MAKING NUMERICAL JAVA PROGRAMS EXECUTE FASTER

BY

ZEHRA NOMAN SURA

B.E., Nagpur University, 1998

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2001

Urbana, Illinois

Abstract

The Java programming language encourages us to write programs that are structured and easy to understand. Features like object-oriented design, dynamic class loading, dynamic method invocation, automatic storage management, type safety, precise exception handling, and support for multithreading, graphics, and networking make it the language of choice for most programming purposes. The language specification defines the semantics for floating point arithmetic. This ensures that the results obtained are predictable and the code is portable across all platforms. However, a straightforward implementation of these language features can cause Java to be significantly less efficient than FORTRAN or native C/C++ implementations. Since performance is critical in the domain of numerical computing, the use of Java for numerical programs can be limited.

In this thesis, we present a technique to make numerical Java programs run faster. We capitalize on the observation that most programs spend 80% or more of dynamic execution time on 20% or less of the static code. In numerical programs, most of this static code comprises of a set of basic computational operations that occur frequently. We analyse programs to detect these basic operations, and invoke native code for executing them. Using this technique, our performance overheads are comparable to that of a just-in-time compiler.

Acknowledgments

I would like to thank my advisor, Professor David Padua, for his guidance and insightful comments. I would also like to thank members of the Polaris research group for their help and patience. Special thanks to Peng Wu . . . it was a pleasure to work with her.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work aims to improve the performance of the Java$^{(TM)}$[1] programming language and make it feasible for numerical codes to be written in it.

In this chapter, we describe the problem and explain how Java programs are executed using the basic model specified in the Java Virtual Machine Specification [LY97]. In Chapter 2, we describe previous work that has been done to improve Java performance. In Chapter 3, we give an overview of the approach taken by us. Chapter 4 talks about our program analyses and transformations and their implementation. Chapter 5 lists the common operations in programs that are covered by our technique. Chapter 6 describes the modifications to the run-time interpretation. In Chapter 7 we present experimental results. We present our conclusions in Chapter 8. In Chapter 9, we describe future work.

1.1 Problem Statement

Java has many characteristics that make it desirable to use:

- a clean object-oriented design.

[1] Java is a trademark of Sun Microsystems, Inc.

- dynamic class loading and method invocation.
- automatic storage management.
- type safety.
- precise exception handling.
- built in support for multithreading.
- support for graphics.
- portability, security, and networking support that makes it easy to share applications over the Internet (*write-once, run-everywhere* model).

These help to make programs simple and easy to understand and debug. The language specification also defines the semantics for floating point arithmetic. This ensures that the results obtained are predictable and the code is portable across all platforms. Java's strong typing and good pointer behavior help make the analysis and optimization of Java source programs simple and more precise than languages without strong typing.

However, in general, Java programs execute slower than the corresponding programs in other high level languages like C or FORTRAN. Some reasons for this difference in performance are:

- Java type safety requires array bounds checking. To provide security and reliability, every array subscript expression is checked to see if it is within the bounds of the array. Since this is done dynamically, it incurs run-time overhead.
- During execution, each reference value must be checked to see if it is valid (non-null) before it can be dereferenced.
- Precise exception handling prevents code motion across statements that can possibly throw an exception. Two statements can be safely reordered only if they, and all the intervening statements between them, are exception free. This does not

allow us to apply optimizations that need reordering of statements at all program points (e.g. common sub-expression elimination).

- Objects of user-defined type, e.g. complex numbers, cannot be inlined in Java [For98]. Inlining is needed for efficient execution as it can avoid the overhead due to memory allocation, dereferencing, and copying of objects. Such inlining is prevented by language features that operate on the reference value itself, and not the contents of the reference. These include the $==$ operator, and *null* checks for references.

- The language is defined such that the results of all operations are predictable irrespective of the platform a program is running on. This prevents the use of special floating-point features on some architectures, e.g. the fused-multiply-add instruction of the PowerPC. This also prevents the use of math identities in compiler optimizations. For example, the expression $0 * y$ cannot be replaced by 0. This is because if y has the special value *Inf* resulting from an overflow or underflow, or *NaN* (used to denote the result of an undefined operation such as dividing zero by zero), then the result of the expression will be *NaN* and not 0. Likewise, $4 * y * 0.5$ cannot be replaced by $2 * y$ because if $4 * y$ overflows, the result will be *Inf.*

- Java supports only one-dimensional arrays. Multidimensional arrays in Java must be constructed as arrays of arrays. The shape of an array is defined by the length of each of its dimensions. The shape of a Java array can change dynamically. Figure 1.1 shows the difference between the more common rectangular 2-dimensional array and the corresponding Java array.

