Effective Enhancement of Loop Versioning in Java

Vitaly V. Mikheev, Stanislav A. Fedoseev, Vladimir V. Sukharev, and Nikita V. Lipsky

A. P. Ershov Institute of Informatics Systems, Excelsior, LLC
Novosibirsk, Russia
{vmikheev,sfedoseev,vsukharev,nlipsky}@excelsior-usa.com

Abstract. Run-time exception checking is required by the Java Language Specification (JLS). Though providing higher software reliability, that mechanism negatively affects performance of Java programs, especially those computationally intensive. This paper pursues loop versioning, a simple program transformation which often helps to avoid the checking overhead. Basing upon the Java Memory Model precisely defined in JLS, the work proposes a set of sufficient conditions for applicability of loop versioning. Scalable intra- and interprocedural analyses that efficiently check fulfilment of the conditions are also described. Implemented in Excelsior JET, an ahead-of-time compiler for Java, the developed technique results in significant performance improvements on some computational benchmarks.

Keywords: Java, performance, loop optimizations, ahead-of-time compilation

1 Introduction

To date, Java has become an industry-standard programming language. As Java bytecode [3] is a (portable) form of intermediate representation, a great wealth of dynamic and static optimization techniques was proposed to improve the originally poor performance of Java applications. One of the reasons for the insufficient performance is obligatory run-time checks for array elements access operations. JLS [2] requires two checks[1] for read/write of each array element `a[i]`: first, `a` has not to be `null` (otherwise `NullPointerException` should be thrown), then `i` must be in the range `0<=i<a.length` (if not, `IndexOutOfBoundException` should be thrown). It is easy to see that the major execution overhead occurs in loop-carried array operations. A Java implementation could yield to temptation to provide a mode in which the run-time checks are disabled but that would effectively violate JLS terms.

Traditional static analyses may ameliorate the situation to some extent. For instance, no checks are required if `a` is known to be a non-null value and `i` has

[1] Writes to an array of a reference type may also require type inclusion check

© Springer-Verlag Berlin Heidelberg 2002

proved to be in the proper range. Intraprocedural static analyses are able to infer such program properties quite effectively, of course, if array objects are created and then used within the same Java method ([12], [13]). Unfortunately, the optimization of real Java programs often requires *global* analyses. It is enough to note that array references are typically stored in *shared memory variables* (static or instance fields). However, Java dynamic facilities such as Reflection API, JNI, dynamic class loading inhibit applicability of the analyses, not to mention their high spatial and time complexity.

An original technique called *loop versioning* was proposed to optimize loops without global flow analysis ([8]). The key idea is to keep a part of checks in the resulting code but move them out of the loop body as illustrated in Figures 1, 2.

```
for(i=0; i<=ub; i++)
  chk_null(A)[chk_idx(A,i)] = 2*chk_null(B)[chk_idx(B,i+1)];
```

Fig. 1. Original loop code

```
if ((A!=null) && (B!=null) && (ub<A.length) && (ub+1<B.length))
//version with no checks
  for(i=0; i<=ub; i++)
    A[i] = 2*B[i+1];
else
// original version
  for(i=0; i<=ub; i++)
    chk_null(A)[chk_idx(A,i)] = 2*chk_null(B)[chk_idx(B,i+1)];
```

Fig. 2. Versioned loop code

In such case, two copies (or versions) of a loop have to be generated. One copy is checks-free provided all required conditions are tested before loop. The other is the original loop with checks. The technique has a great advantage over static analysis: the program properties which are extremely hard to analyze statically, now may be just checked at run-time before loop execution. However, care must be taken to transform the original program correctly, as array reference variables, index expressions and the final value of the inductive variable have to be loop invariants. Thus, any loop versioning implementation should advocate (easy provable) conditions for correctness of the optimization.

