

Optimizing Java-specific Overheads: Java at the Speed of C?

Ronald S. Veldema, Thilo Kielmann, and Henri E. Bal

Division of Mathematics and Computer Science
Faculty of Sciences, Vrije Universiteit
{rveldema,kielmann,bal}@cs.vu.nl
<http://www.cs.vu.nl/manta/>

Abstract. Manta is a highly optimizing compiler that translates Java source code to binary executables. In this paper, we discuss four Java-specific code optimizations and their impact on application performance. We assess the execution time of three application kernels, comparing Manta with the IBM JIT 1.3.0, and with C-versions of the codes, compiled with GCC. With all three kernels, Manta generates faster code than the IBM JIT. With two kernels, the Manta versions are even faster than their C counterparts.

1 Introduction

Java has become increasingly popular as a general-purpose programming language. Key to Java's success is its intermediate "byte code" representation that can be exchanged and executed by Java Virtual Machines (JVMs) on almost any computing platform. Along with Java's widespread use, the need for a more efficient execution mode has become apparent. A common approach is to compile byte code to executable code. Modern JVMs come with a just-in-time compiler (JIT) that combines Java's platform independence with improved application speed [4]. A more radical approach is to completely avoid byte-code interpretation by statically compiling byte code to executable programs [1, 7, 12]. Combinations of JIT and static compilation are also possible [11].

A disadvantage of byte-code compilation, however, is the computational model of the byte code which implements a stack machine [14]. Mapping this (virtual) stack machine on existing CPUs (that are register based) is harder than directly generating register-oriented code. Our Manta compiler thus follows a different approach and translates Java *source code* to executable programs. Manta performs code optimizations across the boundaries of Java classes and source files by using a temporary database of the intermediate code of all classes of an application program. With this large base of information, the Manta compiler can generate very efficient code. In addition to generating efficient sequential code, a source-level compiler allows us to add slight language extensions, like new special interfaces, for purposes of parallel computing [8, 13].

Until recently, many scientific codes have been implemented in the C language, mainly for reasons of efficiency. From a software-development point-of-view, it is desirable to use Java instead of C, allowing to use features like threads, objects, exceptions, runtime type checks, and array bounds checks in order to ease programming and

debugging. Unfortunately, several of these features introduce performance overheads, making it difficult to obtain the same sequential speed for Java as for C. In this paper, we discuss to what extent these Java-specific overheads can be optimized away. We implemented a range of existing compiler optimizations for Java, and study their performance impact on application kernels.

2 The Manta Compiler

The Manta compiler is part of the Manta high-performance Java system. Intended for parallel computing, Manta also provides highly-efficient communication mechanisms like remote method invocation (RMI) [9] and object replication [8]. Figure 1 illustrates the compilation process. Manta directly translates Java source code to executables for the Intel x86 platform. The supported operating system is Linux. For extensive program analysis and optimization, the Manta compiler relies on its intermediate code data base. The availability of intermediate code for all classes of an application allows optimizations across the borders of classes and source files. Manta's efficient RMI mechanism comes with a compatibility mode to Sun's RMI, allowing Manta programs to communicate with other JVMs. For this purpose, the exchange of byte code is necessary. Manta uses *javac* to generate byte code which is also stored in the executable program for the sole purpose of sending it to a JVM along with an RMI. For receiving byte code, Manta executable programs contain a just-in-time compiler that compiles and dynamically links new classes into running programs. The treatment of byte code is described in [9]. In this paper, we focus on the generation and optimization of sequential programs by the Manta compiler.

The Manta compiler implements several standard code optimization techniques like common-subexpression elimination and loop unrolling [10]. The intermediate-code data base (see Figure 1) allows extensive, inter-procedural analyses, even across several classes of a Java application. In the following, we discuss four code optimization techniques (object inlining, method inlining, escape analysis, and bounds-check elimination) and two programmer assertions (closed-world assumption and bounds-check deactivation) that are related to Java's language features. Manta allows its optimizations and assertions to be turned on and off individually via command-line options.

2.1 Closed-world Assumption

Many compiler optimizations require knowledge about the complete set of Java classes that are part of an application. Java's polymorphism, in combination with dynamic class loading, however, prevents such optimizations. In this case, the programmer has to explicitly annotate methods as *final* in order to enable a large set of optimizations.

