Java for Large-Scale Scientific Computations?

Andreas Krall[0] and Philipp Tomsich[1]

[1] Institut für Computersprachen, Technische Universität Wien
Argentinierstraße 8, A–1040 Wien, Austria
andi@complang.tuwien.ac.at
[2] Compilers and Tools, Silicon Graphics, Inc.
1600 Amphitheatre Parkway, Mountain View, CA 94043
ptomsich@sgi.com

Abstract. The Java programming language has its origins in the development of portable internet applications, that are interpreted on the client machine. However, a number of software projects have adopted it as the language of choice for a wide variety of applications, including numerically intensive scientific computing. Given its heritage, the suitability of Java for such application domains remains questionable, which is reflected in large number of users reporting poor performance compared to native compilers for C or Fortran.
At heart, Java is an object-oriented language enabling the rapid development of modular and maintainable programs. It provides an integral security model and features array bounds checking, arbitrarily shaped arrays, a deterministic floating-point arithmetic on all platforms, automatic memory management using garbage collection, multi-threaded execution and a portable byte code representation. These features ease the development of scientific applications but may hinder efficient execution of the applications. This article shows state of the art compilation techniques addressing these language features to achieve optimal performance. Efficient solutions for a large number of performance problems encountered in the past are available in the current generation of Java compilers. We may thus conclude that a maturing Java is suited for large scale scientific applications.

1 Introduction

The Java programming language is an object-oriented, general-purpose language which has its origins in the development of portable internet applications. It features simple object semantics, cross-platform portability, arbitrarily shaped arrays and security. All this is highly desirable from a software engineering viewpoint and increases programmer productivity. However, these features also come at a cost: they negatively impact the performance of Java applications. Object-orientation requires additional indirections and introduces method dispatching overheads, while index checks increase overall run-time.

The object-oriented programming model employed by Java leads to large numbers of small methods and lightweight classes causing excessive overheads,

even when operating on objects that have value semantics such as complex numbers. The arbitrary shape of arrays and the fact that the Java language does not support arrays of rank greater than one leads to the implementation of multidimensional arrays as arrays-of-arrays, requiring multiple dereferencing and address calculations during element accesses. This compares very unfavorably—particularly for scientific applications that depend on true multidimensional arrays—to the rectangular arrays used in C or Fortran, where the offset for each element can be calculated statically and only a single memory access needs to be executed. Type checking at run-time introduces additional, significantly sized instruction sequences and multiple branch instructions on most current implementations. Further, the Java language specification mandates a deterministic behavior of floating-point arithmetic among all platforms, thus effectively disallowing the use of hardware-depended optimizations and extensions, such as fused multiply-add instructions. As a result, the performance of numerically intensive, scientific application executed using commercial Java environments can be as low as one percent of equivalent Fortran programs[MMG+00].

Interpretation was the first available choice for Java implementations and helped accelerate early Java adoption due to its rapid retargetability. Its poor performance, however, quickly led to the proliferation of just-in-time (JIT) compilers, which translate bytecode to native code at run-time, executing and caching it. As the time required for compilation is added to the overall execution time of a program, time-consuming optimizations must be used sparingly or replaced by less costly algorithms. Code quality remains poor in comparison to traditional compilers due to this design requirement of fast compilation. Most modern systems with JIT compilers can be described as mixed-mode, in that they combine an interpreter with a JIT compiler: the interpreter runs initially and collects profiling information, and performance-critical methods are identified and compiled as execution progresses[SOY+00]. This in turn allows the creation of compilers that implement more costly optimizations, since they are invoked more selectively. Extending this approach further, multiple compilers at different levels of sophistication can be employed within a single virtual machine [DA97].

With the growing importance of Java for long-running computationally intensive applications and the continuing demand for higher performance than that provided by early JIT compilers, implementors attempted to leverage existing mature compiler infrastructure for Java either by translating Java to C or by connecting a Java front end to a common optimization and compilation back end. The result of this approach is a system that compiles Java to native code ahead-of-time (AOT). Such compilers can produce completely static standalone executables, or they can work within the context of a traditional virtual machine which also supports interpretation or just-in-time compilation of dynamically loaded bytecode. Examples include TowerJ[Tow], MS Marmot[FKR+00], Compaq Swift[SRGD00] and the NaturalBridge[Nat] compilers. Ahead-of-time compilers offer the hope of higher performance than what is available with traditional virtual machines and JIT compilers. Achieving the performance of natively

compiled code while maintaining compatibility with the dynamic aspects of Java remains a promise[SSTP00].

2 Optimization techniques for Java

Active research into efficient implementation techniques for Java has yielded a number of optimization techniques to reduce the overheads associated with bounds checking for Java arrays, multidimensional arrays, Java floating-point semantics and run-time type checking.

