RC 21166 (94594) 04/28/98
Computer Science/Mathematics

IBM Research Report

From Flop to MegaFlops: Java for Technical Computing

José E. Moreira, Samuel P. Midkiff, Manish Gupta

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

This page intentionally left blank.

From Flop to MegaFlops: Java for Technical Computing

J. E. Moreira S. P. Midkiff M. Gupta
{*moreira,midkiff,mgupta*}@*watson.ibm.com*

IBM T. J. Watson Research Center
P. O. Box 218
Yorktown Heights, NY 10598-0218

Although there has been some experimentation with Java as a language for numerically intensive computing, there is a perception by many that the language is unsuited for such work. In this paper we show how optimizing array bounds checks and **null** pointer checks creates loop nests on which aggressive optimizations can be used. Applying these optimizations by hand to a simple matrix-multiply test case leads to Java compliant programs whose performance is in excess of 500 Mflops on an RS/6000 SP 332MHz SMP node. We also report in this paper the effect that each optimization has on performance. Since all of these optimizations can be automated, we conclude that Java will soon be a serious contender for numerically intensive computing.

1 Introduction

The scientific programming community has recently demonstrated a great deal of interest in the use of Java for technical computing. There are many compelling reasons for such use of Java: a large supply of programmers, it is object-oriented without excessive complications (in contrast to C++), and has support for networking and graphics. Technical computing is moving more and more towards a network-centric model of computation. In this context, it can be expected that Java will first be used where it is most natural: for visualization and networking components. Eventually, Java will spread into the core computational components of technical applications.

Nevertheless, a major obstacle remains to the pervasive use of Java in technical computing: performance. Let us start by looking into the performance of a simple matrix-multiply routine in Java, as shown in Fig. 1. This routine computes $C = C + A \times B$, where C is an $m \times p$ matrix, A is an $m \times n$ matrix, and B is an $n \times p$ matrix. We use that routine to multiply two 500×500 matrices ($m = n = p = 500$) on an RS/6000 SP 332MHz SMP node. This machine contains 4×332 MHz PowerPC 604e processors, each with a peak performance of 664 Mflops. We refer to this simple benchmark as MATMUL.

The Java code is compiled into a native executable by the IBM High Performance Compiler for Java (HPCJ) [11], and achieves a performance of 5 Mflops on a 332 MHz PowerPC 604e processor. The equivalent Fortran code, compiled by the IBM XLF compiler, achieves 265 Mflops! A 50-fold performance degradation makes it hard to justify using using Java for a technical application. The single-threaded version of ESSL, carefully hand-tuned to achieve optimum performance, executes the matrix multiplication at 289 Mflops. The multi-threaded version of ESSL achieves 1183 Mflops!

```
static void matmul(double[][] A, double[][] B, double[][] C,
                   int m, int n, int p) {

    int i, j, k;
    for (i=0; i<m; i++)
      for (j=0; j<p; j++)
        for (k=0; k<n; k++)
          C[i][j] += A[i][k]*B[k][j];
}
```

Fig. 1. Simple matrix-multiply code in Java.

Why is the performance of Java so bad, relative to Fortran? In the case of matrix-multiply, the IBM XLF Fortran compiler uses several high-order transformations to achieve high-performance, including: blocking (for better cache reuse), loop unrolling, and scalar replacement. Because of Java's strict sequential semantics, combined with the generation of exceptions when an array index is out-of-bounds, these same transformations are not legal in Java.

The goal of this paper is to show that we can close the performance gap between Java and Fortran or C++. The key to achieving high-performance in Java is a new transformation that creates regions of an iteration space in which all array indices are guaranteed to be in-bounds. This has both direct and indirect benefits. First, the actual checks for array bounds can be removed in those regions. More important, though, because all iterations in these *safe* regions will execute without exceptions, many of the high-order transformations applicable to Fortran and C++ (including parallelization) can be used. In the particular case of MATMUL on a RS/6000 SP 332MHz SMP node, we have achieved fully-compliant Java performance in the excess of 500 Mflops.

The rest of this paper is organized as follows. Section 2 discusses one of our methods to create safe regions, without out-of-bounds array indices. Section 3 then discusses how to apply high-order transformations to these regions and lists performance improvements obtained with Java. Section 4 reviews some extensions that have been proposed to Java to better support technical computing, and discusses how they relate to our work. Section 5 shows how we can parallelize MATMUL and presents more performance results. In Section 6 we discuss some related work. In particular, we show that speculative execution is an alternative strategy to enable many of the high-order transformations. Finally, in Section 7 we conclude this paper and discuss some future work.

2 Optimization of array bounds checking

A major impediment to optimizing Java is the requirement that exceptions be thrown precisely. An exception must be thrown for any attempt to dereference a **null** pointer or perform an out-of-bounds array access [18]. Because most references in computationally intense Java loops are to arrays or via pointers, most references have the potential to generate an exception. Because of the precise exception requirement, it is impossible to do code motion across potential exceptions, and in general it is impossible to perform many optimizations.

Our solution to this problem is to transform the program so that most computation in programs that perform few, or no, invalid accesses is in loops that are provably free of invalid accesses. Intuitively, these loops are formed by tiling the iteration space of the loop nest with *regions*. A region is a set of adjacent iterations in the loop nest such that either (i) all accesses in the region are provably safe, *i.e.* will not cause an exception to be thrown, or (ii) one or more accesses may be invalid. An iteration i is placed into a type (i) region (also called a *safe* region) if no reference in iteration i causes an invalid access exception. Otherwise the iteration is placed into a type (ii) region (also called an *unsafe* region).

To execute the two types of regions, two versions of the loop nest are formed. One version, which executes the iterations in safe regions, has no code to check for access violations. The second version, which executes the iterations in the unsafe regions, performs all access violation checks explicitly mandated by the Java VM specification. By executing the regions and the iterations within the regions in proper order, the iteration space of the loop is executed in the original execution order. However, checks are only performed in unsafe regions.

The rest of this section will explain how iterations that perform safe accesses for a reference are determined, how this information is used to tile the iteration space with safe and unsafe regions, and how code is generated to execute the iterations in the regions using the proper loop version. For more details on this and related optimizations, see [25].

2.1 Determining iterations with safe accesses

Let $A[f(i)]$ be the reference for which the safe bounds are being found. Let i be the loop index variable of a normalized loop with stride one. We refer to this loop as loop i. Let $f(i)$ be a linear function of the form $c \cdot i + b$, where c and b are invariant within the loop nest, and $c > 0$. See [25] for details on how to handle more general subscripts. In the case of arrays of arrays, used in Java to simulate multi-dimensional arrays, we treat the access to each dimension of the array as a separate access.