We propose a simple and effective algorithm of loop versioning that can be used in production Java compilers. We implemented it in Excelsior JET [23], an ahead-of-time Java bytecode to native code compiler. The rest of the paper is organized as follows. Section 2 highlights certain aspects of the Java Memory Model with respect to applicability of loop versioning. Sections 3, 4 describe program analysis and transformation required for the optimization. Section 5 outlines our implementation of loop versioning in the Excelsior JET optimizing compiler. The obtained results are presented in Section 6. Section 7 pursues related works and, finally, Section 8 concludes.

2 Java Memory Model

Let us consider an example in Figure 3. The question is under which circumstances `A`, `B` and `UB` are loop invariants? If they are not, loop versioning may not be a correct transformation for such loops. This section helps answer the question. Note that we discuss a general case when the expressions may include not only locals but also static or instance fields.

```
for(i=0; i<UB; i++) {
A[i] = B[i+1];
<operator 1>;
...
<operator k>;
}
```

Fig. 3. If `A`, `B` and `UB` are loop invariants?

The "Threads and Locks" chapter of JLS rigorously defines the Java Memory Model for (generally) multi-threaded programs with the help of three abstract machines: *main memory*, *thread working memory* and *thread execution engine* as depicted in Figure 4. The main memory keeps track of shared variables status performing the *read/write* actions. Each thread has working memory, its own "local view" of the main memory. A thread working memory holds *working copies* of shared variables and communicates with the main memory through *load/store* message streams specific for each shared variable. Thread execution engines carry out Java code according to the language semantics and exchange data with working memory through *use/assign* message streams.

The main concern of the specification is that reading and writing shared variables are *non-atomic* w.r.t. thread switching. For instance, the entire read-load-use action chain is not guaranteed to be executed in one time slice of a thread though each of the actions is atomic by definition. The main rules related to shared variables are[2]:

1. A thread execution engine is free to use a working copy of a particular shared variable provided that copy was loaded from the main memory at least once.
2. If a thread updates a working copy through an assign action, subsequent use actions that occur in the thread, should return the most recently assigned value.

[2] The memory model has a more rich set of restrictions. For our purposes, we shortly describe only those useful for loop invariant computation.

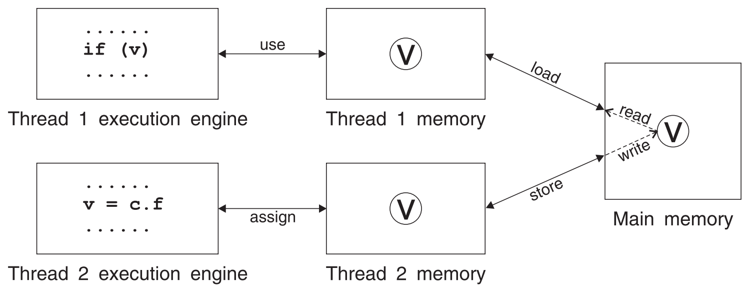

Fig. 4. The Java Memory Model

In fact, the specification imposes a strict order on use and assign actions only whereas other actions updating the main and working memory may be issued at any time, at the whim of implementation[3]. However, there are two exceptional cases in which working copies have to be *in sync* with the main memory:

(i) A shared variable has the `volatile` modifier. In such case, a working copy should be synchronized with the main memory each time a use or assign action occurs in a thread.

(ii) A `synchronized` block (or call to a `synchronized` method) is present on execution path. If so, *all* working copies should be synchronized with the main memory at enter and exit of the block.

Proposition 1 (”Localization” of shared variables). *Let a loop-carried statement include an expression with shared variables. If neither of the above conditions holds, the variables may be read before the loop and then treated as locals.*

Thus, if the expressions to be proved loop invariants contain shared variables, our algorithm analyzes loop body to check the conditions (i), (ii). If they are not fulfilled, the analysis concludes that the involved shared variables *may* be invariants, otherwise it makes a conservative assumption they are not. Loop-carried calls are discussed in the next Section.