2.1 Removing array bound checking

Removing of array bound checking is the most important optimization for scientific programs. Up to 40% of the run-time can be spent in array bound checking code. Different algorithms with increasing complexity have been designed for JIT and AOT compilers.

The CACAO JIT compiler has been designed for extremely short compilation times [Kra98]. It consequently does not support profiling. Therefore, to find array access instructions which are worthwhile candidates for removal, loop analysis is performed. For array access instructions inside loops the variables in simple loop expressions are analyzed and their possible range is computed for simple index modifications. If it can be determined that the index variables lies in the correct range, the array bound check is either removed, or moved before the loop (eventually copying the loop for correct exception behavior). This algorithm increases the compile time from 118 to 176 milliseconds for `javac`, but reduces the run-time by 33% for the `sieve` benchmark.

ABCD is a light-weight algorithm for elimination of bound checks on demand by Bodik et. al. [BGS00]. ABCD works by adding a few edges to the SSA value graph and performing a simple traversal of the graph. ABCD works on a sparse representation and requires on average fewer than 10 simple analysis steps per bound check. On the benchmarks ABCD removes on average 45% of dynamic bound check instructions, sometimes achieving near-optimal optimization.

The Sable research group presented a framework for optimizing Java using attributes [PQVR+00]. The array bound check analysis collects constraints of nodes and propagates them along the control flow graph until a fixed point is reached. The information is stored in attributes and used by interpreters and compilers. Between 26% and 59% of the bound checks can be removed and performance is improved by 5.8% to 35.6% in the IBM high performance compiler.

2.2 Optimized multidimensional arrays

For scientific and engineering computations, multidimensional rectangular arrays are the most important data structure. While the Java language does not directly support arrays of rank greater than one, arrays-of-arrays can be constructed which are far more flexible, but do not offer a dense representation. While this

allows the definition of ragged arrays and even of arrays which alias some of their rows, it renders it impossible to calculate an element position within a multidimensional array in Java using a base address, shape information and indices only. Unfortunately this generality weighs down on performance even for numerical applications that do not require these rich array semantics.

While it is difficult to replace the default array implementation used within the virtual machine, it is easy to extend the Java run-time environment with natively implemented classes to provide optimized multidimensional arrays with more favorable semantics[MMG+00]. Such arrays are accessed through Java objects that encode the index ranges and reference a flat memory space which contains a dense representation of the array elements. When the array is constructed, its shape is specified and remains immutable afterwards. Indexing elements is done cheaply using a direct address calculation and a single memory fetch.

2.3 Floating-point optimizations

The original Java language specification required a deterministic behavior of floating-point arithmetic on all platforms as specified in IEEE 754. In particular, Java requires full support of IEEE 754 denormalized floating-point numbers and gradual underflow. This specification disallowed hardware specific optimizations like fused multiply-add operations or the use of higher precision arithmetic like Intels 80bit arithmetic.

These inefficiencies lead to a change in the Java 2 language specification as implemented with JDK 1.2. A modifier `strictfp` was added to specify methods and classes which strictly have to follow the IEEE 754 standard. The default mode was changed to non-strict and a strict mathematic library was added. Non-strict operations are allowed to use a higher precision extended arithmetic.

Early models of Alpha processors handle infinities and NaNs in software using imprecise exceptions. To enable an IEEE compliant floating-point behavior a trap barrier instruction has to be placed between two floating point instructions or before the end of a basic block leading to an increase in code size and slower execution speed. The CACAO just-in-time compiler has a global switch which can disable IEEE compliant behavior raising an exception whenever NaNs or infinities occur.

2.4 64 bit Java virtual machines

The Java virtual machine is specified as a 32 bit stack machine. For this reason, and because a few bytecodes such as `pop2` or `dup2` introduce difficulties in their definition for 64 bit Java interpreters, the initial Java execution environments were exclusively available as 32bit binaries. Unfortunately this contradicts the requirements of scientific applications that operate on large data sets and large arrays, as these need the larger address space provided by a 64bit implementation. Many specialized libraries used in data analysis and other scientific applications are available as 64 bit binaries only and can therefore not interface

with the JavaVM through the Java native interface. Further, many 32 bit Java VMs that are available on 64bit architectures cause unaligned loads and stores which significantly impact performance.

Today an increasing number of 64bit JVMs is becoming available. CACAO [KG97] was the first 64bit JVM, although available for the Alpha architecture only. Compaq later released a commercial-quality 64bit JavaVM and finally Sun is planning for 64 bit support in the JDK 1.4 release.

2.5 Multithreading

Java threads independently execute code that operates on values and objects residing in a shared memory. Java threads on multiprocessor systems efficiently can be used to execute parallelized large scale scientific applications. Performance problems can happen if threads need synchronization to allow safe access to shared data. To tackle this performance bottleneck two approaches have to be combined: efficient implementation of synchronization and elimination of synchronization.