Let lo(A) and up(A) be the lower and upper bounds of array A, respectively. Note that in Java programs, lo(A) will always be zero, but we specify it symbolically for generality. Let l_i and u_i be the lower and upper bounds of i, respectively. The safe iterations of i are defined as all iterations i such that $f(i) \geq \mathrm{lo}(A)$ and $f(i) \leq \mathrm{up}(A)$. Solving these two inequalities for i, and intersecting with the loop iteration space, gives the range of i that will not cause access violations:

$$\left(\left\lceil \frac{\mathrm{lo}(A) - b}{c} \right\rceil \leq i \leq \left\lfloor \frac{\mathrm{up}(A) - b}{c} \right\rfloor\right) \bigcap (l_i \leq i \leq u_i) \quad (1)$$

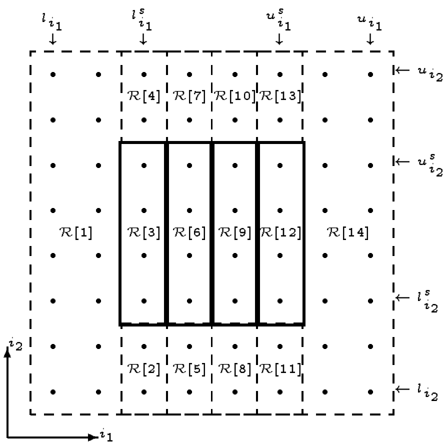
The intersection of all ranges of i defined by Eq. 1 for all references in loop i indexed by i defines the lower and upper bounds of the *safe region* for i. We denote these bounds l_i^s and u_i^s, respectively. The safe region for the loop i implicitly defines two other regions for loop i. One is the lower unsafe region, $l_i \leq i < l_i^s$, the other is the upper unsafe region $u_i^s < i \leq u_i$.

We note that if the array is known to be rectangular, then l_i^s and u_i^s need only be computed once per dimension. If the array is ragged then computing l_i^s and u_i^s for some reference requires computing it for each possible value of the other indices. That is, in a nested (i, j) loop, the values of l_j^s and u_j^s can, in general, be functions of i.

Null pointer checks can be performed on array base pointers and other pointers while computing the safe region. If a **null** is found, the safe region can be made empty, and all of the iterations placed in either the lower or upper unsafe region. Details on **null** pointer issues, handling the situation where the intersection of safe regions is empty, and other boundary cases, are found in [25].

2.2 Tiling the iteration space with regions

Consider a perfect loop nest defined by the loops $i_1, i_2, \ldots, i_d$. We use the full (l_{i_k}, u_{i_k}) and safe ($l_{i_k}^s$, $u_{i_k}^s$) bounds of the loops to tile the iteration space. An example of that is shown in Fig. 2 for a 2-dimensional iteration space. The outer loop is divided into at most two unsafe regions, plus its safe regions. The inner loop is, in the worst case, divided into at most three regions for each iteration of the safe region of the outer loop. In the example, there are four iterations in the outer loop safe region (*i.e.*, those iterations between $l_{i_1}^s$ and $u_{i_1}^s$). Note that each region has its own characteristics as to which reference tests need to be performed. We are particularly interested in finding the safe regions (no tests). We treat any region needing at least one test as needing all tests.



(a) Partitioning the iteration space into regions

- $\mathcal{R}[1]$: tests in i_1 and i_2
- $\mathcal{R}[2]$, $\mathcal{R}[5]$, $\mathcal{R}[8]$, $\mathcal{R}[11]$: tests in i_2
- $\mathcal{R}[3]$, $\mathcal{R}[6]$, $\mathcal{R}[9]$, $\mathcal{R}[12]$: no tests
- $\mathcal{R}[4]$, $\mathcal{R}[7]$, $\mathcal{R}[10]$, $\mathcal{R}[13]$: tests in i_2
- $\mathcal{R}[14]$: tests in i_1 and i_2

(b) Mandatory tests in each region

Fig. 2. Iteration space for a perfectly nested two-dimensional loop.

The procedure *regions*() in Fig. 3 recursively computes the regions that tile the iteration space of a perfect loop nest of depth d (see Fig. 5(a)). See [25] for a discussion on how to optimize non-perfect loop nests. Procedure *regions*() takes seven parameters. The first five are input parameters, and describe the loop nest being tiled. They are:

1. The index j indicating that region extents along index variable i_j are being computed.
2. The vector $(\alpha_1, \alpha_2, \ldots, \alpha_{j-1})$, where α_k is the lower bound for loop index i_k in the regions to be computed.
3. The vector $(\omega_1, \omega_2, \ldots, \omega_{j-1})$, where ω_k is the upper bound for loop index i_k in the regions to be computed.
4. The dimensionality (or depth) d of the loop nest.
5. The vector $\mathcal{B}[1 : d]$, where $\mathcal{B}[k] = (l_{i_k}, u_{i_k}, l^s_{i_k}, u^s_{i_k})$ contains the full and safe bounds (as computed in Section 2.1) for loop i_k.

The next two parameters are the output of the procedure. The first output parameter is the vector of regions, $\mathcal{R}$, which has the format:

$$\mathcal{R}[\delta] = (\mathbf{l}(\delta), \mathbf{u}(\delta), \tau), \tag{2}$$

$$\mathbf{l}(\delta) = (l_{i_1}(\delta), l_{i_2}(\delta), \ldots, l_{i_d}(\delta)), \tag{3}$$

$$\mathbf{u}(\delta) = (u_{i_1}(\delta), u_{i_2}(\delta), \ldots, u_{i_d}(\delta)), \tag{4}$$

$$\tau = \{\textbf{test} \mid \textbf{notest}\}. \tag{5}$$

The elements $l_{i_j}(\delta)$ and $u_{i_j}(\delta)$ denote the lower and upper bounds, respectively, of loop i_j in the region $\mathcal{R}[\delta]$. The definition of a region also contains a flag, τ, that has the value **test** if the region requires any array reference to be tested, and **notest** otherwise. The second output parameter is u_δ, the number of regions used to tile the iteration space.

Each invocation of procedure *regions* for some value of j tiles the loop nest formed by loops $i_j, i_{j+1}, \ldots, i_d$. Loop i_j is partitioned into three parts: the lower non-safe region $l_{i_j} \leq i_j < l^s_{i_j}$, the safe region $l^s_{i_j} \leq i_j \leq u^s_{i_j}$, and the upper non-safe region $u^s_{i_j} < i_j \leq u_{i_j}$. The lower non-safe region is formed in lines 2-5 of procedure *regions*, and the upper non-safe region is formed in lines 17-20. Note that once a region is non-safe in some loop of the loop nest, it is treated non-safe in all loops of the nest. Thus, the formation of the non-safe regions does not require further recursive calls to *regions*, but instead uses the full iteration space of nested loops.

The *safe* region is formed in lines 6-16. Lines 6-8 handle the case of i_j being the innermost loop by making the bounds of the safe region for loop i_j be the safe bounds computed in Section 2.1. Lines 9-16 handle the case when i_j is not the innermost loop. There are two possibilities: (i) the safe region of all loops nested within i_j is identical to the iteration space of the loops, or (ii) the first condition does not hold. The call to function *nochecks* (line 9) tests if this condition holds. If it does hold, the safe region of i_j and all nested loops form a single $(d - j + 1)$-dimensional safe region, which is computed in line 11. This leads to the tiling of the iteration space found in Fig. 4 (for $d = 2$). If *nocheck* returns **false** the condition does not hold and recursive calls are made to *regions* in line 11 to fully partition the region based on the safe region information for loops i_{j+1} through i_d. This leads to a tiling of the loop as shown in Fig. 2 (for $d = 2$).

