Just-In-Time Optimizations for High-Performance Java Programs

Michał Cierniak
Wei Li

Department of Computer Science
University of Rochester
Rochester, NY 14627
{cierniak,wei}@cs.rochester.edu

April 1997

Abstract

Our previous experience with an off-line Java optimizer [6] has shown that some traditional algorithms used in compilers are too slow for a JIT compiler. In this paper we propose and implement faster ways of performing analyses needed for our optimizations. For instance we have replaced *reaching definition* with *constant values* and *loop induction variables* with *loop-defined variables*. As the result Briki, our JIT compiler, is extremely fast, so that its running time is negligible even when the data sets used with our benchmarks result in execution times of a few seconds. The impact for the same benchmarks running on more realistic problem sizes would be even smaller.

Currently the speedups resulting from applying our optimizations are between 10% and 20%, but when the JIT compiler will perform standard optimizations which are absent in the current version of the JIT compiler used by us, the speedups should be similar to the ones observed for Fortran programs—about 50%.

1 Introduction

Compiling scientific applications requires sophisticated compiler optimizations. Those optimizations are commonly available in commercial compilers for languages traditionally used for high performance computing. Those languages (the most successful ones are Fortran and, to a smaller extend, C++) are compiled off-line into a stand-alone binary. This approach is different from the dominant compilation model for Java in which the source program is translated into machine-independent `class` files which are then interpreted or dynamically (just-in-time) compiled into the target machine language.

Many of the existing high-performance optimizations cannot be easily adapted to just-in-time compilation. The reasons include:

- Sophisticated optimization techniques consume a lot of hardware resources: CPU time and memory.
- Often, a global view of the program is necessary (*whole-program optimization* [2, 4]).

In this paper we address the issue of performing optimizations which are as good or almost as good as traditional optimizations while running much faster and using less memory. For our study we use an optimizations technique called *data transformations*. In previous work we have shown that data transformations can be very effective in compiling Fortran or C programs [3]. Recently we have adapted the technique for Java bytecodes [5, 6]. However, our previous experiments with optimizing Java bytecodes have been performed with an off-line compiler and the compiler algorithms were not as fast as a competitive JIT compiler is expected to be. To perform data transformations we need more program structure than is directly available in the bytecodes. Our previous implementation was recovering full high-level structure of the original Java program before applying any optimizations. This high-level view of the input bytecodes was represented with an intermediate representation called *JavaIR*. JavaIR is very high-level which means that high-performance optimizations, like data remapping, can be performed very efficiently. Unfortunately, our experience has shown that recovering full structure from a `class` file is very expensive. In our implementation, the time needed to convert bytecodes to JavaIR is an order of magnitude longer than the time to perform the optimizations. Since most the high-level structure is not needed to perform data transformation, we have decided to use a different approach in our JIT optimizer.

This paper shows how to perform the same optimizations as described in [5, 6] while recovering only as much structure as needed and using faster (although not necessarily as accurate) analysis techniques than those traditionally used in off-line compilers [1, 11].

2 Overview of Array Transformations

Array transformations change the layout of array elements in memory to increase the spatial locality of the application. Spatial locality benefits application by amortizing the cost of fetching a cache line over the uses of array elements co-located in that cache line.

As an example consider an array declared as

```
double A[][] = new double[n][m];
```

The Java language specification [7] and the Java Virtual Machine specification [8] do not define how an array is being laid out in memory. However, the natural implementation of a multi-dimensional Java array would represent `A` as shown in Figure 1. This representation is used by Kaffe and many other Java Virtual Machines.

In the loop nest

```
for(int i = 0; i < m; i++) {
  for(int j = 0; j < n; j++) {
    ... A[j][i] ...
  }
}
```

the reference to array `A` exhibits poor spatial locality. References to array elements which occupy consecutive words in memory, e.g., `A[0][0]` and `A[0][1]`, occur in consecutive iterations of the outer loop and are separated by a whole execution of the inner loop which accesses many words in memory and is very likely to replace the cache line fetched in the previous iteration of the outer loop. In our example, even though the reference to `A[0][0]` would bring into the cache the element `A[0][1]`, the reference to `A[0][1]` in the next iteration of the outer loop would suffer the penalty of a cache miss.

