIT compiler and Sun interpreter must do more work at run time than Toba to execute benchmarks. Both systems must dynamically load each class file, and the Guava JIT compiler must compile each method before it can be run. The micro-benchmark times do not include the time to load class files, while application benchmarks do include this time.

## 7.2 Application Benchmarks

Table 2 describes the application benchmarks. Figure 9 shows the execution times of the benchmarks on the three systems, normalized to the Toba time. Each data point represents the average of ten runs of the benchmark. Guava's results include the time to JIT compile the benchmark. The Toba-generated benchmarks are 1.5–10 times faster than those same benchmarks running under other systems. Toba-generated code runs 3–10 times faster than programs running under the JDK interpreter, and 1.5–2.2 times faster than under the Guava JIT compiler. This speedup results in a tangible improvement in the time to complete the benchmark; the JavaLex benchmark, for example, improved from 208

| Application | Description | Input |
|---|---|---|
| JavaLex | Lexical analyzer generator that translates regular expressions into finite-state machines that are subsequently translated into Java | Specification that includes 77 patterns |
| JavaCUP | LALR(1) parser generator that translates context-free grammars into push-down automata that are subsequently translated into Java | Grammar that includes 24 terminals, 32 nonterminals, and 65 productions |
| javac | Sun's Java compiler that translates Java source programs into class files (bytecode) | Toba source files consisting of 3891 lines of Java |
| espresso | Translates Java source programs into class files (bytecode) | Toba source files consisting of 3891 lines of Java |
| Toba | Bytecode-to-C translator described in this paper | Toba's 18 class files (77,718 bytes) |
| JHLZip | Combines multiple files into one archive file, but does no compression | 5.9MB of English dictionary words |
| JHLUnzip | Extracts multiple files from JHLZip archives | 5.9MB of English dictionary words |

Table 2: Application Benchmarks

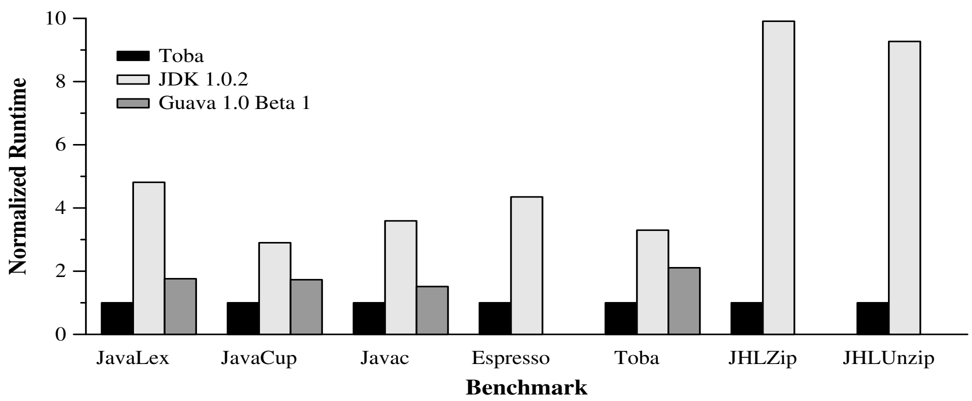


Figure 9: Normalized Application Timings

| Benchmark | Toba (sec.) | JDK (sec.) | JDK/Toba | Guava (sec.) | Guava/Toba |
|---|---|---|---|---|---|
| JavaLex | $43.2 \pm 3.2$ | $207.7 \pm 18.4$ | 4.8 | $76.0 \pm 1.0$ | 1.8 |
| JavaCup | $3.7 \pm 0.1$ | $10.6 \pm 2.1$ | 2.9 | $6.3 \pm 0.1$ | 1.7 |
| javac | $13.6 \pm 0.9$ | $48.9 \pm 2.3$ | 3.6 | $20.6 \pm 0.4$ | 1.5 |
| espresso | $6.0 \pm 0.4$ | $26.1 \pm 0.9$ | 4.4 | Abort | N/A |
| Toba | $18.8 \pm 1.1$ | $62.0 \pm 2.8$ | 3.3 | $39.6 \pm 2.0$ | 2.1 |
| JHLZip | $5.3 \pm 0.1$ | $52.7 \pm 0.3$ | 9.9 | Abort | N/A |
| JHLUnzip | $5.6 \pm 0.2$ | $51.5 \pm 0.1$ | 9.3 | Abort | N/A |

