High-Performance Java Codes for Computational Fluid Dynamics*

Christopher Riley† cjriley@cs.unc.edu
Siddhartha Chatterjee† sc@cs.unc.edu
Rupak Biswas* rbiswas@nas.nasa.gov

†Department of Computer Science
The University of North Carolina
Chapel Hill, NC 27599-3175

*NAS Systems Division, MS T27A-1
NASA Ames Research Center
Moffett Field, CA 94035

ABSTRACT

The computational science community is reluctant to write large-scale computationally-intensive applications in Java due to concerns over Java's poor performance, despite the claimed software engineering advantages of its object-oriented features. Naive Java implementations of numerical algorithms can perform poorly compared to corresponding Fortran or C implementations. To achieve high performance, Java applications must be designed with good performance as a primary goal. This paper presents the object-oriented design and implementation of two real-world applications from the field of Computational Fluid Dynamics (CFD): a finite-volume fluid flow solver (LAURA, from NASA Langley Research Center), and an unstructured mesh adaptation algorithm (2D_TAG, from NASA Ames Research Center). This work builds on our previous experience with the design of high-performance numerical libraries in Java. We examine the performance of the applications using the currently available Java infrastructure and show that the Java version of the flow solver LAURA performs almost within a factor of 2 of the original procedural version. Our Java version of the mesh adaptation algorithm 2D_TAG performs within a factor of 1.5 of its original procedural version on certain platforms. Our results demonstrate that object-oriented software design principles are not necessarily inimical to high performance.

1. INTRODUCTION

The Java programming language has many features that are attractive for both general-purpose and scientific computing. Among them are: support for object-oriented concepts such as inheritance, encapsulation, and polymorphism that allow the abstraction of common physical concepts and the development of reusable class libraries for them; the architecture-neutrality of Java byte code, which enables portability across multiple platforms; garbage collection, which simplifies memory management; and language-level support for multithreading, which allows parallel applications to be developed more easily in Java. However, scientific programs written in Java usually run slower than corresponding programs written in Fortran and C, due either to the intrinsic overhead of these features or to the relative immaturity of current Java Virtual Machine (JVM) implementations. Because high performance is a primary concern in computational science, that community has been reluctant to adopt Java as the language of choice for numerical applications.

To demonstrate the viability of high-performance computing in Java and to encourage its greater adoption in the computational science community, several authors have ported numerical libraries to Java [1, 5, 19], written Fortran-to-Java translators [8, 10], developed compilation technology for Java [6, 7, 25], and written class libraries to address deficiencies in the Java language for numerical computing [26]. Although these studies demonstrate the potential of Java for high-performance computing, several factors limit their usefulness in determining whether large-scale scientific applications written in Java can achieve high performance. First, numerical and class libraries form only part of such applications. The only previous ports of large-scale codes to Java that we know of are a geophysical simulation by Jacob et al. [20] and a parallel multi-pass renderer by Yamauchi et al. [31]. Second, "line-by-line" translations of procedural codes written in Fortran or C to Java do not exploit object-oriented techniques, one of the main advantages to programming in Java.

The primary goal of the work described in this paper is to demonstrate that realistic scientific applications can be written in Java that make full use of the language's object-oriented capabilities and still show good performance on current standards-conforming JVM implementations. Secondary goals include characterizing the performance and identifying the bottlenecks in different JVM implementations, identifying design principles for portable high-performance object-oriented software in Java, and making available additional benchmarks for the high-performance Java community. It is our thesis that in order to achieve high performance with such applications, one must consciously design for performance in both the definition and the implementation of the software components that comprise them. We choose two real-world examples from the field of Computational Fluid Dynamics (CFD) for our study. The first example is the Langley Aerothermodynamic Upwind Relaxation Algorithm (LAURA) [9, 12], a finite-volume flow solver developed at NASA Langley Research Center for multiblock, structured grids. LAURA has been widely used to compute hypersonic, viscous, reacting-gas flows over reentry vehicles such as the Shuttle Orbiter [14], the Mars Pathfinder [24], and the X-33 Reusable Launch Vehicle [15]. The second example is the Two-Dimensional Triangular Adaptive Grid (2D_TAG) code [27] developed at NASA Ames Research Center for adaptation of unstructured meshes. This

*This work was supported in part by NSF Grant CCR-9711438.