There exist two techniques for improving the spatial locality of the reference `A[j][i]` above: loop transformations and array transformations. For the simple example above either of those two would help

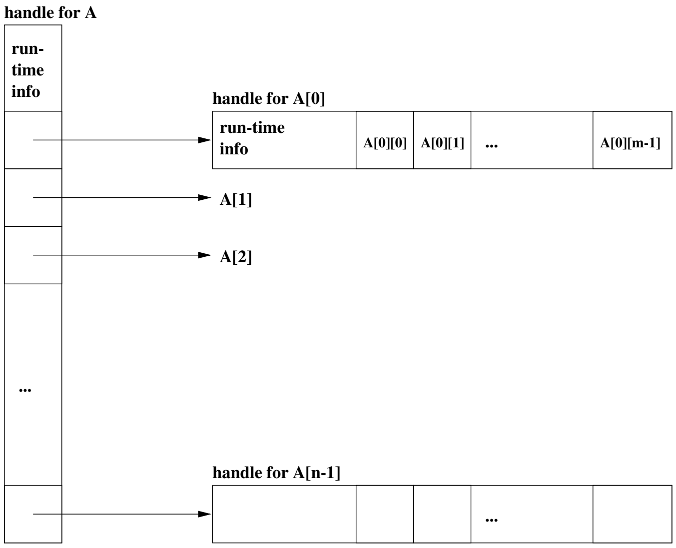

Figure 1: Internal representation of a Java array `double[n][m]`

(assuming that other parts of the program do not prevent us from applying loop or data transformation). However, we have demonstrated in [3] that none of those two techniques alone is sufficient for many real, scientific application. Applicability of loop transformations is beyond the scope of this paper—we focus here on array transformations.

Spatial locality of the loop nest discussed in this section can be improved by the array transformation which transposes the two dimensions of `A`. Our optimizations are performed in a JIT compiler without access to the source, but—for clarity—the transformation can be visualized as rewriting the source as:

```
double A[][] = new double[m][n];
...
for(int i = 0; i < m; i++) {
  for(int j = 0; j < n; j++) {
    ... A[i][j] ...
  }
}
```

The declaration of the array is changed as well as all references to that array. In the optimized code, two consecutive iterations of the inner loop access elements of `A` which occupy consecutive words in memory. For most applications this optimization will result in fewer cache misses and thus a better performance.

Optimization by data remapping must solve two problems:

- Ensuring the legality of an array transformation. Our array transformations permute dimensions of multidimensional array and therefore require the arrays to be rectangular. However, unlike in Fortran or C, a multidimensional array in Java does not need to be rectangular. Our compiler proves that an array is initialized with a rectangular shape and that it is not reshaped during its lifetime.

  The compiler must also make sure that if an array is accessed via different names due to aliasing, all references use the new, transformed type.

- Finding the optimal array layout. It is possible that different references to the same array require different transformations. The compiler must determine what is a globally optimal transformation (we do not allow dynamic reshaping of arrays).

Both issues are addressed in the next section.

3 Implementation

Performing high-level optimizations in a Java JIT compiler is more difficult than in a traditional compiler. The main reasons are:

- Much of the information about program structure is lost in the process of translation from Java source to bytecodes. This information (e.g., multidimensional array references) must be recovered before any optimizations can be attempted. In our particular implementation that recovery is even more difficult since, for performance reasons, we have decided to directly use the Kaffe IR which is even lower than bytecodes.
- Since the compiler is being invoked every time the program is being run, the speed of compilation is much more critical and many traditional analysis techniques must be replaced with faster alternatives. Such new algorithms are proposed in this section.
- Typically, to improve response time, a JIT compiler compiles bytecodes on demand one class at a time or one method at a time (Kaffe uses the latter approach). Therefore, optimizations which require global information are not possible (or at least cannot be trivially implemented).

Since our compiler is embedded in Kaffe we have decided to use Kaffe IR directly. A brief overview of the Kaffe architecture is presented in Section 3.1. Briki performs the optimizations in the following steps:

1. Build a control flow graph (CFG) for the method.
2. Find dominators.
3. Identify loops.
4. Compute def-use information.
5. Compute *constant values*.
6. Find *loop-defined variables*.
7. Identify multidimensional arrays which can be legally remapped.
8. Find optimal mappings.
9. Remap arrays.