3 Program Analysis

This section gives a set of sufficient conditions for applicability of loop versioning and describes several analyses to effectively check them.

[3] Obviously, CPU architectures with a lot of registers especially benefit from the memory model. Although on Intel x86, shared variables are unlikely to be allocated on registers for a long time, an implementation may provide better cache behaviour if working copies are assigned to local temporaries

3.1 Alias Analysis

If expressions to be proved loop invariants contain instance fields or array elements, the analysis has to infer that they are immutable in the loop body. The problem is that instance fields and array elements may be *aliased* in Java so that writing one variable changes the value of the other. For example, two expressions `o1.f` and `o2.f` are aliases, if both `o1` and `o2` refer to the same object. In general, the property is practically undiscoverable at compile-time even with (computationally hard) global flow analysis. Instead, our algorithm detects which expressions *may not* be aliases employing the following simple criteria[4]:

1. Instance fields with different names may not be aliases (e.g. expressions `expr1.f` and `expr2.g` may not be aliases).

2. Let us consider two expressions `o1.f` and `o2.f`. Let `C1` and `C2` be static (declared) classes for objects `o1` and `o2`. Furthermore, `SuperC1`(`SuperC2`) is the superclass of `C1`(`C2`) in which instance field `f` was declared. Expressions `o1.f` and `o2.f` may not be aliases if `SuperC1` and `SuperC2` are different classes.

3. Let `T1[]` be a static (declared) type for array object `a1` and `T2[]` be a static type for array object `a2`. Expressions `a1[n]` and `a2[m]` may not be aliases if either
   – at least one of types `T1`, `T2` is primitive and they are different types
   – types `T1`, `T2` are not interfaces[5] and neither can be cast to the other

In other cases, the analysis conservatively concludes that the expressions may be aliases. Of course, the technique gives us correct but (generally) non-precise results. Nevertheless, we prefer to use it for effective computability. For instance, the criteria for arrays do not work if a loop computes arrays of the `int[]` type and an invariant expression contains an access to another (immutable) integer array `a[expr]`. However, it works fine if the loop computes `float[]` or `double[]` arrays only.

3.2 Loop Invariants Computation

For the sake of simplicity, the augment of inductive variable is required to be a constant and, thus, is invariant. However, the following entities have to be proved loop invariants for correct application of loop versioning:

– the expression that denotes the final value of the inductive variable
– array references which are subjects for check removal
– index expressions provided the inductive variable is fixed[6]

[4] The proposed criteria benefit from the strict type system and absence of address arithmetic in Java.

[5] If at least one type is an interface, there may exist a class that implements it. In such case, the non-aliasing property can not be determined without a global type analysis.

[6] To be more precise, if each occurence of the inductive variable in an index expression is replaced with a constant, the expression would become loop invariant.

First, our algorithm performs "localization" of shared variables as proposed in Section 2. Then it proves that shared variables appearing in the left side of assignments may not be aliases of those which are part of invariant expressions being analyzed. Finally, a traditional local flow-sensitive analysis [1] is employed to check whether the expressions are invariants. If an either test fails, versioning is not applied to the loop.

3.3 Checking Boundaries of Index Expressions

As long as index expressions (with a fixed inductive variable) are loop invariants, they may be thought of as a set of functions $\{f_k(i) : [a..b] \to int\}_k$, where a and b are initial and final values of inductive variable i. For practical consideration, the algorithm recognizes only linear functions in the form $f(i) = k * i + l$ which give minimum and maximum at the margins of the domain range. Thus, for each array access `arr[k*i+l]`, the compiler should emit checking this pre-condition formula for positive augment of the inductive variable[7].

```
(k > 0) ? 0 <= k*a+l && k*b+l <arr.length :
          0 <= k*b+l && k*a+l <arr.length
```

Thus, the compiler has to generate a concatenation of similar formulas for each index expression within loop. Note that as a rule, the resulting formula will be essentially reduced during further local constant propagation and range analyses that our compiler performs.