Figure 10: Application Benchmark Timings

| Arithmetic Benchmarks | |
|---|---|
| add-int | Add two integers |
| multiply-int | Multiply two integers |
| add-double | Add two double-precision floating point numbers |
| multiply-double | Multiply two double-precision floating point numbers |
| **Class Access Benchmarks** | |
| instance-var | Read an integer instance variable |
| method-local | Invoke a method defined in the current (`this`) object |
| method-remote | Invoke a method defined in a different object |
| method-interface | Invoke an interface method |
| **Exception Handling Benchmarks** | |
| exception-local | Throw and catch an exception within the same method |
| exception-caller | Throw an exception caught by method's caller |
| exception-remote | Throw an exception caught by a method ten levels up the call chain |
| exception-bypass | Throw and catch an exception past an exception handler that does not catch the thrown exception |
| **Synchronization Benchmarks** | |
| sync-block-single | Enter a `synchronized` block in a single-threaded program |
| sync-method-single | Call a `synchronized` method in a single-threaded program |
| sync-block-multi | Enter a `synchronized` block in a multi-threaded program |
| sync-method-multi | Call a `synchronized` method in a multi-threaded program |
| **Miscellaneous Benchmarks** | |
| null-loop | Once around an empty loop |
| array-assign | Assign to an element of an integer array |
| thread-yield | Perform yields in 3 separate threads |
| write-small | `Write` 1 byte to a file |
| write-big | `Write` 10,000 bytes to a file |

Table 3: Micro-Benchmarks

seconds on JDK and 76 seconds on Guava to only 43 seconds on Toba. The average execution times of the benchmarks, plus standard deviations, are given in Figure 10.

Toba-generated code is faster than Sun's interpreter because compiling class files removes the overhead of interpretation and of dynamic loading. Toba-generated code is faster than Guava's because Toba does not incur code generation costs at run time, and, possibly, because the C compiler optimizes code more aggressively than does Guava's JIT compiler. For stand-alone applications that do not rely on dynamic loading, Toba provides large performance benefits over other systems.

## 7.3 Micro-benchmarks

Table 3 describes the micro-benchmarks used to isolate the performance differences in the systems. These benchmarks are an expanded version of the UCSD Java Microbenchmarks [GP96].

Table 4 shows results of running the benchmarks on each system. For accurate timing, each micro-benchmark was iterated in a loop until the total execution time was at least 5 seconds. This varied between 100 and 100,000,000 iterations, depending on the benchmark.

The results show that Toba outperforms the other systems on almost all benchmarks. For example, Toba is 13–32 times faster than JDK on the arithmetic and class-access benchmarks; this is directly attributable to JDK's use of an