The rest of this section will discuss our implementation decisions for the above algorithm.

3.1 Kaffe architecture

Kaffe [10] is a free JIT implementation of the Java Virtual Machine [1]. Kaffe runs on different architectures: i386, Sparc, Alpha, M68K. This is possible because Kaffe is designed to have two parts: the machine-independent front-end which translates bytecodes into a low-level intermediate representation, *Kaffe IR*, and a machine-dependent back-end which translates Kaffe IR into the required machine language.

Kaffe IR is designed to closely resemble a modern microprocessor architecture so that writing a new back-end is relatively simple. Kaffe IR defines a virtual architecture with a very large number of pseudoregisters (every local variable and every stack location is mapped to a different pseudoregister) and it contains instruction to move data between registers[2] or between registers and memory, perform arithmetic and logical operations on values in registers, branch conditionally to a label, call a subroutine, etc.

Every back-end must provide a set of core virtual machine instructions. If a virtual instruction cannot be mapped directly to the target instruction set, it is implemented in software. A back-end may also implement one of optional instructions which will be used by Kaffe to implement some bytecodes more efficiently.

A Kaffe IR (virtual) instruction is represented internally as a `sequence` structure with a pointer to the back-end function which takes as a parameter a pointer to the `sequence` and translates it into the target instruction set. The `sequence` structure also contains all operands (pseudoregisters, constants or labels) to that virtual instruction.

The translation process consists of two basic steps. First the bytecodes are translated by the front-end into the Kaffe IR, then the list of the `sequence` structures is traversed and every back-end function is called generating native code.

The developers of Kaffe are very good about releasing updates to Kaffe. In the 14 months since the Kaffe project started, there have been 21 releases of Kaffe (the latest one is 0.8.3), or—on average—one every 3 weeks. This caused a technical problem for us, since the most convenient and efficient way of implementing our optimizations would involve modifying internal data structures of Kaffe, but to keep in sync with the updates of Kaffe we would have to merge our software with the new Kaffe every three weeks thus stretching our resources. To avoid it, we have decided to keep our optimizer as separate from Kaffe as possible. While this made our project manageable, it resulted in extra overheads both in memory and processing time. In that light we are glad that our optimizations are very fast (see Section 4 for timings), because there is room for additional improvement which can be achieved by a tighter integration with Kaffe. Main sources of those, unnecessary, overheads are

- Need to allocate a shadow data structure for every `sequence` structure to keep additional information needed during optimization. Splitting the information into two structures results in lost spatial locality and unnecessary indirection.
- Need to recognize individual instructions in Kaffe IR. Since the only way to tell what instruction is implemented by a given `sequence` structure is to compare the back-end function pointer against the addresses of back-end functions extra time is spent where a single instruction would suffice if we could change Kaffe IR to suit our needs.

In the current implementation, we let Kaffe front-end generate its IR (the `sequence` structures), then we invoke our optimizer which analyzes and transforms Kaffe IR, and after the optimization is done, we return to Kaffe and the back-end translates the optimized Kaffe IR form into native code. That design means that

[1] Kaffe can also run as an interpreter, but we use it only in its JIT mode.

[2] We will use the terms *register*, *pseudoregister* and *variable* interchangeably.

the only change to Kaffe source code needed to plug in our otimizer is the insertion of a single function call just before the back-end invocation.

3.2 Control Flow Graph

We build the CFG for the Kaffe IR form of a method by searching for the `startBlock` and `endBlock` instructions. Than we use standard algorithms to find dominators and identify loops [1]. Those algorithms take about 10% of the total optimization cost (compare Figure 2).

3.3 Computing Def-Use Information

This step computes def-use information for every basic block. The operation is very simple and requires visiting every Kaffe IR instruction exactly once to perform a sequence of simple bit operations. In our implementation this the most expensive part of the array transformation optimization (compare Figure 2).

The long time to perform this optimization is an artifact of our software engineering decision explained in Section 3.1. For relative independence of briki from changes in Kaffe we pay the price of

- The need for keeping a shadow structure for every Kaffe IR instruction (which results in worse cache utilization and unnecessary indirection in accessing the structures).
- Very slow code to determine which registers are being modified in a given instruction.