3.4 Handling Loop-Carried Calls

In general, a loop body may include calls to other methods among operators in Fig. 3. Though called methods cannot access locals of the caller, they may modify shared variables, contain `synchronized` blocks or invoke yet other methods which do that. In order to get more precise results, our algorithm makes a simple interprocedural analysis to check operators from the called methods[8]. One might note that the same effect may be achieved through simply inlining such methods. However, care must be taken to prevent excessive inlining. Not to mention the growth of the code size, it may result in decreasing performance. Let us imagine a loop containing a call to quite a large method on a rarely executed branch. If the call is inlined, it may consume extra CPU registers and worsen instruction cache behaviour as noted in the work [14]. Because our compiler framework is able to perform adaptive profile-based optimizations (including inlining), we prefer to implement scalable interprocedural analyses (if possible), and not to rely on increasing inline aggressiveness.

[7] As mentioned above, the sign of the inductive variable augment is known at compile-time. We give the formula for positive augments only as it is symmetric for negative ones.

[8] Of course, virtual method invocation hinders the analysis. Our compiler accomplishes local type propagation which often helps to "devirtualize" such methods. If that is not possible, the analysis treats such methods as potentially unsafe and declines versioning if invariant expressions contain shared variables.

3.5 Complexity

The described algorithms scale linearly in the size of the program. The flow-insensitive analysis of loop-carried operators and the simple alias analysis give the complexity proportional to N (program size) + G (non-virtual call graph size). Thus, our algorithm runs in $O(N+G)$ both time and space.

Strictly speaking, our compiler performs a flow-sensitive intraprocedural analysis that runs in $O(n^2)$, where n is the number of local temporaries. As it takes effect during compilation of each method anyway, it does not matter whether loop versioning is applied. This is why we give "pure" complexity of the versioning analysis not taking into account other local optimizations.

4 Program Transformation

If the described analyses have succeeded, the compiler can safely perform loop versioning. The necessary program transformation is very simple and includes the following steps:

1. Generation of pre-conditions for nullness of array references
2. Generation of pre-conditions for index bounds
3. Replication of loop body with removal of checks

The only important note is that index bound checks *must* follow nullness checks in the pre-condition formula concatenated. As the index check conditions dereference array variables (in the form `expr<A.length`), `NullPointerException` may occur before entering the loop if the formula is constructed in the reverse order. As can be seen, that would change the Java semantics (namely, the precise exception model [2]) because loop operators may have side-effects.

5 Implementation

This section highlights our implementation of loop versioning and particularly focuses on the benefits of ahead-of-time (static) compilation.

5.1 Excelsior JET

We implemented the described algorithm in Excelsior JET, a static compiler which converts Java bytecode to native (platform specific) code before execution. JET is based on the Excelsior's compiler construction framework which architecture is shown in Figure 5. Organization of the framework is similar to those of other known compilers, e.g. Marmot [9], HPJC [10].
The main advantage of static compilation is that it is performed only once, on a developer's machine and typically, the majority of classes is known at

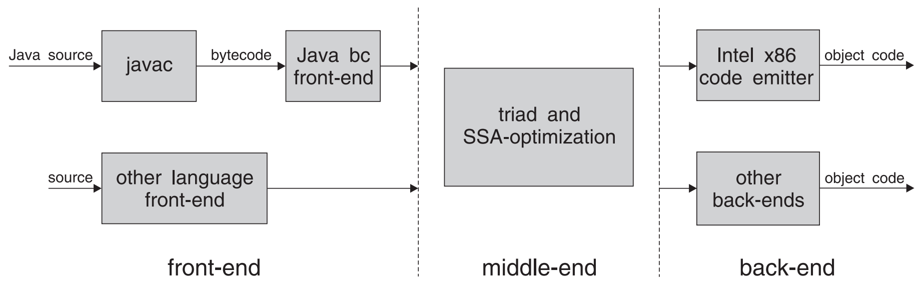

Fig. 5. The Excelsior's compiler construction framework

compile-time[9]. Thus, the compiler is free to employ any time- and memory-expensive optimization technique, resulting in much better code quality than in the case of dynamic (just-in-time) compilation. In that sense, the JET abbreviation stands for Just-Enough-Time compiler. The Java "Write Once, Run Anywhere"^{TM} paradigm is supported by providing static Java compilers for all major platforms, just like it is supported right now by providing a JVM for each of them. Currently, JET is targeting the Wintel platform, however, porting to other platforms is under consideration.