However, the *final* declaration has only limited applicability as it selectively disables polymorphism. Its use for improving application performance furthermore contradicts its original intention as a means for class-hierarchy design. Fortunately, many (scientific) high-performance applications consist of a fixed set of classes and do not use dynamic class loading at all. Such applications can be compiled under a *closed-world assumption*: all classes are available at compile time. The Manta compiler has a

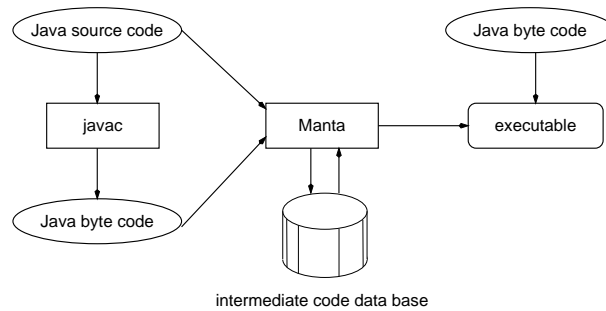


Fig. 1. The Manta compilation process.

command line option by which the programmer can assert or deny the validity of the closed-world assumption.

2.2 Object Inlining

Java's object model leads to many small objects with references to other objects. For improving application performance it is desirable to aggregate several small objects to a larger one, similar to C's struct mechanism. Such an aggregation is called *object inlining* [6]. Performance is improved by reducing overheads from object creation, garbage collection, and pointer dereferencing. The following code fragment shows an example of object inlining. When Manta can derive (via proper *final* declarations or the closed-world assumption) that the array *a* is never reassigned in objects of class *A* (left), then the array can be statically inlined into objects of class *A* (right). Note that the shown optimization can not be implemented manually because Java lacks a corresponding syntactical construct for arrays. In order to allow proper use, objects are inlined along with their header information, including, for example, vtables for method dispatch. For feasibility of the analysis, Manta inlines only objects created directly in field initializations.

<pre>class A { int [] a = new int[10]; }</pre>	<pre>class A { int a[10]; }</pre>
original class <i>A</i> with separate array object	<i>A</i> with inlined array

2.3 Method Inlining

In C-like languages, function inlining is a well-known optimization for avoiding the costs of function invocation. In object-oriented programs it is common to have many, small methods, making method inlining desirable for Java. For any object, the methods of its most-specific class have to be invoked, even after casting an object reference to a less specific class. These program semantics, which are at the core of object-oriented programming, prevent efficient method inlining. Only proper *final* declarations or the closed-world assumption enable efficient method inlining. However, in the presence of

polymorphism and dynamic class loading, methods can be safely inlined if they are either declared as *static* or if the compiler can statically infer the object type. In the following example, the method *inc* would be an ideal candidate for inlining due to its small size. It can be inlined if the compiler can safely derive that there exists no subclass of *A* that replaces the implementation of *inc*. Manta does not inline methods in the presence of *try/catch* blocks. Also, methods are only inlined if they do not exceed 200 assembly instructions or 20 recursively inlined methods.

```
class A {  int a;
          void inc() { a++; }
          void other() { inc(); } }
```

2.4 Escape Analysis

Escape analysis considers the objects created by a given method [2, 5]. When the compiler can derive that such an object can never *escape* the scope of its creating thread (for example, by assignment to a static variable), then the object becomes unreachable after the method has terminated. In this case, object allocation and garbage collection can be avoided altogether by creating the object on the stack of the running thread rather than via the general-purpose (heap) memory. In the case of creation on the stack, method-local objects can be as efficient as function-local variables in C.

One problem with stack allocation of objects is the limited stack size. To resolve this, Manta maintains separate *escape stacks* that can grow independently of the call stack, while keeping the local objects apart from the general allocation and garbage-collection mechanism. To further minimize the size of the escape stacks, Manta detects repetitive creation of temporary objects in loop iterations and re-uses escape stack entries if possible. In general, Manta implements escape analysis using backward interprocedural data-flow analysis combined with heap analysis.

2.5 Array Bounds-check Deactivation and Elimination

The violation of array boundaries is a frequently occurring programming mistake with C-like languages. To avoid these mistakes, Java requires array bounds to be checked at runtime, possibly causing runtime exceptions [3]. This additional safety comes at the price of a performance penalty. A simple-minded, but unsafe optimization is to suppress the code generation for array-bounds checks altogether. The idea is that boundary violations will not occur after some successful, initial program testing with bounds checks activated. Completely deactivating array-bounds checks thus gives the unsafety of C at the speed of C. Manta has a command-line switch by which the programmer can assert that array-bounds checks can be completely deactivated.

A safe alternative to bounds-check deactivation is implemented in the Manta compiler. Inside the code of a method, Manta can safely eliminate those bounds checks that are known to repeat already successfully passed checks. For example, if a method accesses $a[i]$ in more than one statement, then only the first access needs a bounds check. For all other accesses, the checks can safely be omitted as long as Manta can derive from the code that neither the array base a nor the index i have been changed in the

meantime. For this purpose, Manta performs a data-flow analysis, keeping track of the array bases and related sets of index identifiers for which bounds checks have already been issued [10]. The current implementation, however, does not yet perform this data-flow analysis across method boundaries. Currently, method invocation between array accesses empties the affected sets of already-checked array identifiers but does propagate this information to the called methods.