We think that this time can be shortened when Kaffe becomes more stable and a tighter integration of Kaffe and Briki becomes possible.

3.4 Computing Constant Values

For every array reference we have to be able to identify which array allocation statement is associated with this reference. There are many known algorithms for solving this problem of *reaching definitions*. Existing algorithms are not well suited for JIT compilation since they use large amounts of memory to store the reaching definitions information (e.g., in the form of bitmaps for every instruction or *ud-chains* [1]).

For our purposes we solve a simpler problem which can be computed faster and represented in a more compact way. The representation contains one bit vector per basic block. The bit vector represents *constant values* with the following definition. A variable v is constant-value for a basic block B if and only if

1. v is defined in B.
2. v is not defined in any other basic block.
3. v is used only in basic blocks *dominated* by B.
4. v is not used before its definition in B.

We only analyze and possibly transform arrays which are stored in constant-value variables. Elements of the array may of course be modified many times, but we make sure that the variable containing the pointer to the array handle is a constant-value variable. Note that the fact that an array is stored in constant-value variable does not suffice to determine if an array transformation is legal. For example a shape of a 2-D array could be changed by changing the length of one of the rows, or a row could be passed as an argument or returned as a result. We conservatively assume that only arrays which have been allocated as rectangular arrays (using the construct corresponding to the `multianewarray` bytecode) and whose all uses are references using all dimensions can be remapped.

We can find constant-value arrays efficiently. For every array allocation, we note the variable, v, the array is stored in, and the basic block B that contains this allocation and we traverse all other basic blocks and make sure that

- v is not defined.
- v is used only in basic blocks which are dominated by B.

The above operation can be performed as a single scan of all basic blocks using the def-use information computed for every basic block in Section 3.3. During the same scan we verify that all uses of v access the array using all its dimensions.

Note that for block B we have to traverse all its instructions rather than just use the summary def-use information computed for B in Section 3.3.

3.5 Identifying Loop-defined Variables

To analyze precisely locality properties of array references contained in loops, we would have to determine which variables are loop induction variables and what is their step for the corresponding loops. Again, this operation is expensive and we have decided to approximate the notion of an induction variable with a *loop-defined variable*. For a loop L, a loop-defined variable is any variable which is defined in loop L. This simplification is justified by an observation that for scientific programs if a variable is assigned a new value in a loop then the value of that variable will be different in every loop iteration and the variable will usually be an induction variable with a unitary step.

Loop-defined variables can be identified a single scan of all basic blocks. For every basic block the set of variables defined in this block is added to the set of loop-defined variables for all (if any) enclosing loops. Enclosing loops can be accessed quickly because every basic block has a pointer to the innermost loop containing this block and every loop has a pointer an enclosing loop.

3.6 Recovering Multidimensional Array Structure

Array transformations can be only applied to multidimensional arrays. Multidimensional array references are present in the source program, but the translation to bytecodes lowers them into a sequence of one-dimensional references. Those can be converted back into multidimensional references with the techniques from our previous work [6].

The dimension recovery is more difficult in the JIT version of Briki. While in principle we could operate on bytecodes, for reasons explained in Section 3.1, we perform our optimizations on Kaffe IR which is even lower than bytecodes. In Kaffe IR, an array reference is converted to a sequence of low-level instructions like memory loads, register moves, shifts and additions. Briki analyzes each such sequence and represents it as a high-level multidimensional array reference. This high-level structure is not stored anywhere—it is recomputed every time it is needed. This approach results in a fast implementation. For cholesky, the time to ensure legality of a transformation, finding the optimal mapping (Section 3.7) and performing the transformation (Section 3.8) constitutes only about 20% of the time spent in Briki.

3.7 Finding Optimal Mappings

Since multidimensional arrays in the Kaffe VM are not represented as contiguous portions of memory, but rather as arrays of 1-D arrays, we use a much restricted set of array transformations as compared to the one from [3].

For every array reference we determine which subscript contains loop-defined variables of the innermost loop (or of the next enclosing loop, if the loop-defined variables of the innermost loop are not present in any subscript).

The best mapping for this array would make the dimension which corresponds to this subscript the right-most dimension to increase spatial locality.