procedure *regions*$(j, (\alpha_1, \alpha_2, \ldots, \alpha_{j-1}), (\omega_1, \omega_2, \ldots, \omega_{j-1}), d, \mathcal{B}, \mathcal{R}, u_\delta)$ {
  if $(l_{i_j} < l^s_{i_j})$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, l_{i_j}, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, l^s_{i_j} - 1, u_{i_{j+1}}, \ldots, u_{i_d}),$ **test**)
  }
  if $(j == d)$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{d-1}, l^s_{i_d}), (\omega_1, \ldots, \omega_{d-1}, u^s_{i_d}),$ **notest**)
  } **elsif** (*nochecks*$((l_{i_{j+1}}, l_{i_{j+2}}, \ldots, l_{i_d}), (l^s_{i_{j+1}}, l^s_{i_{j+2}}, \ldots, l^s_{i_d}),$
      $(u_{i_{j+1}}, u_{i_{j+2}}, \ldots, u_{i_d}), (u^s_{i_{j+1}}, u^s_{i_{j+2}}, \ldots, u^s_{i_d}))$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, l^s_{i_j}, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, u^s_{i_j}, u_{i_{j+1}}, \ldots, u_{i_d}),$ **notest**)
  } **else** {
    for $(k = l^s_{i_j};\ k \le u^s_{i_j};\ k{+}{+})$ {
      regions$(\mathcal{R}, j + 1, (\alpha_1, \alpha_2, \ldots, \alpha_{j-1}, k), (\omega_1, \omega_2, \ldots, \omega_{j-1}, k), d, \mathcal{B}, \mathcal{R}, u_\delta)$
    }
  }
  if $(u_{i_j} > u^s_{i_j})$ {
    $u_\delta = u_\delta + 1$
    $\mathcal{R}[u_\delta] = ((\alpha_1, \ldots, \alpha_{j-1}, u^s_{i_{j+1}}, l_{i_{j+1}}, \ldots, l_{i_d}), (\omega_1, \ldots, \omega_{j-1}, u_{i_j}, u_{i_{j+1}}, \ldots, u_{i_d}),$ **test**)
  }
}

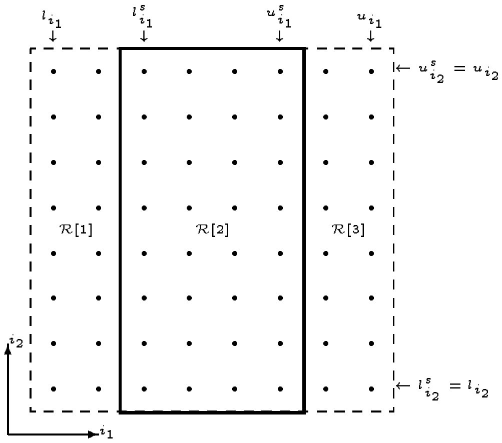
boolean function *nochecks*$((l_{i_1}, \ldots, l_{i_m}), (l^s_{i_1}, \ldots, l^s_{i_m}), (u_{i_1}, \ldots, u_{i_m}), (u^s_{i_1}, \ldots, u^s_{i_m}))$ {
  if $(((l_{i_j} = l^s_{i_j}) \wedge (u_{i_j} = u^s_{i_j}))\forall j = 1, \ldots, m)$
    return true
  else
    return false
}

Fig. 3. Procedure to compute the regions for a loop nest.

2.3 Code generation

Consider a general d-dimensional rectangular loop nest, as shown in Fig. 5(a). The body of the loop is denoted by $B(i_1, i_2, \ldots, i_d)$. Fig. 5(b) shows the code that is generated to form a loop nest that is exception-free on access violations. The version of the loop body that requires no tests is denoted $B_{\textbf{notest}}(i_1, i_2, \ldots, i_d)$, and the version of the body that explicitly performs all tests specified by the Java VM specification is denoted by $B_{\textbf{test}}(i_1, i_2, \ldots, i_d)$.

In Fig. 5(b), a call is first made to *regions* to form the safe and unsafe regions. The δ driver loop iterates over the regions. If the region is not safe and requires tests, the first loop, whose body contains all necessary tests, is executed. Otherwise, the second, check-free loop nest, is executed.



(a) Partitioning of the iteration space into regions

- $\mathcal{R}[1]$: tests in i_1 and i_2
- $\mathcal{R}[2]$: no tests
- $\mathcal{R}[3]$: tests in i_1 and i_2

(b) Mandatory tests in each region

Fig. 4. Iteration space for a perfectly nested two-dimensional loop.

for $(i_1 = l_{i_1}; i_1 \leq u_{i_1}; i_1\text{++})$
  for $(i_2 = l_{i_2}; i_2 \leq u_{i_2}; i_2\text{++})$
    $\ldots$
      for $(i_d = l_{i_d}; i_d \leq u_{i_d}; i_d\text{++})$
        $B(i_1, i_2, \ldots, i_d)$

(a) A d-dimensional loop nest

$regions(1,\ (\ ),\ (\ ),\ d,\ \mathcal{B},\ \mathcal{R},\ u_\delta = 0)$
for $(\delta = 1; \delta \leq u_\delta; \delta\text{++})$ {
  if $(\mathcal{R}[\delta].\tau ==$ **test**$)$ {
    for $(i_1 = l_{i_1}(\delta); i_1 \leq u_{i_1}(\delta); i_1\text{++})$
      for $(i_2 = l_{i_2}(\delta); i_2 \leq u_{i_2}(\delta); i_2\text{++})$
        $\ldots$
          for $(i_d = l_{i_d}(\delta); i_d \leq u_{i_d}(\delta); i_d\text{++})$
            $B_{\textbf{test}}(i_1, i_2, \ldots, i_d)$
  } **else** {
    for $(i_1 = l_{i_1}(\delta); i_1 \leq u_{i_1}(\delta); i_1\text{++})$
      for $(i_2 = l_{i_2}(\delta); i_2 \leq u_{i_2}(\delta); i_2\text{++})$
        $\ldots$
          for $(i_d = l_{i_d}(\delta); i_d \leq u_{i_d}(\delta); i_d\text{++})$
            $B_{\textbf{notest}}(i_1, i_2, \ldots, \imath_d)$
  }
}

(b) The loop-nest transformed

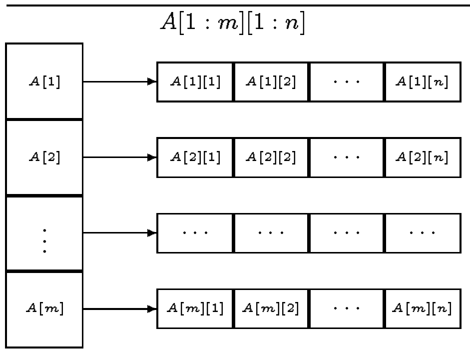
Fig. 5. Bounds optimization code generation example

2.4 Thread safety

Threads are an integral part of the Java language specification. In general, it is not possible to determine at compile-time if a method will be sharing data with other concurrent threads as it executes. Therefore, it is necessary that our optimization be thread safe.

Thread safety of the bounds checking optimization can be ensured if the bounds of an array used during the execution of procedure *regions* is the same as the bounds of the array when it is accessed in the **notest** loop.