5.2 Implementation Notes

Now we describe certain aspects of versioning implementation in Excelsior JET.

Decompilation of loop operators. The Java bytecode which JET takes as input language, contains stack-based VM instructions [3]. JET bytecode front-end employs quite complex algorithms of abstract interpretation and symbolic computations to reconstruct (or decompile) structural operators. The reconstruction algorithms recognize loops with inductive variables and, thus, are not limited to `for` operators only. For instance, the loops might be written with the use of `while` or `do-while` operators in the original Java sources. In essence, the exploited algorithms are similar to those of related works [18], [19]. However, in order to make them work properly on a variety of real-world Java applications, we had to carefully adapt the algorithms to the Java bytecode specification.

Powerful local optimizations. As in most advanced compilers, JET middle-end has 3-address value internal representation and features local SSA-based optimizations [1]. If possible, checks are removed during local constant propagation and range analysis. However, along with CSE (Common Subexpressions

[9] If a class is unknown beforehand, JET provides *caching* dynamic compilation through Mixed Compilation Model [23] sacrificing several inreprocedural optimizations in favour of lower resource consumption at run-time

Elimination), the optimizations are extremely useful for the reduction of precondition formulas generated during the loop versioning transformation.

Generation of run-time checks. JET, as well as other compilers, co-operates with the run-time system to handle `NullPointerException`. In fact, dereference of zero or another (small) value is treated as the exception. Intel x86 assembly code for checking instructions is presented in Table 1.

Table 1. Excelsior JET check instructions

Null check	Combined null/index check
`// eax holds address of array`	`// eax holds address of array`
`cmp eax, [eax]`	`// ebx holds index value`
	`cmp [eax+arrLenOffset], ebx`
	`jbe IndexOutOfBound`

Note that it would make little sense to remove null checks while preserving index ones as both are performed at once by a single CPU instruction.

Adaptive optimizations. Our compiler framework supports profile-based optimizations. The collected profile often recommends not to inline particular methods as they are rarely executed. This is the reason that caused us to implement interprocedural analysis of loop-carried calls.

Interprocedural analysis. In order to allow JET to perform interprocedural optimizations (e.g. escape analysis [7], inlining etc.), we have implemented *syntax tree object persistency* permitting arbitrary tree object graphs to be saved to/restored from file or cached into memory, if they are intensively used. This technique resembles the slim binaries approach proposed in [16] as an alternative to the Java bytecode during dynamic compilation. However, we restrict its use to static code analysis and optimization only. The mechanism was just recycled for analyzing loop-carried calls. In it, we benefit from the ahead-of-time compilation approach because most dynamic compilers cannot afford even simple interprocedural analyses due to time and memory limitations.

6 Experimental Results

This section gives the results we obtained on two series of benchmarks. One series is provided to discover the "pure" effect of versioning when only array access operations are executed in loops. The other series is well-known standard benchmark suites - JavaGrande/EPCC Sequential 2.0 [20] and SciMark 2.0 [21]. All tests were run on the same system: AMD Athlon^{TM} running at

1400MHz/768MB RAM/Windows 2000 Professional. In order to see the best results that versioning may potentially give and the actual performance impact of the optimization, we provide the results for the following execution modes:

1. checks enabled, versioning disabled
2. both checks and versioning enabled
3. all checks disabled

6.1 Pure Effect of Versioning

Sum1, Sum5 and Sum10 are simple benchmarks which sum the contents of one, five and ten `int[200000]` arrays in the innermost loop during a number of iterations. Table 2 shows the execution time in seconds.