| Benchmark | Toba ($\mu$sec.) | JDK ($\mu$sec.) | JDK/Toba | Guava ($\mu$sec.) | Guava/Toba |
|---|---|---|---|---|---|
| add-int | 0.12 | 3.81 | 32 | 0.17 | 1.4 |
| multiply-int | 0.20 | 4.44 | 22 | 0.22 | 1.1 |
| add-double | 0.23 | 5.37 | 23 | 0.62 | 2.7 |
| multiply-double | 0.24 | 4.91 | 20 | 0.55 | 2.3 |
| instance-var | 0.15 | 3.28 | 22 | 0.20 | 1.3 |
| method-local | 0.28 | 3.66 | 13 | 0.44 | 1.6 |
| method-remote | 0.33 | 4.84 | 15 | 0.49 | 1.5 |
| method-interface | 1.84 | 5.27 | 2.9 | 1.72 | 0.9 |
| exception-local | 1.74 | 6.68 | 3.8 | 16.79 | 9.7 |
| exception-caller | 8.05 | 14.29 | 1.8 | 27.71 | 3.4 |
| exception-remote | 9.45 | 57.92 | 6.1 | 141.29 | 15 |
| exception-bypass | 9.39 | 14.71 | 1.6 | 29.73 | 3.2 |
| sync-block-single | 2.22 | 18.11 | 8.2 | 7.05 | 3.2 |
| sync-method-single | 3.18 | 19.73 | 6.2 | 8.14 | 2.6 |
| sync-block-multi | 4.94 | 20.47 | 4.1 | 7.20 | 1.5 |
| sync-method-multi | 6.29 | 20.49 | 3.3 | 7.39 | 1.2 |
| null-loop | 0.04 | 1.33 | 30 | 0.08 | 2 |
| array-assign | 0.24 | 5.93 | 25 | 0.29 | 1.2 |
| thread-yield | 11.40 | 119.43 | 10 | 58.52 | 5.2 |
| write-small | 31.70 | 62.66 | 2.0 | 33.01 | 1.0 |
| write-big | 800 | 700 | 0.9 | 900 | 1.1 |

Table 4: Micro-Benchmark Timings

interpreter, as Guava is nearly as fast as Toba on these benchmarks.

Toba is also 2–15 times as fast as the other systems at handling exceptions. This is because Toba does not explicitly unwind the stack when an exception is thrown. Instead, Toba implements exception handling via `goto` or `setjmp/longjmp`, depending on whether the handler is within the same method or not. This makes exception handling in Toba extremely fast.

Synchronization is also fast in Toba, particularly in single-threaded programs because Toba optimizes monitor accesses in this situation. Although single-threaded programs need no synchronization, they may still make use of library classes that use synchronization.

Toba performs slightly worse than Guava on the interface-method invocation benchmark, and slightly worse than JDK on the "write-big" benchmark. Toba dominated both on all other programs, large and small.

### 7.4 Code Size

Toba emits naive C code and relies on an optimizing C compiler to do register allocation, copy propagation, and branch elimination to produce efficient code. Table 5 indicates the sizes of the benchmark programs in bytes of class file, lines of C, and bytes of object code. Object code sizes do not include the Toba run-time system, which is a dynamic shared library. This library contains 915,000 bytes of code.

## 8 Project Status

The Toba system currently runs under Solaris on SPARC workstations. The system includes all of the Java API except for dynamic linking and the graphics and applet libraries. Table 6 summarizes the sizes and implementation languages of its various components.

We intend to port Toba to additional architectures and operating systems. Porting Toba will require thread-specific changes to the run-time system and garbage collector. It will also require OS-specific changes to the run-time system.

| Benchmark | Class-file (bytes) | Emitted C Code (lines) | Object File (bytes) |
|---|---|---|---|
| JavaLex | 84,457 | 25,238 | 231,816 |
| JavaCUP | 119,094 | 50,297 | 446,816 |
| javac | 508,916 | 127,678 | 869,756 |
| espresso | 295,281 | 83,098 | 674,008 |
| Toba | 77,718 | 23,570 | 195,836 |
| JHLZip | 26,754 | 8,380 | 117,368 |
| JHLUnzip | 26,515 | 8,298 | 116,384 |

Table 5: Program Sizes

| Component | Implementation Language | Size (Lines) |
|---|---|---|
| Bytecode Translator | Java | 3891 |
| Run-time Support | C | 2680 |
| API Native Routines | C | 2484 |
| Toba-specific Garbage Collection | C | 30 |

Table 6: Implementation Details

The bytecode translator and header files will change only minimally.

Toba is the first piece of the larger "Sumatra" project. The Sumatra project is exploring many aspects surrounding the efficient execution of mobile code, with emphasis on efficient implementations of the Java Virtual Machine. We developed Toba to bootstrap our development of the JVM API, threads, and garbage collector, as well as to have fast Java applications.

# 9 Related Work

Java is a relatively new programming language and virtual machine. We know of no published results describing implementation and performance characteristics. Popular-press reports and commercial advertisements indicate that many development efforts for Just-In-Time (JIT) compilers are underway or have recently completed, but the available information is sketchy.