Fig. 6(a) shows the layout of a 2-dimensional Java array. Even though lower bounds in Java are always zero, the transformation we are discussing is applicable to other languages. Therefore, we will use non-zero lower bounds in some examples. Each vector (either data vector or pointer vector) contains information about the extent of that vector. The bounds of an array change when an element of a pointer vector is assigned to point to another (data or pointer) vector, whose bounds are different than what originally existed. Issues of thread safety arise when another thread is changing the bounds of an array that is accessed by a concurrently executing thread running code optimized by the transformation of this section. We denote the thread changing the array bound as T_c, and the thread executing optimized code that accesses array elements as T_o, and give a worst case scenario. We stress that while asynchronously changing array extents may be poor programming practice, it is legal Java.



(a) layout of 2-dimensional array

double[][]A' = **new double**[$1 : A$.length][]
$n = \infty$
for ($i = 1; i \leq A$.length; i++) {
  $A'[i] = A[i]$
  $n = \min(n, A[i]$.length)
}

(b) making a private copy

Fig. 6. Thread-safety issues are handled by privatizing data.

First, T_o executes *regions*, and computes l_i^s and u_i^s for the loop index i. Thread T_c then changes the extent of one vector of floating point numbers (v) to be shorter (so much shorter that u_i^s is no longer valid) by replacing it with v'. The storage after v' happens to be filled with values that are illegal floating point numbers. Next, during the execution of the **notest** loop, the value of the subscript function $f(i)$ exceeds the length of v', and a value that is an illegal floating point number is fetched. An operation is executed on the value, and the program is abnormally terminated. Two non-compliant events have occurred: (i) an out-of-bounds access was performed and no exception was thrown, and (ii) an exception that would not have been thrown in the original program was thrown.

Fortunately, the Java memory model provides a solution to this problem. Java allows keeping copies of shared (or *main memory*) values in thread-private memory (or *working memory*) between synchronization points. To guarantee thread safety, we keep the vectors of pointers in private memory. Thus, referring to Fig. 6(a), (the values of) the $A[1:m]$ vector will be kept in private memory during computation. These private values are then used by procedure *regions*, and the **test** and **notest** loops. Thus, any races and asynchronous changes in the size of the array will occur before the computation of l_i^s and u_i^s, and these values will be consistent throughout the execution of the modified code.

We show how to implement the privatization by copying array $A[1:m]$ into a private array $A'[1:m]$ in Fig. 6(b). While copying we can find the minimum extent of a row of A. In computing the safe bounds, A can be treated as rectangular, with this minimum extent as its second-axis extent. Matrix multiplication of $n \times n$ matrices is an $O(n^3)$ operation. The cost of privatizing and verifying the extents of one of the matrices is only $O(n)$. This is certainly insignificant for $n = 500$. We discuss some approaches to deal with very small n or less intensive computations in Section 4.

Finally, we discuss here an alternative to this regions approach. A normal execution of a method such as `matmul` in Fig. 1 is expected to generate no exceptions. An execution that causes an exception is most likely a misuse and it is not really important to optimize. Optimization of the normal case (no exceptions) can be achieved by a simple set of range tests before the execution of a loop nest. Let $|A|_d$ be the length of array A along axis d. The triple-nested loop of Fig. 1 can be transformed to:

if $(m \leq |C|_1) \wedge (p \leq |C|_2) \wedge (m \leq |A|_1) \wedge (n \leq |A|_2) \wedge (n \leq |B|_1) \wedge (p \leq |B|_2)$ {
  notest version of loop-nest
else {
  test version of loop-nest
}

In situations where all iterations of a loop are in a single region, the transformation given in this section is, for all practical purposes, equivalent to implementing two version-code for the loop. This is what has been done in this paper to obtain the experimental results. However, we expect that there will be a trend towards developing codes in which access violations are normal and handled internally. In this case, the regions approach allows at least some of the computation to execute fast.

3 Optimizations

In this section, we describe the impact of various program transformations on the performance of the MATMUL program executing on a single processor. This study was performed using the IBM HPCJ compiler for the POWER family of architectures [29], which converts the Java byte-code into native code of the target machine. The kernel of the MATMUL program is shown in Fig. 1. The transformations that we applied to this program are described below.

3.1 Program Transformations

The first program transformation reduces the overhead of checking for out-of-bounds array access violations, as described in Section 2. We generate two versions of the loop nest, one in which bounds checking computations are performed for each array reference, and the other in which no such computations are needed. A correctly written program is expected to spend all of its time in the loop nest without bounds checking. This transformation not only reduces the overhead of array bounds checking, but also creates program regions that are provably free from exceptions. Those regions can now be transformed using standard loop transformation techniques [32, 3, 30, 26] without violating Java semantics.

The next transformation on the program is to tile (block) the loop nest in the exception-free region, to improve the cache-performance by promoting greater reuse of data [31, 26]. Tiling requires both strip-mining, which does not change the order of computation, and loop-interchange, which does change the order of computation. Loop-interchange (and tiling) can be legally performed only on that part of a Java program in which no exception may be thrown. Based on the existing algorithm in the IBM XLF compiler [26] to select the loop blocking parameters (which uses estimates of the number of distinct cache lines used by the computation [16]), we choose strip size of 40 for each of the inner i, j, and k loops.

Following loop tiling, we perform loop-unrolling [26] to increase the size of the loop body. This enables the back-end to generate a better schedule that exploits the instruction-level parallelism. It also enhances data reuse within registers. We perform outer loop-unrolling, where both the i and the j loops are unrolled by a factor of 4 each.

The next program version uses scalar replacement [8, 26], where references to the array being written, C, are replaced by references to scalar temporary variables in the innermost loop. Additional statements are used to load the temporaries from the array at the beginning of the loop iteration and to store them back at the end. It is well-known that scalar replacement enables the compiler back-end to improve register-allocation for the code, as most back-ends are able to disambiguate scalar references more effectively.

Finally, to determine the performance benefits from the **looseNumerics** extension to Java [17] (described in Section 4), which allows the use of extended-precision (greater than 32- and 64-bit) arithmetic, we modified the compiler options that are passed on by the Java compiler to the back-end. We turned on the option that allows the back-end to use the `fma` (fused multiply-add) instruction. The `fma` instruction in the POWER family of architectures, `fma` a, b, c, x, computes $x = a + b * c$.

The programs were compiled using the `-O` option of the IBM HPCJ compiler, and with the `-qarch` and `-qtune` options set to turn on code tuning for the respective PowerPC and POWER2 architectures.

3.2 Experimental Results

Fig. 7 shows the performance of different versions of the MATMUL program on a PowerPC 604e and a POWER2 workstation, expressed as a fraction of the performance obtained for the best version written in C++. On the PowerPC, the base-line Java version achieves a performance of only 4.9 MFlops. After the transformation to create a safe region with no array bounds checking, the performance improves to 11.1 MFlops. The indirect impact of that transformation is even more significant. The application of tiling, which is enabled by that transformation, followed by other loop transformations