Table 2. Summing elements of large arrays

Benchmark	**+checks -vers**	**+checks +vers**	**-checks**
Sum1	1.07s	0.64s (-39%)	0.64s (-39%)
Sum5	2.89s	2.45s (-15%)	2.43s (-15%)
Sum10	7.55s	6.38s (-15%)	6.38s (-15%)

The best result (39% reduction of execution time) is achieved on the simplest benchmark. Smaller improvement of the others is caused by the CPU data cache behaviour as the tests read elements of several very large arrays in the same loop. It is not surprisingly that benchmarks with enabled versioning are almost as fast as checks-free ones.

6.2 Standard Benchmarks

For our purposes, we selected only those benchmarks which perform array access operations in loops. Effectiveness of our versioning implementation is given in Table 3. For each test, it shows the total number of loops [10] and the number of those to which the optimization was applied.
The benchmarks demonstrate 100% effectiveness, excepting JGFSeq3/Euler which operates on `int[][]` arrays. We did not implement loop versioning for multidimensional arrays intentionally. The matter is that flattening multidimensional arrays [17] is a *different* optimization complementary to loop versioning. If a (rectangular) multidimentional array is flattened, the index expression `a[i][j]` is transformed to `a[i*firstDimLength+j]` which meets our versioning criteria. We plan to implement support for rectangular arrays in future versions.

[10] Note that the total numbers count only the loops with array access operations which are subject for versioning.

Table 3. Effectiveness of loop versioning analysis

Benchmark	**Num. of loops**	**Versioned loops**
SciMark2/FFT	3	3 (100%)
SciMark2/Sparse matmult	2	2 (100%)
SciMark2/SOR	2	2 (100%)
SciMark2/LU	14	14 (100%)
JGFSeq3/Search	5	5 (100%)
JGFSeq3/MonteCarlo	2	2 (100%)
JGFSeq3/RayTracer	2	2 (100%)
JGFSeq3/MolDyn	1	1 (100%)
JGFSeq3/Euler	8	4 (**50%**)

Table 4 gives the results of performance improvement due to loop versioning application. Columns 2-4 hold the numbers of operations per second specific for each benchmark (greater number means better result).

Table 4. Performance improvement

Benchmark	**+checks -vers**	**+checks +vers**	**-checks**
SciMark2/FFT	194.4	196.1 (+0.8%)	196.8 (+1.2%)
SciMark2/	170.8	212.6 **(+24.4%)**	246.9 **(+44.5%)**
Sparse matmult	170.8	212.6 **(+24.4%)**	246.9 **(+44.5%)**
SciMark2/SOR	331.8	331.7 (0%)	333.9 (+0.6%)
SciMark2/LU	266.2	312.7 **(+17.4%)**	323.0 (+21.3%)
JGFSeq3/Search	712513.2	713972.4 (0%)	755216.94 (+1.0%)
JGFSeq3/MonteCarlo	1733.73	1733.4 (0%)	1757.77 (+1.3%)
JGFSeq3/RayTracer	2706.92	2723.31 (+0.6%)	2729.92 (+0.8%)
JGFSeq3/MolDyn	180562.11	247306.3 **(+36.9%)**	251991.67 (+39.5%)
JGFSeq3/Euler	5.14	5.14 (0%)	5.71 **(+11.0%)**

As can be seen, versioning gives performance improvement on the same tests as check disabling does. The performance gap between columns 3 and 4 on JGF-Seq3/Euler is due to not having yet implemented compiler support for rectangular multidimensional arrays. However, the effect on SciMark2/Sparse matmult is a more subtle substance. The benchmark has three nested loops and versioning two innermost ones disrupts (otherwise well-behaved) instruction cache because of swollen code. We intend to ameliorate that as follows. Given the assumption that the checked version is rarely executed, the compiler can move it to the end of the method's code section and then link it with the main code through forward and backward jump instructions [11].