Java 2001 Palo Alto, CA USA

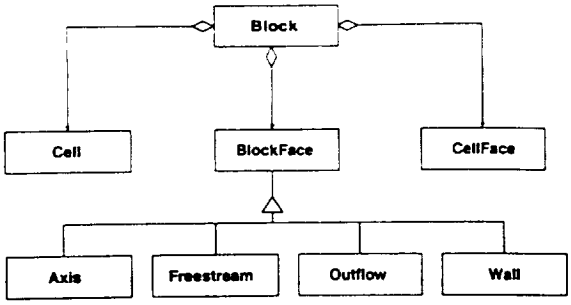

Figure 2: Block class abstraction.

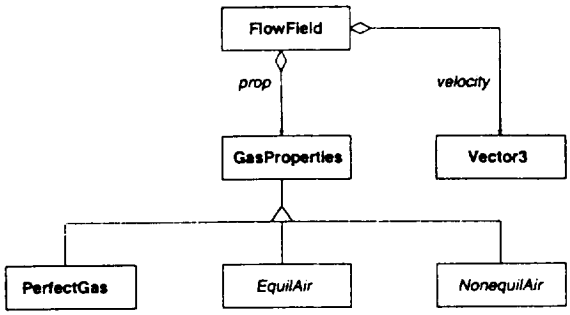

Figure 3: Flowfield class abstraction.

BlockFace, to minimize the effects of adding new chemistry models and boundary conditions to the software. The *GasProperties* class is subclassed for each particular chemistry model. Currently, only the perfect gas model is supported in the Java version as it is the only gas model available in the version of LAURA released to us by NASA Langley Research Center. The *BlockFace* class is subclassed for each of four different boundary conditions. Diagrams showing the geometry and flowfield class abstractions are given in Figures 2 and 3. The *EquilAir* and *NonequilAir* classes pictured in Figure 3 are not currently implemented, but illustrate how subclassing would be used to extend the abstract *GasProperties* class in a full-featured version of LAURA. Where possible, only references to the abstract base classes are made.

2.2.1 Design Patterns

Gamma et al. [11] define *design patterns* as "descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context." Our Java code uses the *Factory* creational pattern to manage the construction of the various geometric objects and the *Singleton* creational pattern to control access to atmospheric constants. However, there are additional design patterns that we did not use. The *Iterators* pattern would be useful in sweeping over a block, but the multidimensional array package we use does not contain them. The *Strategy* pattern would be useful in managing changes to the choice of relaxation algorithms. We did not use it in our version because the original Fortran version of LAURA supports only one relaxation algorithm.

2.2.2 Multithreading

We added multithreading by subdividing a single block into multiple blocks and assigning a Java thread to each sub-block. This follows the domain decomposition strategy used to parallelize the original Fortran version of LAURA for distributed machines [28]. In this multithreaded version, two *Block* objects will share a single

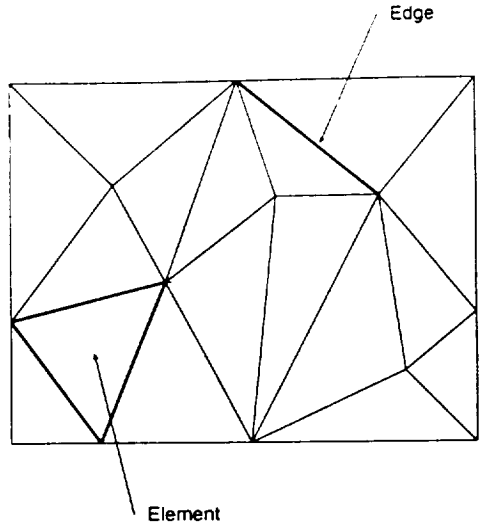

Figure 4: An example of an unstructured mesh.

BlockFace object. Conflicts may occur if each *Block* tries to modify the shared *BlockFace* at the same time. Java provides a monitor locking mechanism with the `synchronized` keyword for performing mutual exclusion on an object. Use of `synchronized` for the appropriate methods is the safest approach to handling this block interface conflict in that it prevents an inconsistent update of the boundary cells and cell faces at the boundary between two blocks. However, by using the point-implicit relaxation strategy, blocks may be relaxed asynchronously. Blocks do not need to be kept in lock step at the same iteration level. For this algorithmic reason, our code does not use the `synchronized` keyword.