Compiling higher-level languages to C is not new. Many language systems leverage existing compilers and use C as an intermediate language in the compilation process. Systems for Smalltalk [Git94], SR [And82], Scheme [Bar89], Icon [Wal91], Forth [EM96], SML [TAL90], Pascal [Gil90], Cedar [ADH+89], and Fortran [FGMS90] are well known. For traditionally compiled languages like Pascal and Fortran, translation to C improved portability. For Scheme, Forth, and Icon, translation removed interpretation overhead. Similarly, Toba removes interpretation overhead from Java programs.

Several other projects for compiling Java bytecodes to C are currently underway. `j2c` [And96] is a restricted bytecode to C compiler, currently ported to several platforms. `j2c` (version 1 beta 5) does not support threads, monitors, or network resources. In addition, native routines cannot throw exceptions in `j2c`. Toba does not have these restrictions.

Vortex[DDG+96] is another project that compiles Java bytecodes to C. Vortex provides front ends for C++, Cecil, Modula-3, and Java. These languages are compiled to a common internal representation, and C code is generated from this representation. The Vortex project studies the effectiveness of optimizations for object-oriented languages. The Vortex project reports that Java programs speed up by as much as a factor of 8 as a result of these aggressive optimizations. Toba does not currently perform any of these optimizations. Vortex does not support threads, which has a global impact on performance. No published information is available about other details of Java run-time system support from Vortex.

`Jolt` [Sir96] also compiles Java bytecodes to C. `Jolt` generates a C function for some methods in a class file, and then generates a new class file with these methods marked as native. Method overloading is not supported, and `Jolt` cannot compile class initialization methods. `Jolt` produces class files that are used by the standard Java interpreter. Toba produces stand-alone executables.

## 10 Availability

The Toba system is freely available via anonymous ftp. All distribution information is described on the World Wide Web at `http://www.cs.arizona.edu/sumatra/toba/`.

## References


[ADH+89] Russ Atkinson, Alan J. Demers, Carl Hauser, Christian Jacobi, Peter Kessler, and Mark Weiser. Experiences creating a portable Cedar. pages 322–329, 1989.

[And82] G. R. Andrews. The distributed programming language SR—mechanisms, design and implementation. *Software—Practice and Experience*, 12:719–753, 1982.

[And96] Yukio Andoh. j2c. URL: http://www.webcity.co.jp/info/andoh/java/j2c.html, 1996.

[Bar84] J.G.P. Barnes. *Programming in Ada*. Addison-Wesley Publishing Company, 1984. ISBN 0-201-13799-2.

[Bar89] Joel F. Bartlett. Scheme→C a portable Scheme-to-C compiler. Technical Report DEC-WRL-89-1, Digital Equipment Corporation, Western Research Lab, 1989.

[BW88] Hans Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, pages 807–820, September 1988.

[DDG+96] Jeffrey Dean, Greg DeFouw, David Grove, Vassily Litvinov, and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of OOPSLA'96*, October 1996.

[EM96] M. Anton Ertl and Martin Maierhofer. Translating Forth to efficient C. 1996.

[FGMS90] S. I. Feldman, D. M. Gay, M. W. Maimone, and N. L. Schryer. A Fortran-to-C converter. Computing Science Technical Report No. 149, AT&T Bell Laboratories, Murray Hill, NJ, 1990.

[Gil90] Dave Gillespie. p2c. p2c is one of several publicly available Pascal to `C` compilers, 1990.

[Git94] Claus Gittinger. Smalltalk/x. Smalltalk/X is a widely available Smalltalk to C compiler, 1994.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, 1996. ISBN 0-201-63451-1.

[GP96] William G. Griswold and Paul S. Phillips. Microbenchmarks for `java`. URL: http://www-cse.uscd.edu/wgg/JavaProf/javaprof.html, 1996.