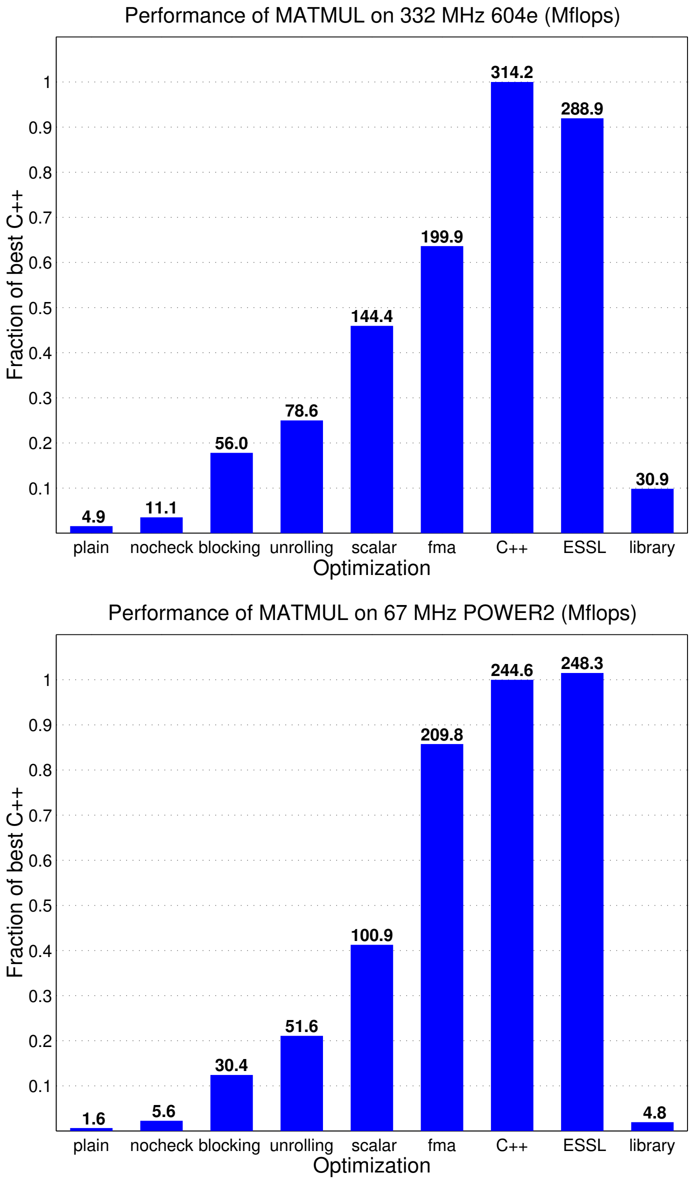


Fig. 7. Summary of results.

described above, steadily improves the performance of the program. The final Java-compliant optimized program achieves a performance of 144.4 MFlops on the PowerPC, while the version which exploits the proposed **looseNumerics** Java extension (by using `fma` instructions) achieves 199.9 MFlops. This is is 63.6% of the performance of the equivalent C++ version with identical loop transformations, and 69.2% of the hand-tuned ESSL version.

On the POWER2, the impact of these transformations is even more dramatic. The base-line Java program achieves 1.6 MFlops, whereas the optimized version with `fma` instructions achieves a performance of 209.8 MFlops, for an improvement by a factor of 131. The fully optimized Java program achieves 85.8% of the performance of the corresponding C++ version, and 84.5% of the ESSL performance.

For both machines, we also show the performance of the best Java version (with `fma` instructions) where only the transformation to create safe, exception-free, regions is not performed (these are the "library" bars). It would not be legal for a compiler to derive such a version from the given base-line program, as it has different semantics under Java when exceptions are possible. However, this measurement helps quantify the overhead of array bounds checking in an otherwise tuned (BLAS-3 style [15]) matrix-multiply routine. It can be seen that the performance of this version is quite poor on both the machines: 30.9 MFlops on a PowerPC and 4.8 MFlops on a POWER2.

In summary, the transformation to create exception-free regions in the MATMUL program has a large positive impact on the performance of the program, in two ways. First, it eliminates the actual overhead of bounds checking (which is more significant on the higher-performance version of the program). Second, it enables other loop transformations, which can be performed automatically by a compiler.

4 Java Enhancements

Proposing extensions and enhancements to Java is a delicate subject. It is tempting to propose extensions that fix one's particular problem. However, we have to keep in mind that Java is aimed at being 100% compatible across all platforms, and that there is a very specific procedure for making official changes to the language. Rather than proposing enhancements ourselves, we discuss here the impact of some extensions proposed in [17]. We believe those extensions have a reasonable chance of making to the official standard in the near future. The extensions that most directly impact our work in this paper are three: (i) the **looseNumerics** modifier, (ii) the **idealizedNumerics** modifier, and (iii) the *Matrices* objects. We discuss each of these extensions in more detail below.

4.1 looseNumerics

The **looseNumerics** modifier can be added to a method or to a class. It allows the floating-point operators to return values that are more precise than the strict definition for IEEE 754 32-bit **float** and 64-bit **double**. It also allows intermediate values on the operand stack and local variables to hold values in extended precision. Because the exact "loosening" (number of extended bits used) is implementation-dependent, different JVM implementations may return different values for the same sequence of floating-point computation. For example, the result of $x = a + b * c$, where a, b, and c are floating-point values, can differ on the least-significant bit of x, depending on the implementation. This difference would be observable even if x is coerced to one of the strict **float** or **double** representations. To apply this modifier to our `matmul` we would simply change its header to:

```
static looseNumerics matmul(double[][] A, double[][]B,
                            double[][] C,
                            int m, int n, int p)
```

The performance impact of this modifier can be substantial. For the *x86* architecture, it allows intermediate and local variables to be kept in the 80-bit operand stack. This drastically reduces the number of explicit operations to coerce values to precise 32-bit and 64-bit formats. For the POWER family of architectures, used in the RS/6000 products, this modifier allows the `fma` instruction to be used. The intermediate multiplication result has a 106-bit mantissa, and therefore it is not legal by the strict floating-point semantics of Java. Therefore, the `fma` instruction is not currently in use by HPCJ and other Java environments for the RS/6000.

The ability to use `fma` instructions doubles the peak floating-point performance of POWER processors. A 332 MHz PowerPC 604e can achieve 664 Mflops peak and a 67 MHz POWER2 can achieve 266 Mflops peak. In terms of impact on our MATMUL benchmark, the results are summarized in Table 1. For each type of processor (604e and POWER2) we list the peak performance achieved in MATMUL in four cases: (i) strict Java, (ii) Java with `fmas`, (iii) C++ with `fmas` disabled, and (iv) regular C++ with `fmas` enabled. We observe performance improvements from 40 to 125% by enabling the `fma` instructions.

processor	performance (Mflops)			
	Java	Java+`fma`	C++ (no `fma`)	C++
604e/332 MHz	144	200	187	314
POWER2/67 MHz	101	210	100	225

Table 1. Summary of results from enabling `fma` instructions.

Recently, Sun has proposed a modification to the Java programming language and Java virtual machine specification. This modification would allow the use of extended precision in some situations, including MATMUL, even without the **looseNumerics** modifier. The public review of this proposal is expected to begin May 1st, 1998 [24].

4.2 idealizedNumerics

The **idealizedNumerics** modifier can also be applied to a method or a class. It allows the compiler/execution environment to treat all floating-point operations as if they were performed with infinite accuracy. One of the more important implications for performance is that addition and multiplication can be treated as associative (*i.e.*, $a+b+c = (a+b)+c = a+(b+c)$). Associativity is a fundamental property to enable the parallelization of recurrences (parallel-prefix is an example of a strategy enabled by associativity). Associativity allows the computation of a recurrence to be partitioned into segments. Each segment computes a partial result, that at the end is contributed to the final result.