3. 2D_TAG

2D_TAG [4, 27] is a two-dimensional, unstructured mesh adaptation scheme that locally refines and/or coarsens the computational grid. It is written in C and is a simplified version of a corresponding three-dimensional algorithm. Biswas and Strawn [4] describe the details of the algorithm. Oliker and Biswas [27] discuss its performance under different parallel processing strategies.

It is important to note here the different roles of a mesh adaptation algorithm like 2D_TAG and a flow solver like LAURA. A mesh adaptation code is a tool used to support the flow solver by concentrating the solver's computation in regions of interest such as shocks and shear flows. During the computation of a steady-state flow field, the mesh is adapted infrequently, on the order of once every several hundred solver iterations. In this scenario, the mesh adaptation may consume 5% of the total running time. For time-dependent calculations, the mesh is adapted much more frequently and may account for 30–40% of the total computation time. Good performance of both the solver and of the mesh adaptation scheme is therefore desirable.

3.1 Algorithm

A mesh is described as a collection of two-dimensional triangular elements, as shown in Figure 4. The mesh is *unstructured* in that there is no logical ordering of an element and its neighbors as with the structured grids used with LAURA. The mesh is *locally adapted*. This involves adding points to the existing grid in regions where some user-specified error indicator is high, and removing points from regions where the indicator is low. The advantage of

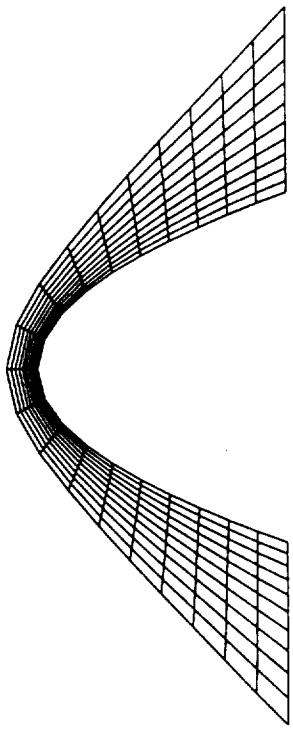

Figure 6: Side view of the 10×10×10 grid around the paraboloid.

Mesh	*Vertices*	*Triangles*	*Edges*
Initial	14,605	28,404	43,009
Level 1	26,189	59,000	88,991
Level 2	62,926	156,498	235,331
Level 3	169,933	441,147	662,344
Level 4	380,877	1,003,313	1,505,024
Level 5	488,574	1,291,834	1,935,619

Table 2: Progression of grid sizes through five levels of adaptation.

We use the computational mesh over an airfoil to test the performance of 2D_TAG. This is the same test case used by Oliker and Biswas [27]. For an actual flow simulation over an airfoil traveling at transonic Mach numbers, shocks form on both the upper and lower surfaces of the airfoil. The mesh is typically refined in the area containing the shocks as well as around the stagnation point located at the leading edge of the airfoil. This scenario is simulated by geometrically refining the mesh in these regions. The actual test case consists of reading the initial coarse mesh into 2D_TAG and proceeding through five levels of refinement. Table 2 gives the resulting grid sizes at each level of refinement. Note that the final mesh is more than 40 times larger than the initial one. Figure 7 shows a close-up view of the initial mesh.

4.2 Testing Environment

Table 3 lists the platforms and JVMs we used for measuring performance. All JVMs were run using Just-In-Time (JIT) compilation and with garbage collection enabled. All machines were relatively unloaded at the time of each test, and several runs were made at each test condition with the best time recorded. The initial and maximum heap sizes were set using the `-Xms` and `-Xmx` options to values on the order of 500 MB to discourage excessive garbage collection. The typical variation in run times was less than 1 second for run times on the order of 60 seconds. All JVMs are freely available except for Jalapeño [2, 7], a research JVM developed at IBM T.J. Watson Research Center. Although the Jalapeño JVM contains an optimizing JIT compiler, it is not tuned for scientific codes. For instance, it does not perform traditional optimizations for scientific computing such as loop unrolling. Jalapeño also has an adaptive compiler using dynamic feedback[3] that may recompile methods at different optimization levels during the course of a code's execution. We chose to use Jalapeño's optimizing compiler instead, because it provided us with more control over optimization options.