[GYT96] James Gosling, Frank Yellin, and The Java Team. *The Java Application Programming Interface*, volume 1. Addison-Wesley Publishing Company, 1996. ISBN 0-201-63453-8.

[LY97] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997. ISBN 0-201-63452-X.

[MDP96] David Mosberger, Peter Druschel, and Larry L. Peterson. Implementing atomic sequences on uniprocessors using rollforward. *Software—Practice and Experience*, 26:1–23, 1996.

[MW91] H. Mossenbock and N. Wirth. The programming language Oberon-2. Technical report, Institute for Computer systems, ETH, 1991.

[Sir96] KB Siram. Jolt. URL: http://substance.blackdown.org/~kbs/jolt.html, 1996.

[Str86] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986. ISBN 0-201-12078-X.

[TAL90] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Nov 90.

[Wal91] Kenneth Walker. The implementation of an optimizing compiler for Icon. Technical Report TR 91-16, University of Arizona, August 1991.

# A A Larger Example

Figure 11 expands on the example shown earlier by adding exception handling. An implicit branch (from the `try` block to the `return`) has also been added.

Figure 12 gives Toba's translation into C code. Exception handling has enlarged the code significantly, and the effect is especially noticeable because the original example was so small. Besides the boilerplate code that is the same for all exception-catching methods, there are also assignments to `pc` that maintain the JVM program counter and `case` labels used for dispatching a caught exception.

```
class d {
   static int div(int i, int j) {
      try {
         i = i / j;
      } catch (ArithmeticException e) {
         i = j;
      }
      return i;
   }
}
```

```
Method int div(int,int)
   0 iload_0
   1 iload_1
   2 idiv
   3 istore_0
   4 goto 10
   7 pop
   8 iload_1
   9 istore_0
  10 iload_0
  11 ireturn
Exception table:
   from   to  target type
     0     4     7   <Class java.lang.ArithmeticException>
```

Figure 11: Sample Java Program and Bytecode

```
Int div_ii_3WIeN(Int p1, Int p2)                                int div(int, int)
{
static struct handler htable[] = {                              exception handler list
      &cl_java_lang_ArithmeticException.C, 0, 4, 1,             go to L1 if 0 ≤ pc < 4
};
struct mythread *tdata;                      thread data pointer
jmp_buf newbuf;                              jump buffer
void *oldbuf;                                pointer to previous buffer
volatile int pc;                             JVM program counter
int tgt;                                     jump target
Int rv;                                      return value
Object a0, a1, a2;                           reference stack
Int i0, i1, i2;                              integer stack
volatile Int iv0, iv1;                       integer variables

      iv0 = p1;                              initialize variables from parameters
      iv1 = p2;

      tdata = mythread();                    set thread data pointer
      oldbuf = tdata->jmpbuf;                save old jmpbuf pointer
      tgt = 0;                               dispatch first to entry point
      if (setjmp(newbuf)) {                  set up jump buffer
          sthread_got_exception();           exception was caught:
CATCH:    a1 = tdata->exception;             load exception value
          if ((tgt = findhandler(htable, 1, a1, pc)) < 0)     find handler
              longjmp(oldbuf, 1);            no handler; pass upward
          }
      tdata->jmpbuf = newbuf;                register jump buffer for thread

TOP:  switch(tgt) {                          dispatch entry, ret, or exception

L0:   case 0:
      pc = 0;                                set pc for exception handling
      i1 = iv0;              iload_0
      i2 = iv1;              iload_1
      if (!i2)               idiv
          throwDivisionByZeroException();
      i1 = i1 / i2;
      iv0 = i1;              istore_0
      pc = 4;                                reset pc on leaving exception range
      goto L2;               goto 10

L1:   case 1:
      pc = 7;                                reset pc after catching exception
      i1 = iv1;              iload_1
      iv0 = i1;              istore_0
L2:   case 2:
      i1 = iv0;              iload_0
      rv = i1;               ireturn
      goto RETURN;

}
RETURN:
      tdata->jmpbuf = oldbuf;                restore previous jump buffer
      return rv;                             return result
}
```

Figure 12: Sample Toba Output