In any real program it is very likely that there would be conflicts between desired mappings for different array references. We resolve the conflicts by assigning priority to every array reference. The priority is based on the loop nesting—the higher the nesting the higher priority. This policy is adopted since *usually* the statements with higher nesting are executed more times than statements with lower nesting. This is of course not guaranteed and in general it is not possible to predict at compile-time how many times a given statement will be executed.

In the case of a conflict we choose the array mapping preferred by the greatest number of references with the highest priority. For instance, if four is the highest priority and three references with priority four require dimension 0 in the right-most position and one reference with the same priority requires dimension 1 in the right-most position then dimension 0 is permuted into the right-most position.

3.8 Remapping Arrays

Once we know which dimension should be permuted to the rightmost position, remapping is very simple. In this process we use the dataflow information calculated in the step described in Section 3.3.

4 Experiments

For our array-based benchmarks, the JIT version of briki performs the same optimizations as its off-line predecessor. Furthermore for the our data sets the time to perform the optimizations just-in-time is several orders of magnitude shorter than the time to run the benchmark, even though we chose small (by scientific computing standards) problem sizes—none of our benchmarks took more than 15s to complete. Since the overhead for JIT optimizations is so small, the speedups are identical (within three significant digits) with the results published elsewhere [6] and range from 10% to 20%.

The speedups of Java benchmarks are not as good as the results of applying the same optimizations to Fortran versions of the same programs (those were as high as 50%). This can be explained by the lack of standard optimizations in the pre-release version of Kaffe used in our experiments. The quality of code generated by Kaffe will, undoubtfully, improve over time. The current version produces very poor code for array references. For example the i386 instruction sequence for a simple reference to a 2-D array: `a[i][j]` contains 30 instructions including two branches (for bounds checking) and ten memory references (for accesses to array handles and register spills and reloads). A high-performance Fortran compiler would translate an array reference like that (if `i` and `j` are loop variables) to one or two instructions with just one memory reference—the load of the array element. Our optimizations reduce the cost of this one load, by increasing the cache hit ratio for this instruction. In the code generated by Kaffe this gain is dampened by the cost of the extra nine memory references and 20 other instructions which are not improved by our optimizations.

To provide a better context for the discussion in Section 3, we present here times spent in various steps of our optimizer. We ran those experiments on a Linux computer with a 200 MHz PentiumPro processor. Some of the times measured by us were too short for the granularity of standard Unix timing routines. To obtain accurate results, we have used cycle counters defined by the Pentium architecture. The resolution of those counters was more than sufficient for our needs.

Figure 2 shows the break-down of the array transformation time for our cholesky benchmark. The time represented in the figure corresponds to less than half of the total JIT time (if we include the time of the Kaffe front- and back-ends). The absolute time for the array transformation optimization on a 200 MHz PentiumPro is 8.4ms.