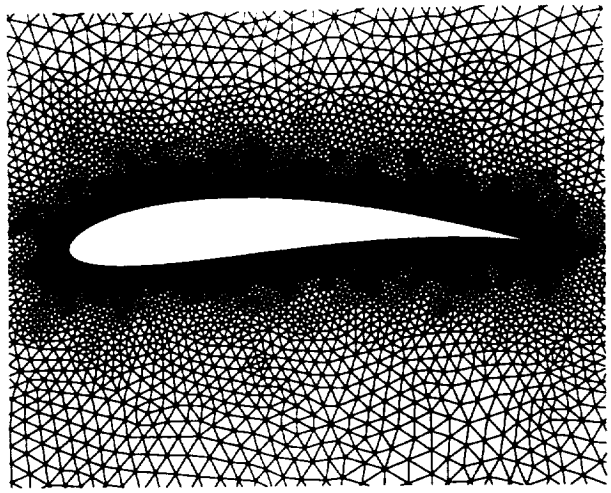

Figure 7: Close-up view of the initial mesh around the airfoil.

4.3 LAURA Results

Tables 4 and 5 give the performance of the single-precision, native and Java versions of LAURA on the different platforms. Running times are normalized by the total number of cells and iterations. The times are normalized to help show variations in running time with grid size and to help determine running time for different grid sizes and iteration counts. Recall that the C version of LAURA (see Section 2.1) does not contain the copying to and from temporary work arrays that is present in the original Fortran version. We consider the performance of the C version to be the benchmark against which the performance of the Java version should be compared.

Several observations regarding the timing measurements are in order.

1. As shown in Table 5, the performance of the Java version of LAURA is within a factor of 3 of the corresponding C version on the Sun Ultra and the Pentium when using the Sun JVM and within a factor of 2.5 on the Pentium when using the IBM JVM. The fastest Java version on the PowerPC with the Jalapeño JVM is still 3 times slower than the native C version. The focus of the Jalapeño JVM research project has not been on optimization of scientific codes and floating point calculations.

2. The relative performance of the Java version on the SGI is

JVM	*Grid*	IBM [26]	Colt [18]	Native Java
Sun JDK 1.3.0	10×10×10	49.3	51.6	51.7
	20×20×20	55.8	57.5	58.0
	40×40×40	60.0	64.9	64.7
IBM JDK 1.3.0	10×10×10	39.4	39.4	39.5
	20×20×20	39.8	40.0	40.2
	40×40×40	44.0	45.8	46.2

Table 6: Time per cell per iteration (μsec) of LAURA-Java using different array packages on Pentium.

is more difficult in that many of the array bounds checks may have already been removed by the `-O1` and `-O2` optimization levels. Because a run with the `-O0 -no-bounds-checks` options was not made, we hesitate to draw definitive conclusions from the data except to say that array bounds checks (difference between `-O2` and `-O2 -no-bounds-checks` times) account for at least 4% of the total running time.

4.3.1 Array Packages

We next examine the effect of the multidimensional array package on the performance of LAURA. Table 6 lists the performance of several versions of LAURA on the Pentium, each using a different array package to manage the collections of geometric objects found in a block. The performance of LAURA using native Java arrays is also listed. The conventional wisdom is that native Java arrays are inefficient due to the array bounds checking that Java requires and the possible lack of data locality of the Java "array of arrays" [26]. While this statement is certainly true, it must be evaluated in terms of the actual application. Unlike many other CFD applications, we are not using multidimensional arrays in LAURA to store primitive floating-point values such as pressure, velocity, and density, nor are we using arrays to perform matrix factorization or multiplication. In the case of LAURA, we are using multidimensional arrays as containers of *Cell*, *CellFace*, and *Point* objects which in turn store the primitive data. Most of the running time is spent in getting this data from memory to the registers and then executing floating point operations with them. The time spent in `get` and `set` operations in the various array packages is small in comparison. Therefore, the choice of array packages has little impact on the performance of LAURA.

4.3.2 Multithreaded Performance

Figure 8 shows the multiprocessor speedup of the Java version of LAURA on the 4-processor Pentium machine and the 12-processor PowerPC machine. As mentioned in Section 2.2.2, our Java version of LAURA does not use `synchronized` methods due to the asynchronous relaxation allowed by the point-implicit relaxation strategy. With only minimal modifications to the code, we are able to achieve near ideal speedup using the Jalapeño JVM on the PowerPC and a modest speedup of around 3 on the 4-processor Pentium machine.