In MATMUL, we honored the non-associativity of IEEE floating-point addition by unrolling and parallelizing (see Section 3 and Section 5) only the outer i and j loops, which do not carry a reduction dependence. Consider, however, the computation of the dot-product $s = \sum_{i=0}^{n-1} x_i y_i$ of two vector x and y of n elements. (This is what the inner k loop of MATMUL is doing.) A straightforward implementation of this dot-product operation in Java and C++ is shown in Fig. 8.

```
double ddot(int n, double[] x, double[] y) {
    double s = 0;
    for (int i=0; i<n; i++) {
        s += x[i]*y[i];
    }
    return s;
}
```

Fig. 8. Straightforward implementation of $s = \sum_{i=0}^{n-1} x_i y_i$.

The IBM `xlC` C++ compiler parallelizes the computation of dot-product at the instruction level. It first decomposes the original dot-product into 4 interleaved dot-products, and in the end it adds the partial results together. The corresponding implementation is shown in Fig. 9. Note that this transformation is illegal in Java without the **idealizedNumerics** modifier. This transformation allows better utilization of the pipelines and dual floating-point units of the POWER2. In an RS/6000 model 590, the performance of dot-product is boosted from 67 Mflops to 174 Mflops by this transformation.

4.3 *Matrices*

Java does not directly support multi-dimensional arrays. Multi-dimensional arrays are well supported in Fortran 90 and, to a lesser extent, in C++. Some of the functionality of multi-dimensional arrays is provided in Java by arrays of arrays, but the two are not entirely equivalent. Therefore, [17] includes a proposal for Fortran-like matrix objects. Fig. 10 shows the difference between a Fortran-like $m \times n$ two-dimensional array (represented by $A(1:m, 1:n)$) and a Java-like array with m elements, each an n-element array (represented by $A[1:m][1:n]$).

There are several implications to this difference in organizations between $A(1:m, 1:n)$ and $A[1:m][1:n]$:

```
double ddot(int n, double[] x, double[] y) {
    double s, s0, s1, s2, s3;
    s = s0 = s1 = s2 = s3 = 0;
    for (int i=0; i<n-3; i+=4) {
        s0 += x[i]*y[i];
        s1 += x[i+1]*y[i+1];
        s2 += x[i+2]*y[i+2];
        s3 += x[i+3]*y[i+3];
    }
    s = s0+s1+s2+s3;
    for (; i<n; i++) {
        s += x[i]*y[i];
    }
    return s;
}
```

Fig. 9. 4-way unrolled implementation of dot-product.

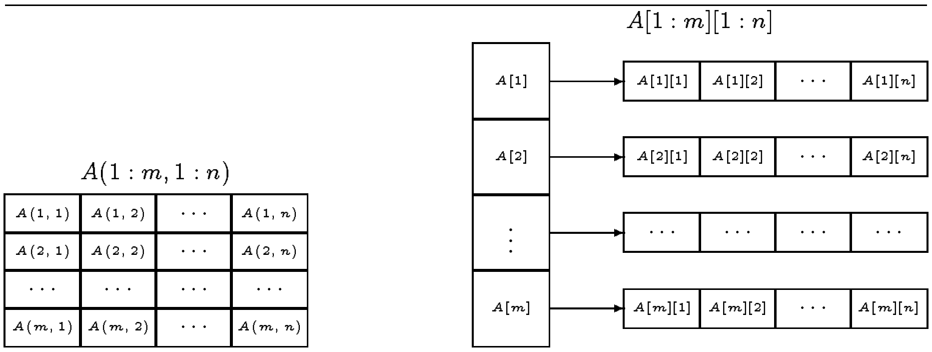


Fig. 10. Differences between an $m \times n$ array and an m-element array of n-element arrays.

1. The location of an element $A(i, j)$ can be computed using simple arithmetic. Finding $A[i][j]$ involves, in general, pointer chasing.
2. There is no guarantee that $A[1 : m][1 : n]$ is rectangular. Even if it was rectangular when created it may have been modified. For $O(n^3)$ operations, like MATMUL and BLAS3 in general, run-time checking that an array of arrays is rectangular represents a negligible overhead. The same is not necessarily true for $O(n^2)$ operations or for very small values of n.
3. Aliasing disambiguation. Given two matrix objects A and B it is sufficient to show that $A \neq B$ to prove there is no aliasing between them. In the case of two arrays of arrays $A[][]$ and $B[][]$ one has to show that $A[i] \neq B[j]\ \forall i, j$ to prove there is no aliasing.
4. Regular array sections for the matrix object $A(1 : m, 1 : n)$ can be represented compactly. Because these sections are necessarily rectangular, it suffices to store

one triplet $(l : u : s)$ representing the lower bound l, upper bound u, and stride s of the section for each axis.

5. Disambiguation between two array sections $A(l_1^A : u_1^A : s_1^A, l_2^A : u_2^A : s_2^A)$ and $B(l_1^B : u_1^B : s_1^B, l_2^B : u_2^B : s_2^B)$ reduces to proving that the source arrays are different ($A \neq B$) or that the sections do not intersect ($((l_1^A : u_1^A : s_1^A) \cap (l_1^B : u_1^B : s_1^B) = \emptyset$ or $(l_2^A : u_2^A : s_2^A) \cap (l_2^B : u_2^B : s_2^B) = \emptyset)$.
6. Privatizing matrix $A(1 : m, 1 : n)$, for thread safety, requires copying only one reference. Privatizing array $A[1 : m][1 : n]$ requires creating a new array with m elements and copying m references. This can become a performance problem for array operations with complexity $O(mn)$ or less.

Basically, matrices are uncoupled axes data structures while arrays of arrays are coupled axes data structures. The length of row $A[i]$ depends on i and there is not a clear meaning to the length of column j. The length of row $A(i, :)$ is always n and the length of column $A(:, j)$ is always m.

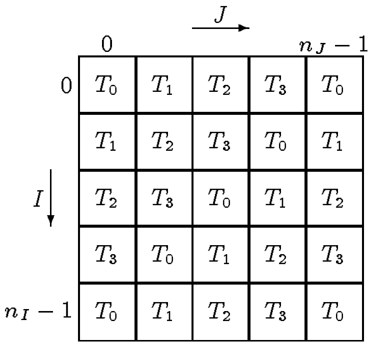
One alternative to having explicit rectangular arrays is to have the compiler identify Java arrays that have rectangular shape. Such approach is described in [10]. For general applicability, it requires good inter-procedural analysis in the presence of multithreading. It also requires good alias analysis. Even if the shape of an array changes during the course of a computation, proving that it is rectangular and fixed during certain regions can be very beneficial.

5 Parallelization

The final step in optimizing the performance of MATMUL is parallelization. There is plenty of parallelism to exploit in that computation, and parallelizing the C++ version is straightforward. However, it is not possible to directly parallelize the (i, j, k) loop nest of the Java version of MATMUL. Java requires the execution of that loop nest to follow a strict sequential semantics. Because exceptions can be thrown in any iteration of the loop nest, we have to guarantee that the values of $C[i][j]$ are modified in the sequential order of the iterations.

Fortunately, the transformation of Section 2 creates regions that are exception-free. For each of these regions, we know for sure that all the iteration points in the region will execute. The same transformation that enabled blocking, unrolling, and the other optimizations of Section 3 also enables parallelization.