One might be interested to see the results of performance comparison between Excelsior JET, the most current Java VMs (e.g. Sun's HotSpot Server VM which performs powerful SSA-based optimizations [4]) and other generally available ahead-of-time compilers for Java. Although independent studies (e.g. [22]) show that our compiler outperforms them on many benchmarks including those cited in this paper, we do not give the results on purpose. That would require us to consider the entire variety of optimizations implemented in JET and other compilers, not only loop versioning we pursue in this paper.

7 Related Works

Byler et al. [8] seemingly pioneered the loop versioning optimization. Their work aimed at dealing with the lack of compile-time information when optimizing computational programs for parallel architectures. Versioning was proposed to detect alias-safe array regions thereby allowing concurrent computations. Pugh [15] gives an excellent description of the Java Memory Model in details and proposes further improvements to make more optimizations applicable to Java. The works [12], [13] pursue various static analyses for array checks removal. The proposed techniques are able to eliminate checks with the use of local (intraprocedural) optimizations. However, real-world Java programs are unlikely to create and use arrays within the same local scope of a method. Global flow-sensitive analyses are computationally hard. In general, any global analysis may not be used for Java due to presence of dynamic loading of classes and metaprogramming facilities. Artigas et al. [11] consider loop versioning for Java implemented in IBM High Performance Compiler [10]. Though the work mentions the possibility of loop versioning in Java, conditions of its applicability are not discussed. The main concern of their work is the use of versioning for parallel processing alias-safe array regions as in [8]. We find the optimization very useful even for single-processor architectures [12] and plan to implement it in future releases of Excelsior JET. Fitzgerald et al. at Microsoft Research, the authors of the Marmot ahead-of-time compiler for Java [9] do not regard the loop versioning optimization. The work [5] describing the architecture of the IBM Just-In-Time compiler for Java, most directly relates to ours. The authors propose an algorithm which relies on loop invariants, however, they do not describe invariant computation and alias analysis (if the last was employed). Because their work is an overview of the entire compiler architecture, it is hard to compare effectiveness of invariant computation which is not described in details. Moreover, their algorithm limits recognizable index expressions to $i + constant$ whereas our algorithms permits expressions $k * i + l$, where k and l are loop invariants not necessary constants.

[11] Some people from the compiler community wittily call the technique ”siberian code sections”.

[12] The guarantee of alias-free arrays contributes to more effective code generation as many redundant load instructions may be eliminated.

The form of index expression (potentially) allows us to use versioning for rectangular multidimensional arrays as well. Finally, the algorithm employed by IBM JIT does not perform versioning if the loop includes calls. Our implementation makes simple interprocedural analysis to handle loop-carried calls.

8 Conclusion

This paper presented loop versioning, a technique for removal of null and index checks in Java programs working with arrays. A set of sufficient conditions for correct application of loop versioning in Java was given. In order to effectively check the conditions, this work proposed algorithms of alias analysis and loop invariant computation which scale linearly in the size of the program. Implemented in Excelsior JET, an ahead-of-time compiler for Java, the developed technique results in significant performance improvements on computational benchmarks. The interesting area for future works is to provide the current implementation with support for alias-free array regions and rectangular multidimensional arrays.

Acknowledgements

Without the ongoing support of the entire Excelsior Java team, this work would not have been possible. A special thank-you to John O. Osbourne for his support.