4.4 2D_TAG Results

Tables 7 and 8 list the performance of the C and Java versions of 2D_TAG on the various architectures. The "Startup" time represents the time spent reading in the original mesh and allocating and initializing relevant data structures. The "Adapt" time represents the time spent in refining the mesh. The total running time of a version is the sum of these times. The "Adapt Time Ratio" for

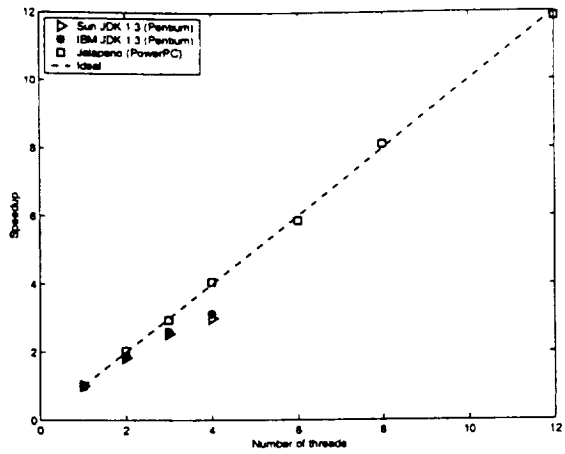

Figure 8: Speedup of LAURA-Java on 4-processor Pentium and on 12-processor PowerPC with Jalapeno. The grids used are 24×24×24 (Pentium) and 32×32×32 (PowerPC).

Machine	Startup (*sec*)	Adapt (*sec*)
Sun Ultra*	1.60	4.01
SGI	0.77	7.89
Pentium	0.58	3.75
PowerPC	1.94	7.17

Table 7: Running times of C version of 2D_TAG for five levels of refinement. Note that the grid is refined only three times on the Sun Ultra due to memory constraints.

a Java version is the ratio of its adaptation time to the adaptation time of the original C version. Smaller values of this ratio indicate better performance of the Java code. The "GC" time represents the time spent in garbage collection by the Java versions, as indicated by the `-verbosegc` flag. Due to memory constraints, the initial mesh is refined only three times on the Sun Ultra instead of five. We tested three Java versions: a procedural version obtained by a "line-by-line" translation of the original C version; an object-oriented (O-O) version as described in Section 3.2 that uses Java arrays to store *Element*, *Edge*, and *Vertex* objects; and an O-O version that uses linked lists to store the collections of mesh objects. Using linked lists avoids array bounds checks and also avoids the need to preallocate the storage for the final refined mesh. Because the original C version as well as the other Java versions use arrays for storage, they require the final mesh size to be specified at run time. The difference in performance between the three Java versions gives an imperfect indication of the performance "cost" of writing object-oriented code.

Generally, the Java versions perform quite well, with the O-O versions running within 6% of the performance of native code on the PowerPC using the Jalapeño JVM and within 33% on the Pentium with the IBM JVM. In fact, the O-O versions actually outperform the Java procedural version on the PowerPC, SGI, and Sun Ultra. Examining the hardware counters on the SGI using `perfex` shows an increased number of Translation Lookaside Buffer (TLB) misses in the procedural version. We conjecture that the O-O version does a better job of achieving locality in the TLB by encapsulating data in the *Vertex*, *Edge*, and *Element* classes.

One of the main differences between the procedural and O-O

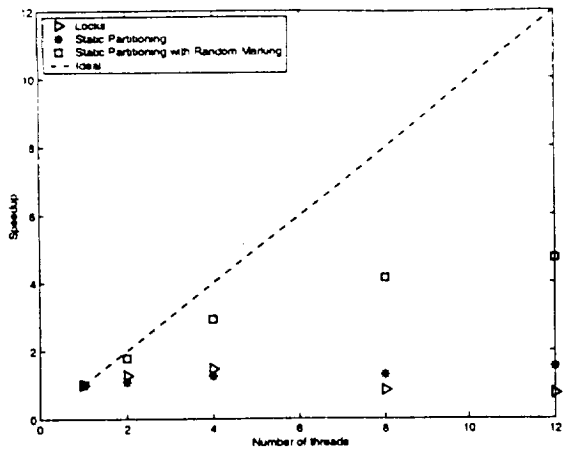

Figure 9: Speedup of 2D_TAG on 12-processor PowerPC with Jalapeno.