We parallelize the last version from Section 3, which includes blocking, unrolling, and scalar replacement. The loop nest in this version is six-dimensional. There are 3 loops over the blocks (I,J, and K), and 3 loops over the elements in those blocks (i,j, and k). As a good compromise between load balance and overhead, we parallelize the two outer block loops I and J. This corresponds to parallelization by partitioning the computation of the blocks of the C matrix. The parallel execution is performed by n_T threads. Each threads is numbered from 0 to $n_T - 1$. The variable t in each thread contains the identifying number for that thread. The computation of the blocks of matrix C are static scheduled among n_T threads as follows: (i) the blocks are ordered in row-major order and (ii) are assigned cyclically to the threads. Fig. 11 shows the partitioning of 200×200 matrix C onto four threads, identified by T_0, T_1, T_2, and T_3. We use blocks of size 40×40 and, therefore, there are 25 blocks overall.



for $(b = t; b < n_I n_J; b\mathrel{+}= n_T)$ {
  $I = (b/n_I) \times b_I$
  $J = (b \bmod n_I) \times b_J$
  for $(K = 0; K < n; K\mathrel{+}= b_K)$ {
    for $(i = I; i < \min(m, I + b_I); i{+}{+})$
      for $(j = J; j < \min(p, J + b_J); j{+}{+})$
        for $(k = K; k < \min(n, K + b_K); k{+}{+})$
          $C_{i,j} \mathrel{+}= A_{i,k} B_{k,j}$
  }
}

Fig. 11. Partitioning of a 200×200 matrix C onto four threads.

Also shown in Fig. 11 is the pseudo-code of the computation performed by each thread. The variable b works as a block counter. It starts from t, the thread number, and it is incremented by n_T, the number of threads, in each iteration. There are a total of $n_I \times n_J$ blocks. The corresponding values for I and J can be directly computed from the value of b. The other loops (K, i, j, and k) are exactly as in Section 3. For the sake of clarity, we do not show here the unrolling of the i and j loops or the scalar replacement.

The results from our parallelization are summarized in Fig. 12. We group the results by number of threads (1, 2, 3, or 4) and within each group we show the performance of three versions of MATMUL: (i) a version that conforms to current Java semantics (identified by JAVA), (ii) a version with Java using the `fma` instructions (identified by JFMA), and (iii) a C++ version (identified by C++). The numbers at the top of the bars are the absolute Mflops achieved by each version on that number of threads. The speedup in each case, indicated by the length of the bar along the y-axis, is computed with respect to the best sequential case, 314 Mflops. (The performance of a Fortran version we tried was no better.)

We complete the reporting of our results with Table 2. In that table we list the Mflops achieved in each version with different number of threads. We also list the best serial performance achieved for each version. The S column represents the version-specific speedup, computed with respect to the best corresponding serial performance. Finally, the f_{ESSL} column list the fraction of the top ESSL performance (1183 Mflops) achieved in each case.

Overall, we observe good scaling with the number of threads in all three versions: JAVA, JFMA, and C++. The threading mechanisms is inherently the same for all versions, since it is directly provided by AIX. The overhead for multithreaded operation is, therefore, proportionally larger for the faster versions. That explains why the version-

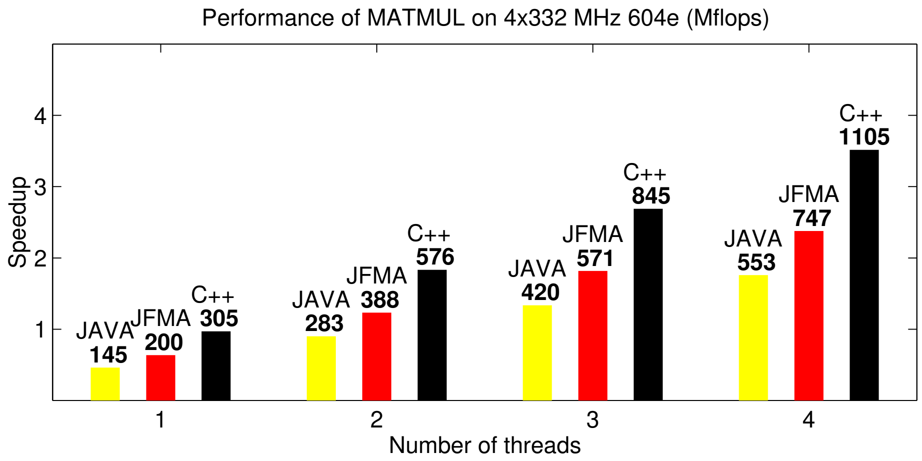


Fig. 12. Performance results comparing Java, Java with `fmas`, and C++.

# threads	JAVA			JFMA			C++		
	Mflops	S	f_{ESSL}	Mflops	S	f_{ESSL}	Mflops	S	f_{ESSL}
serial	144	1.0	0.12	200	1.0	0.17	314	1.0	0.27
1	145	1.0	0.12	200	1.0	0.17	305	1.0	0.26
2	283	2.0	0.24	388	1.9	0.33	576	1.8	0.49
3	420	2.9	0.35	571	2.9	0.48	845	2.7	0.71
4	553	3.8	0.47	747	3.7	0.63	1105	3.5	0.93

Table 2. Summary of results for parallelization.

specific speedup on four threads is 3.8 for JAVA, 3.7 for JFMA, and only 3.5 for C++. In terms of absolute performance, the C++ version achieves more than 90% of the peak ESSL performance. The version of Java with `fma` instructions, JFMA, achieves almost $2/3$ of the peak ESSL performance. While this is still 30% less than the C++ version, it is a very respectable result. Even the strict Java version, JAVA, achieved more 500 Mflops. This represents a more than 100-fold improvement in performance over our starting point!

6 Related Work

Examples of projects using Java for substantial numerical computing are described in [2, 6, 9, 27]. In some cases, results have been positive despite the performance disadvantages of Java. As shown in [6], for some computations and certain machine configurations, current Java implementations can achieve within 50% of C performance. We note however, that the best performance reported in [6] for matrix multiplication in Java is only 50 Mflops (on a slower processor).

There are two main approaches in the literature to optimizing array bounds checks: (i) the use of static data-flow analysis information to determine that a test is unnecessary [12, 21, 13, 14, 28], and (ii) the use of data-flow information and symbolic analysis at compile time to reduce the dynamic number of tests remaining in the program [22, 20, 1, 19, 23].

Work in the first group uses data-flow information to prove at compile time that an array bounds violation cannot occur at run-time, and therefore that the test for the violation is unnecessary. Using the terms of our discussion, the goal of this work is to identify loops that are safe regions. In contrast, the goal of our work is to transform loops in order to *create* safe regions.

Work in the second group attempts to reduce the dynamic and static number of bounds tests. It also attempts to reduce the overhead induced by a test even if it cannot be eliminated. This is done (i) by hoisting tests out of loops when possible [23] and (ii) by also determining that a test is covered by another test [22, 20, 1, 19] and can be eliminated.

Neither of these optimizations are usable with Java in general because the Java semantics requiring precise exceptions make the hoisting and reordering of tests illegal in many situations. Also, when an access violation exception is caught by a **try** block in the loop body, the loop should not be terminated (as would occur when hoisting tests).