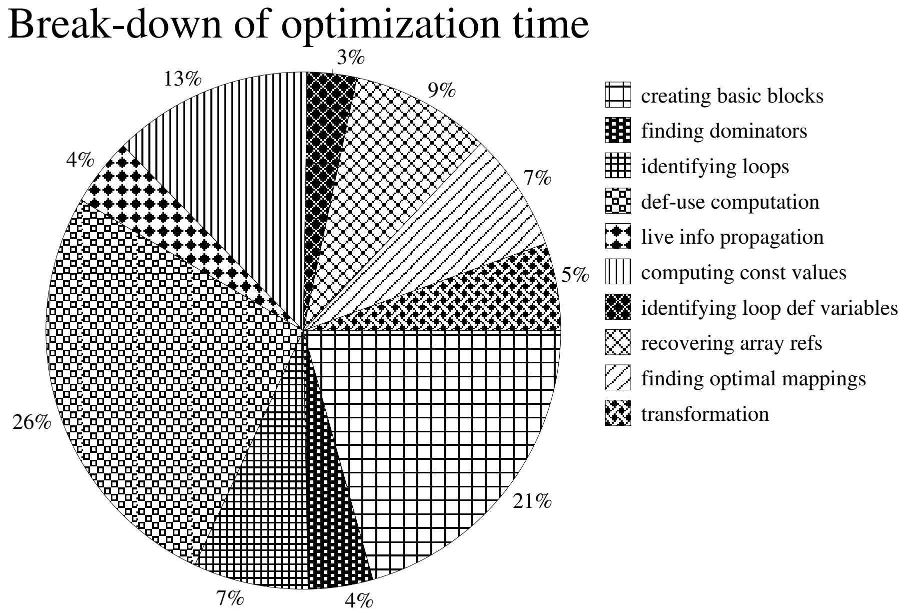

Figure 2: Break-down of the optimization time for cholesky

Figure 3 shows the overhead for performing array transformations for Java versions of cholesky and mxm kernels from the SPEC CPU92 Benchmarks [9]. Note that we have increased data sizes so that the execution times are meaningful. For scientific computing standards the problem sizes are still very small: on a 200 MHz PentiumPro the execution times are 4s for mxm and 12s for cholesky. However, the array sizes are large enough to benefit from better locality.

Even for those very small problem sizes, the time spent on our optimizations is completely negligible—three orders of magnitude shorter than the time spent on useful computation. Note that the time spent in Kaffe *doing the translation* is comparable to the time spent in Briki. The time attributed to Kaffe in Figure 3 is larger because we categorize the start-up time and the time to load all class files as "Kaffe overhead."

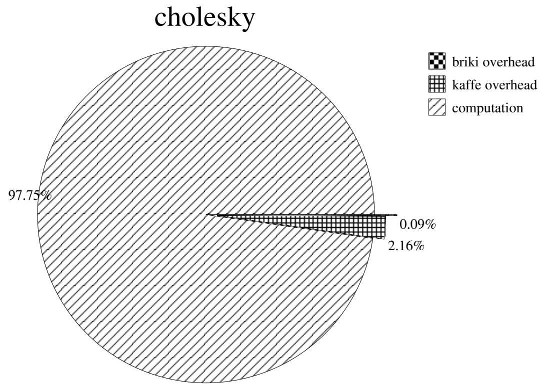

Figure 3: Briki vs. kaffe

5 Conclusion

We have implemented a very fast algorithm to decide and perform an optimizing transformation which can significantly speedup execution times of many scientific benchmarks. Currently the speedups are between 10% and 20%, but when Kaffe, the JIT compiler used by us, will perform standard optimizations which are absent in its current version, the speedups should be similar to the ones observed for Fortran programs—about 50%.

Our previous experience with an off-line Java optimizer has shown that some traditional algorithms used in compilers are too slow for a JIT compiler. We have proposed and implemented faster ways of performing analyses needed for our optimizations. For instance we have replaced the *reaching definition* with *constant values* and *loop induction variables* with *loop-defined variables*. As the result our JIT compiler is extremely fast, so that its running time is negligible even when the data sets used with our benchmarks result in execution times of a few seconds. The impact for the same benchmarks running on more realistic problem sizes would be even smaller.

Acknowledgements

This work was supported in part by an NSF Research Initiation Award (CCR-9409120) and ARPA contract F19628-94-C-0057. We would like to thank the developers of Kaffe [10] for providing an excellent tool for experimenting with Java JIT optimizations.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[2] C. Chambers, J. Dean, and D. Grove. Whole-Program Optimization of Object-Oriented Languages. TR UW-CSE-96-06-02, Department of Computer Science and Engineering, University of Washington, 1996.

[3] M. Cierniak and W. Li. Unifying Data and Control Transformations for Distributed Shared-Memory Machines. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, La Jolla, CA, June 1995.

[4] M. Cierniak and W. Li. Validity of Interprocedural Data Remapping. TR 642, Computer Science Department, University of Rochester, November 1996.

[5] M. Cierniak and W. Li. Briki: an Optimizing Java Compiler. In *Proceedings of the IEEE COMPCON '96*, San Jose, CA, February 1997.

[6] M. Cierniak and W. Li. Optimizing Java Bytecodes. To appear in *Concurrency: Practice and Experience*, June 1997.

[7] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley Longman, Inc., 1996.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Longman, Inc., 1996.

[9] Standard Performance Evaluation Corporation. SPEC CPU92 Benchmarks. April 1996. Available at `http://open.specbench.org/osg/cpu92/`.

[10] T. Wilkinson. Kaffe v0.8.3 — A Free Virtual Machine to Run Java Code. March 1997. Available at `http://www.kaffe.org/`.

[11] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.