static partitioning time is 0.20 *sec* on the PowerPC which is small compared to the startup and adaptation times of 2D_TAG (see Table 8) and is not included in the total running time when computing speedup. The speedup using this strategy is only slightly better than with using locks, reaching a maximum speedup of 1.54 when using 12 threads. The poor performance is not due to excessive synchronization but due to load imbalance. Because we are only partitioning once, the computational load becomes increasingly imbalanced with each level of refinement. The solution is to repartition the refined mesh at each step. To simulate more favorable load balancing, we ran 2D_TAG using "Static Partitioning with Random Marking". Edges are marked for refinement randomly and not according to a error criteria. The submeshes remain approximately the same size during the five levels of refinement, and this fact accounts for the improved speedup, which reaches 4.71 when using 12 threads. Including repartitioning times would increase the running times and reduce these measured speedups somewhat. This is not a realistic scenario, but in the absence of repartitioning, does demonstrate the parallel performance of the multithreaded Java version of 2D_TAG on similarly-sized submeshes.

5. CONCLUSIONS AND FUTURE WORK

We have described the design, implementation, and evaluation of object-oriented "100% Pure" Java codes for two representative CFD applications that differ radically in their characteristics. The flow solver LAURA is structured, largely static, and floating-point intensive. The mesh adaptation algorithm 2D_TAG is unstructured, very dynamic, fine-grained, and limited by the speed of object manipulation. We have demonstrated that LAURA's running time is almost within a factor of 2 and 2D_TAG's is within a factor of 1.5 of their optimized native, procedural counterparts using current off-the-shelf Java compilers and JVM technology. We feel that this level of performance is extremely promising and is susceptible to further improvement as Java technology matures.

Performance should of course be considered in the larger context of the software design cycle. In both codes, the Java versions are smaller in size than the original versions: 5300 lines of Java to 14500 lines of Fortran for LAURA, and 1300 lines of Java to 1900 lines of C for 2D_TAG. In addition, the Java versions are much more modular and hierarchically structured. The Java version of LAURA contains 49 classes organized in a three-deep inheritance hierarchy, while the O-O version of 2D_TAG contains eight classes in a two-deep hierarchy. Such modularization makes it easier to extend the functionality of the code, e.g., to add different boundary conditions, gas chemistry options, or relaxation algorithms in LAURA. Programmer productivity also needs to be considered. The Java versions of both codes were designed and implemented in 18 months by a single programmer with minimal Java experience at the beginning of the effort. The original versions, by contrast, represent many person-years of effort.

The performance of the codes indicates the complex design tradeoffs in JVM implementations and highlights several aspects of JVM behavior that should be investigated further for high-performance Java. First, the user can control garbage collection activity only in very indirect ways, such as by specifying the initial and maximum heap sizes. The wide variety of GC algorithms, their different performance characteristics, and the black-box nature of this component of the JVM, make this level of user interaction inadequate for high-performance applications. Greater user interaction with the garbage collector within the constraints of the Java platform need to be investigated. Second, the performance of multithreading varies widely among platforms, and needs to be improved to scale to large numbers of threads.

Regarding future work, we plan to implement a parallel, three-dimensional version of the 2D_TAG code as well as continue our experimentation.

6. ACKNOWLEDGMENTS

We thank Peter Gnoffo of NASA Langley Research Center for providing us with a copy of the LAURA flow solver and Steve Fink, Peter Sweeney, and Michael Hind of IBM T.J. Watson Research Center for their permission and assistance in running our codes on the Jalapeño JVM.

7. REFERENCES

[1] G. Almasi, F. G. Gustavson, and J. E. Moreira. Design and evaluation of a linear algebra package for Java. In *Proceedings of the ACM 2000 Java Grande Conference*, pages 150–159, June 2000.

[2] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1), 2000.

[3] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive optimization in the Jalapeño JVM. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages & Applications (OOPSLA '00)*, pages 47–65, Oct. 2000.

[4] R. Biswas and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13:437–452, 1994.

[5] B. Blount and S. Chatterjee. An evaluation of Java for numerical computing. *Scientific Programming*, 7:97–110, 1999.

[6] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency: Practice and Experience*, 9(6):445–463, June 1997.

[7] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno dynamic optimizing compiler for Java. In