In [KP98] we showed that inefficient implementation of synchronization can lead to huge a performance degradation. We presented a fast space efficient solution where monitors are implemented in a hash table. For multiprocessor system the shared hashtable can be a bottleneck. Therefore, Bacon et. al. presented thin locks which allocate a 24bit monitor data structure within every object [BKMS98]. If one word is used to store the monitor on average the size of an object is increased by 17% for `javac` (a Java to byte code compiler) and by 0.6% for `linpack` (a linear algebra package). Whereas for average Java programs the increase of object size is high, it is negligible for scientific applications.

Ruf [Ruf00] presents an effective technique for removing unnecessary synchronization operations from statically compiled Java programs. His analysis can eliminate synchronization operations even on objects that escape their allocating threads. For the benchmark programs examined 100% synchronization operations are removed in single-threaded programs and 0–99% synchronization operations are removed in multi-threaded programs.

2.6 Garbage collection

Automatic memory management—or more precisely garbage collection—is an integral aspect of the Java language. Unfortunately it is rather difficult to design a garbage collector that performs equally well for different allocation patterns, as seen with interactive user-interface applications and scientific workloads, respectively. One of the problems arising from the object-orientation of the Java language and the resulting allocation patterns is the fact that objects remain alive for a very long time or are extremely short-lived.

Early VM implementations used either a non-moving mark-and-sweep or a mark-and-compact collector. While simpler in design, the mark-and-sweep collectors were believed to cause heap fragmentation in the context of Java. While

our work[KT99] in the context of the CACAO VM has shown that mark-and-sweep collectors are a viable solution for Java, recent developments have lead to the use of generational garbage collection algorithms, which operate separately on young object and mature objects. The main benefit reaped from this is that only a small fraction of all objects survive more than one garbage collection and can thus be evicted early. As a result the mature generation stays small and for the young generation a garbage collection algorithm may be chosen that performs best when few objects survive. However, using generational garbage collection introduces the penalty of requiring a write barrier that keeps track of intergeneration pointers.

2.7 Efficient run-time type checking

For every type cast or execution of an `instanceof` operator run time type checking has to be done. Static analysis is not very effective in eliminating these cast checks [GRS00]. Therefore, efficient run-time type checking is very important.

A type check tests whether one type is a subtype of another. A subtype test is trivially implemented by traversing a data structure representing the supertypes of the type. For classes this data structure is a simple list, for interfaces it is a directed acyclic graph. Although very efficient constant time type checking algorithms exist [VHK97], most of the currently available JVMs use some variations of the simple algorithm caching one or two supertypes [SOY+00].

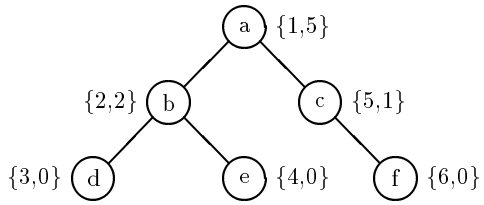

Fig. 1. Relative numbering with {`baseval`, `diffval`} pairs

CACAO uses different very fast constant time subtype tests for classes and interfaces which easily supports dynamic class loading. The subtype test for classes is implemented by relative numbering. Two numbers *low* and *high* are stored for each class in the class hierarchy. A depth first traversal of the hierarchy increments a counter for each class and assigns the counter to the *low* field when the class is first encountered and assigns the counter to the *high* field when the traversal leaves the class. A class is a subtype of another class, if the $super.low \leq sub.low < super.high$. Since a range check is implemented more efficiently by an unsigned comparison, CACAO stores the difference between the *low* and *high* values and compares it against the difference of the *low* values of both classes. The code for `instanceof` looks similar to:

```
return (unsigned) (sub->vftbl->baseval - super->vftbl->baseval) <=
       (unsigned) (super->vftbl->diffval);
```

For leaf nodes in the class hierarchy the `diffval` is 0 which results in a faster test. A JIT compiler can generate the faster test for final classes. An AOT compiler may additionally replace the `baseval` of the superclass by a constant.

CACAO stores an interface table at negative offsets in the virtual function table. This table is needed for the invocation of interface methods. This table is additionally used by the subtype test for interfaces. If the table is empty for the index of the superclass, the subtype test fails. The code for `instanceof` looks similar to:

```
return (sub->vftbl->interfacetable[-super->index] != NULL);
```

Both subtype tests can be implemented by very few machine code instructions without using branches which are expensive on modern processors.