3 Application Kernel Performance

We investigated the performance of the code generated by Manta using three application kernels. For each kernel, we used two versions, one written in Java and one written in C. Both versions were made as similar to each other as possible. For each application, we compare the runtimes of the Manta-compiled code (with all assertions and optimizations turned on) with the same code run by the IBM JIT 1.3.0 and with the C version, compiled by GCC 2.95.2 with optimization level “-O3”. We thus compare Manta with the (Linux) standard C compiler and the most competitive JIT. Unfortunately, performance numbers for other static Java compilers were not available to us.

Manta’s best application speeds have been obtained asserting the closed-world assumption, enabling object inlining, method inlining, and escape analysis, while deactivating array-bounds checks. For evaluating the impact of the individual optimizations, we also provide runtimes with one optimization disabled while keeping all others turned on. Because the closed-world assumption has a strong impact on the other optimizations, we present runtimes for the other optimizations both with and without closed-world assumption. For array-bounds checks, we present runtimes comparing deactivation (included in the best times) with elimination, and general activation. All runtimes presented have been measured on a Pentium III, running at 800 MHz, using the Linux operating system (RedHat 7.0).

3.1 Iterative Deepening A* (IDA*)

Iterative Deepening A* (IDA*) is a combinatorial search algorithm. We use IDA* to solve random instances of the 15-puzzle (the sliding-tile puzzle). The search algorithm maintains a large stack of objects describing possible moves that have to be searched and evaluated. While doing so, many objects are created dynamically. Table 1 lists the runtimes for IDA*. The code generated by Manta, with all assertions and optimizations turned on, needs about 8.3 seconds, compared to 19.5 seconds with the IBM JIT. For IDA*, we actually implemented two C versions. One version keeps all data in static arrays and completes in 4 seconds. A second version emulates the behavior of the Java version by eagerly using *malloc* and *free* to dynamically allocate the search data structures. This version needs 15.6 seconds, almost twice as long as the Manta version. The comparison of the two C versions shows that applications written without object-oriented structure can be much faster than a Java-like version of the same code. However, with similar behavior of the application versions, Java programs can run as fast as C, or even faster.

As can also be seen from Table 1, the most efficient optimization for IDA* is object inlining which drastically reduces the number of dynamically created objects. Under the closed-world assumption, the other optimizations have hardly any impact, except for deactivating array bounds checking. The versions in which array bounds are either active or partially eliminated need 0.5 seconds longer than the best version that completely deactivates all array-bounds checks. Without the closed-world assumption, object inlining is not effective, so the runtimes are in the same order as with completely disabling object inlining.

Table 1. IDA* (15-puzzle), application runtimes in seconds

compiler	best time	Manta, optimizations turned off individually		
			closed world	no closed world
IBM JIT 1.3.0	19.524	no object inlining	26.486	26.516
Manta, best	8.277	no method inlining	8.935	26.661
GCC 2.95.2 -O3	3.959	no escape analysis	8.915	26.561
GCC, malloc/free	15.591	bounds-check elimination	8.717	26.458
		bounds-check activation	8.850	26.631

3.2 Traveling Salesperson Problem (TSP)

TSP computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one specific city. We use a branch-and-bound algorithm which prunes a large part of the search space by ignoring partial routes that are already longer than the current best solution. We run TSP with a 17-cities problem.

The compute-intensive part of the application is very small. It consists of two nested loops that iterate over an array containing the visited paths. As can be seen from Table 2, method inlining is the most effective optimization, by inlining the method calls inside the nested loops. The elimination of array bounds checks produces code that is as fast as the unsafe version with deactivated bounds checks. However, keeping all (unnecessary) bounds checks active, adds more than one second to the completion time. With its optimizations, the Manta-generated code needs only 4.5 seconds, compared to 4.7 seconds of the C version and 6.3 seconds with the JIT. The TSP example shows that Java programs can be optimized to run at a speed that is competitive to C.

3.3 Successive Overrelaxation (SOR)

Red/black SOR is an iterative method for solving discretized Laplace equations on a grid. We run SOR with 1000×1000 grid points. The compute-intensive part of the code runs a fixed number of array convolutions with floating-point arithmetic. As Table 3 shows, the most effective optimization is (safe) elimination of array-bounds checks. All other optimizations have hardly any impact on the overall speed. Manta's code completes after 1.5 seconds, slightly faster than the C version with 1.7 seconds, and substantially faster than the JIT version, which needs 4.4 seconds for the same problem.