References

1. Steven S. Muchnik. *Advanced Compiler Design And Implementation.* Morgan Kaufmann Publishers, 1997. 298, 300
2. J. Gosling, B. Joy and G.Steele. *The Java(tm) Language Specification, Second Edition.* Addison-Wesley, Reading, 2000. 293, 299
3. T. Lindholm, F. Yellin, B. Joy, K. Walrath. *The Java Virtual Machine Specification.* Addison-Wesley, 1996. 293, 300
4. The Java HotSpot(tm) Virtual Machine, *Technical Whitepaper*, Sun Microsystems Inc., 2001. 304
   URL: `http://www.sun.com/solaris/java/wp-hotspot`
5. Suganuma et al. Overview of the IBM Java Just-In-time Compiler, *IBM Systems Journal*, Vol. 39, No. 1, 2000. 304
6. V. Mikheev. Design of Multilingual Retargetable Compilers: Experience of the XDS Framework Evolution. In *Proc. of Joint Modular Languages Conference, JMLC'2000*, Volume 1897 of LNCS, Springer-Verlag, 2000.
7. V. Mikheev, S. Fedoseev. Compiler-Cooperative Memory Management in Java. To appear in *Proc. of 4th International Conference Perspectives of System Informatics, PSI'2001*, LNCS, Springer-Verlag, 2001. 301
8. M. Byler et al. Multiple version loops. In *Proc. of the 1987 International Conference on Parellel Processing*, 1987. 294, 304
9. R. Fitzgerald, T. Knoblock, E.Ruf, B. Steensgaard, D. Tarditi. Marmot: an Optimizing Compiler for Java, Microsoft Research, MSF-TR-99-33, 1999. 299, 304

10. V. Seshadri. IBM high performance compiler for Java. AIXpert Magazine, September 1997. 299, 304
11. P. Artigas, M. Gupta, S. Midkiff and J. Moreira. Automatic Loop Transformations and Parallelization for Java, In Proc. *International Conference on Supercomputing, ICS'00*, 2000. 304
12. D. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceeding of PLDI'00*, 2000. 294, 304
13. P. Pomminvillen et al. A Framework for Optimizing Java Attributes. In Proc. *Compiler Construction, CC'2001*, Volume 2027 of LNCS, Springer-Verlag, 2001. 294, 304
14. M. Arnold, S. Fink, V. Sarkar, and P. Sweeney. A Comparative Study of Static and Profile-Based Heuristics for Inlining. In *Proc. of ACM SIGPLAN 2000 Workshop on Dynamic and Adaptive Compilation and Optimization, DYNAMO'00*, 2000. 298
15. W. Pugh. Fixing the Java Memory Model. In *ACM 1999 Java Grande Conference*, San Francisco, CA, June 1999. 304
16. M. Franz, Th. Kistler. Slim binaries. Technical report 96-24, Department of Information and Computer Science, UC Irvine, 1996. 301
17. J. Moreira, S. Midkiff, M. Gupta. A comparison of three approaches to language, compiler, and library support for multidimensional arrays in Java. In *Proc. of ISCOPE Conference on ACM 2001 Java Grande*, 2001. 302
18. C. Cifuentes. Structuring Decompiled Graphs. In *Proc. of the International Conference on Compiler Construction, CC'96*. Volume 1060 of LNCS, Springer-Verlag, 1996. 300
19. U. Lichtblau. Decompilation of control structures by means of graph transformations. In *Proc. of the International Joint Conference on Theory and Practice of Software Development, TAPSOFT'85*. Volume 185 of LNCS, Springer-Verlag, 1985. 300
20. The Java Grande Forum Sequential Benchmarks, Version 2.0. 301
URL: `http://www.epcc.ed.ac.uk/javagrande/sequential.html`
21. SciMark 2.0. Java benchmark for scientific and numerical computing. 301
URL: `http://math.nist.gov/scimark2/`
22. O. P. Doederlein. The Java Performance Report - Part IV: Static Compilers, and More. JavaLobby, August, 2001 URL:
`http://www.javalobby.org/fr/html/frm/javalobby/features/jpr/part4.html`
304
23. Excelsior JET. *Technical Whitepaper*, Excelsior LLC, 2001. 294, 300
URL: `http://www.excelsior-usa.com/jetwp.html`