3 Conclusion

While a large amount of anecdotal evidence regarding the low performance of Java exists, reality is quickly improving. The current generation of Java just-in-time compilers includes increasingly sophisticated optimizations, which reduce the overheads caused by the modern language features offered by Java: array bound check elimination, optimized multidimensional arrays, optimized floating-point arithmetic, synchronization elimination, efficient run-time type checking.

The availability of ahead-of-time compilers is also promising, as they can ignore some of the more dynamic aspects of the language and generate highly optimized executables for production runs. While Java still is not the perfect environment for scientific computing, major steps towards a competitive performance for numerically intensive applications have been made in the last few years and some applications already achieve 90 percent of the performance of native Fortran implementations[MMG+00]. Today, whether or not to choose Java for a particular scientific application mostly reduces to making a decision between performance and improved programmer productivity and maintainability.

References

[BGS00] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bound checks on demand. In *Conference on Programming Language Design and Implementation*, volume 35(5) of *SIGPLAN*, pages 321–333, Vancouver, 2000. ACM.

[BKMS98] David F. Bacon, Ravi Konuru, Chet Mruthy, and Mauricio Serrano. Thin locks: Featherweight synchronization for Java. In *Conference on Programming Language Design and Implementation*, volume 33(5) of *SIGPLAN*, pages 258–268, Montreal, 1998. ACM.

[DA97] David Detlefs and Ole Agesen. The Case for Multiple Compilers. In *Proc. OOPSLA 1999 VM Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, 1997.

[FKR+00] Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgaard, and David Tarditi. Marmot: an optimizing compiler for Java. *Software – Practice and Experience*, 30(3):199–232, November 2000.

[GRS00] Sanjay Gehmawat, Keith H. Randall, and Daniel J. Scales. Field analysis: Getting useful and low-cost interprocedural information. In *Conference on Programming Language Design and Implementation*, volume 35(5) of *SIGPLAN*, pages 334–344, Vancouver, 2000. ACM.

[KG97] Andreas Krall and Reinhard Grafl. CACAO – a 64 bit JavaVM just-in-time compiler. *Concurrency: Practice and Experience*, 9(11):1017–1030, 1997.

[KP98] Andreas Krall and Mark Probst. Monitors and exceptions: How to implement Java efficiently. *Concurrency: Practice and Experience*, 10(11–13):837–850, 1998.

[Kra98] Andreas Krall. Efficient JavaVM just-in-time compilation. In Jean-Luc Gaudiot, editor, *International Conference on Parallel Architectures and Compilation Techniques*, pages 205–212, Paris, October 1998. IFIP,ACM,IEEE, North-Holland.

[KT99] Andreas Krall and Philipp Tomsich. Garbage collection for large memory Java applications. In *Proc. of the 7th European Conference on High-Performance Computing and Networking (HPCN Europe'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 895–907. Springer Verlag, Apr 1999.

[MMG+00] J. E. Moreira, S. P. Midkoff, M. Gupta, P. V. Artigas, M. Snir, and R. D. Lawrence. Java programming for high-performance numerical computing. *IBM Systems Journal*, 39(1):21–56, 2000.

[Nat] NaturalBridge. *BulletTrainTM optimizing compiler and runtime for JVM bytecode.* `http://www.naturalbridge.com`.

[PQVR+00] Patrice Pominville, Feng Quian, Raja Vallee-Rai, Laurie Hendren, and Clark Verbrugge. A framework for optimizing Java using attrributes. In *CASCON*, Mississauga, 2000. IBM.

[Ruf00] Erik Ruf. Effective synchronization removal for Java. In *Conference on Programming Language Design and Implementation*, volume 35(5) of *SIGPLAN*, pages 208–218, Vancouver, 2000. ACM.

[SOY+00] T. Suganuma, T. Ogasawara, M. TaT. Yasuekeuchi, , M. Kawahito, K. Ishizaki, and H. Komatsuatani. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[SRGD00] Daniel J. Scales, Keith H. Randall, Sanjay Ghemawat, and Jeff Dean. The Swift compiler: Design and implementation. Technical Report 2000/2, Compaq Western Research Laboratory, April 2000.

[SSTP00] Todd Smith, Suresh Srinivas, Philipp Tomsich, and Jinpyo Park. Practical experiences with Java compilation. In *Proceedings of the Intl. Conf. on High-Performance Computing*, volume 1970 of *Lecture Notes in Computer Science*. Springer, December 2000.

[Tow] Tower Technologies. *TowerJ 3.0: A New Generation Native Java Compiler And Runtime Environment.* `http://www.towerj.com`.

[VHK97] Jan Vitek, Nigel Horspool, and Andreas Krall. Efficient type inclusion tests. In Toby Bloom, editor, *Conference on Object Oriented Programming Systems, Languages & Applications (OOPSLA '97)*, pages 142–157, Atlanta, October 1997. ACM.