We emphasize that creating safe regions has the important effect of enabling an entire set of powerful optimization techniques. These techniques are constrained in Java by the requirement of precise exceptions. An alternative approach that enables this optimizations even in presence of (potential) exceptions is *speculative execution*. A fully transformed and optimized version of a loop nest (or code fragment in general) can be generated and executed speculatively. If an access violation is detected, then the computation is rolled-back and restarted in strict order. This approach is analogous to speculative execution to explore parallelism in the presence of (potential) data dependences. Note that some mechanism to detect violations is necessary, so this approach does not get the benefit of eliminating the actual tests.

Finally, several other projects have looked at aggressively optimizing Java. In [5] and [4], the **javar** and **javab** projects are described. These projects are closely related, with **javar** compiling Java source code and **javab** compiling byte code. Both produce Java byte codes. The **javar** program supports user specified parallelism, and the **javab** performs automatic parallelization. In [7], a project begun at the JavaSoft division of Sun Microsystems and its follow-on at Rice University are outlined. The work was aimed at a *dynamic* (or *just-in-time*) compilation system, and focused on local scalar optimizations. Hand compilations using a technique called *object inlining* achieved up to a 7-fold speedup on an *Oopack* routine (an object oriented version of Linpack). The work of both of these projects are complementary to our work, and show the gains that can be made in Java performance.

7 Conclusions

Eliminating and/or reducing the cost of verifying array references for validity is a key factor for achieving high-performance in Java for technical computing. The method we presented in this paper partitions the iteration space of a loop-nest in regions with different access violation characteristics. For technical computing, we expect most or all of the work to be performed in safe and exception-free regions. The benefit of creating safe regions is two-fold: (i) code to explicitly check for valid references in unnecessary in those regions and (ii) operation-reordering optimizations are allowed.

By applying high-order loop transformations (available today in the IBM XLF compiler) we have achieved single-processor Java performance in MATMUL that is within 65-85% of the best library and C++ versions. We want to stress the fact that there are no fundamental reasons for the Java code not to achieve the same performance of C++ in the serial and parallel versions of MATMUL. The current differences are due to differences in the compilers, which result in different instructions schedules. As part of our future work, we want to investigate the causes of these differences and to provide the necessary fixes.

The safe regions also open up the possibility of automatic parallelization of Java codes. Automatic parallelization is being developed at IBM within the context of the *Toronto Portable Optimizer* (TPO) framework. This framework operates at an intermediate language level, and it is equally applicable to Fortran, C, C++, and Java. Parallelization of MATMUL yielded almost linear speedup for Java, achieving 750 Mflops on 4 $\times$ 332 MHz PowerPC 604e machine. We are looking forward to soon smashing the 1 Gflop barrier!

Acknowledgements: The authors wish to thank Rick Lawrence and Marc Snir for fruitful technical discussions, and for strongly supporting our research.

References

1. J.M. Asuru. Optimization of array subscript range checks. *ACM Letters on Programming Languages and Systems*, 1(2):109–118, June 1992.
2. C. A. Atwood, R. P. Garg, and D. DeRyke. A prototype computational fluid dynamics case study in Java. *Concurrency, Pract. Exp. (UK)*, 9(11):1311–18, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.
3. U. Banerjee. Unimodular transformations of double loops. In *Proc. Third Workshop on Programming Languages and Compilers for Parallel Computing*, Irvine, California, August 1990.
4. A. Bik and D. Gannon. **javab** manual (version 1.0 BETA. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. Available at URL `http://www.cs.ucsb.edu/conferences/java98`.

5. A. Bik, J. Villacis, and D. Gannon. **javar**: A prototype Java restructuring compiler. *Concurrency, Pract. Exp. (UK)*, 9(11):1181–91, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.
6. R. F. Boisvert, J. J. Dongarra, R. Pozo, K. A. Remington, and G. W. Stewart. Developing numerical libraries in Java. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.
7. Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. *Concurrency, Pract. Exp. (UK)*, 9(11):445–63, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.
8. D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proc. ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.
9. H. Casanova, J. Dongarra, and D. M. Doolin. Java access to numerical libraries. *Concurrency, Pract. Exp. (UK)*, 9(11):1279–91, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II Las Vegas, NV, USA 21 June 1997.
10. M. Cierniak and W. Li. Just-in-time optimization for high-performance Java programs. *Concurrency, Pract. Exp. (UK)*, 9(11):1063–73, November 1997. Java for Computational Science and Engineering - Simulation and Modeling II, Las Vegas, NV, June 21, 1997.
11. IBM Corporation. IBM High Performance Compiler for Java, 1997. Available for download at `http://www.alphaWorks.ibm.com/formula`.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the 4'th ACM Symposium on Principles of Programming Languages*, pages 238–252, January 1977.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5'th ACM Symposium on Principles of Programming Languages*, pages 84–96, January 1978.
14. P. Cousot and N. Halbwachs. Automatic proofs of the absence of common runtime errors. In *Conference Record of the 5'th ACM Symposium on Principles of Programming Languages*, pages 105–118, January 1978.
15. J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst. *Solving Linear Systems on Vector and Shared Memory Computers*. Society for Industrial and Applied Mathematics, 1991.
16. J. Ferrante, V. Sarkar, and W. Thrash. On estimating and enhancing cache effectiveness. In *Proc. Fourth Workshop on Programming Languages and Compilers for Parallel Computing*, August 1991.
17. James Gosling. The evolution of numerical computing in Java. Document available at Webpage `http://java.sun.com/people/jag/FP.html`. Sun Microsystems.
18. James Gosling, Bill Joy, and Guy Steele. *The Java*$^{\text{(TM)}}$ *Language Specification*. Addison-Wesley, 1996.
19. R. Gupta. A fresh look at optimizing array bounds checking. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 272–282, June 1990.
20. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 2(1-4):135–150, March–December 1993.
21. W.H. Harrison. Compiler analysis for the value ranges for variables. *IEEE Transactions on Software Engineering*, SE3(3):243–250, May 1977.
22. P. Kolte and M. Wolfe. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 270–278, Jun 1995.

23. V. Markstein, J. Cocke, and P. Markstein. Elimination of redundant array subscript range checks. In *Proceedings of the ACM SIGPLAN '82 Conference on Programming Language Design and Implementation*, pages 114–119, June 1982.
24. Sun Microsystems. Sun proposes modification to Java programming languages's floating point specification. `http://www.sun.com/smi/Press/sunflash/9803/sunflash.980324.17.html`, 1998.
25. S. P. Midkiff, J. E. Moreira, and M. Snir. Optimizing bounds checking in java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.
26. V. Sarkar. Automatic selection of high-order transformations in the IBM XL Fortran compilers. *IBM Journal of Research and Development*, 41(3), May 1997.
27. M. Schwab and J. Schroeder. Algebraic java classes for numerical optimization. In *ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM SIGPLAN, 1998. Available at `http://www.cs.ucsb.edu/conferences/java98`.
28. B. Schwarz, W. Kirchgassner, and R. Landwehr. An optimizer for Ada – design, experience and results. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 175–185, June 1988.
29. V. Seshadri. IBM high performance compiler for Java. *AIXpert Magazine*, September 1997. Electronic publication available at URL `http://www.developer.ibm.com/library/aixpert`.
30. M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
31. M. J. Wolfe. Iteration space tiling for memory hierarchies. In *Proc. 3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, Philadelphia, PA, 1987.
32. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, 1989.

This article was processed using the LaTeX macro package with LLNCS style