Table 2. TSP (17 cities), application runtimes in seconds

compiler	best time	Manta, optimizations turned off individually		
		closed world	no closed world	
IBM JIT 1.3.0	6.307	no object inlining	4.513	4.530
Manta, best	4.511	no method inlining	7.158	7.181
GCC 2.95.2 -O3	4.696	no escape analysis	4.567	4.530
		bounds-check elimination	4.511	4.525
		bounds-check activation	5.612	5.612

Table 3. SOR (1000×1000 grid points), application runtimes in seconds

compiler	best time	Manta, optimizations turned off individually		
		closed world	no closed world	
IBM JIT 1.3.0	4.387	no object inlining	1.583	1.542
Manta, best	1.541	no method inlining	1.547	1.548
GCC 2.95.2 -O3	1.768	no escape analysis	1.579	1.541
		bounds-check elimination	1.569	1.548
		bounds-check activation	2.380	2.379

4 Conclusions

The Java language has several properties that make efficient execution more challenging than for C. Java programs typically use many small, dynamically created objects and short methods, resulting in high overheads. Also, Java is a safe language and thus requires array-bounds checking. In this paper, we investigated to what extent these overheads can be eliminated using compiler optimizations. We implemented four existing optimizations in the same compiler framework (Manta, a native source-to-binary compiler). We studied the impact of the optimizations on three application kernels. The results show that, with all optimizations switched on, two of the three applications run faster than C. Object inlining and method inlining each had a high impact on one application. Array bounds-check elimination saved about 35 % for the third application. This safe elimination leads to code which is almost as efficient as unsafe bounds-check deactivation for all three applications. Escape analysis was less effective. Allowing the compiler to use a closed-world assumption (i.e., disallow dynamic class loading) was shown to be essential for certain optimizations like object inlining. To summarize, the performance of application programs is typically dominated by a “bottleneck” which differs from program to program, and which has to be addressed by a specific optimization. In general, to execute Java programs at the speed similar to C versions, many optimizations, both standard techniques and Java-specific optimizations, have to be provided by a compiler.

Interesting future work will assess larger benchmarks (like the SCIMARK suite). Further tests will evaluate elements of Manta’s runtime system like the thread package and the garbage collector.

Acknowledgments

The development of Manta is supported in part by a USF grant from the Vrije Universiteit. We thank Rutger Hofman, Cerial Jacobs, Jason Maassen, and Rob van Nieuwpoort for their contributions to the Manta compiler and runtime system. We thank John Romein and Kees Verstoep for keeping our computing platform in good shape.

References

1. G. Antoniu, L. Bougé, P. Hatcher, M. MacBeth, K. McGuigan, and R. Namyst. The Hyperion system: Compiling multithreaded Java bytecode for distributed execution. *Parallel Computing*, 2001. To appear.
2. B. Blanchet. Escape Analysis for Object Oriented Languages. Application to Java. In *Proc. OOPSLA'99*, pages 20–34, Denver, CO, Nov. 1999.
3. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proc. ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI 2000)*, pages 321–333, Vancouver, BC, June 2000.
4. M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM 1999 Java Grande Conference*, pages 129–141, San Francisco, CA, June 1999.
5. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape Analysis for Java. In *Proc. OOPSLA'99*, pages 1–19, Denver, CO, Nov. 1999.
6. J. Dolby. Automatic inline allocation of objects. In *Proc. 1997 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 7–17, June 1997.
7. R. Fitzgerald, T. Knoblock, E. Ruf, B. Steensgard, and D. Tarditi. Marmot: An optimizing compiler for Java. Technical report 33, Microsoft Research, 1999.
8. J. Maassen, T. Kielmann, and H. E. Bal. Parallel Application Experience with Replicated Method Invocation. *Concurrency and Computation: Practice and Experience*, 2001.
9. J. Maassen, R. van Nieuwpoort, R. Veldema, H. E. Bal, and A. Plaat. An Efficient Implementation of Java's Remote Method Invocation. In *Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'99)*, pages 173–182, Atlanta, GA, May 1999.
10. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 1997.
11. M. Serrano, R. Bordawekar, S. Midkiff, and M. Gupta. Quicksilver: A Quasi-Static Compiler for Java. In *Proc. OOPSLA'00*, pages 66–82, Minneapolis, MN, Oct. 2000.
12. V. Seshadri. IBM High Performance compiler for Java. *AIXpert Magazine*, Sept. 1997. <http://www.developer.ibm.com/library/aixpert>.
13. R. V. van Nieuwpoort, T. Kielmann, and H. E. Bal. Satin: Efficient Parallel Divide-and-Conquer in Java. In *Proc. Euro-PAR 2000*, number 1900 in Lecture Notes in Computer Science, pages 690–699, Munich, Germany, Aug. 2000. Springer.
14. B.-S. Yang, S.-M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. C. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'99)*, 1999.