  Though Java arrays provide greater flexibility, they also result in a loss of performance. The potential aliasing between rows of an array of arrays forces compilers to generate additional loads and stores as discussed in [For98]. For example, in Figure 1.2, an extra load of *a[i]* is required before statement C is executed. This is because, in case *a[i]* and *a[i+1]* are aliased, the value of *a[i]* will also change at statement B. The possibility that the shape of the array may change implies that the analysis to detect redundant array bounds checks is more complicated [MMS98].

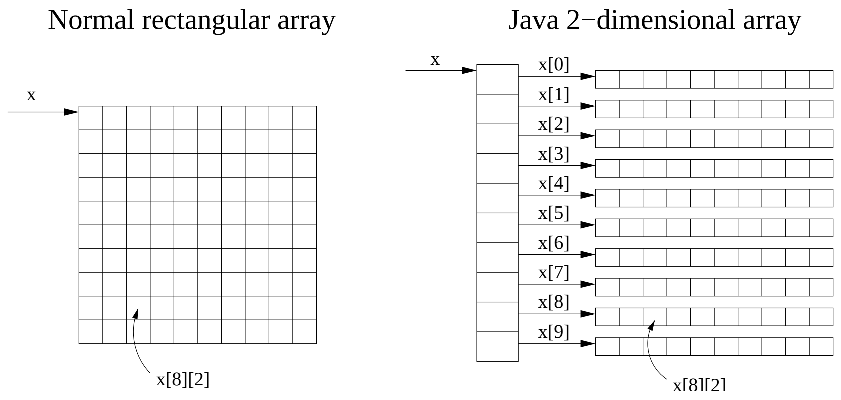


Figure 1.1: Multidimensional Arrays

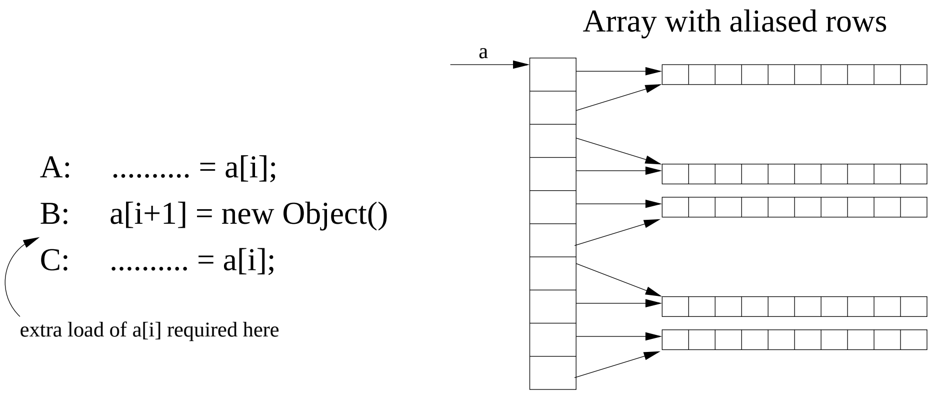


Figure 1.2: Extra Loads Needed Due to Aliasing

> The language does not define the memory layout for array objects. So we cannot guarantee that adjacent array elements are contiguous to each other in actual memory. This prevents developers from writing algorithms tuned for better performance based on the order in which array elements are processed.

Numerical programs are usually computationally intensive and take a long time to run. Hence, their performance is critical. Run-time interpretation and the overhead of dynamic checks required for type safety and security make Java programs execute slowly. We want to use Java's good programming features when writing numerical codes. This will be feasible if Java can be made to execute faster.

1.2 Typical Java Compilation and Execution

Typically, Java source code is compiled into a sequence of instructions which are written out to a *class* file corresponding to the source program file. These instructions and their semantics are defined as part of the Java Virtual Machine (JVM) Specification [LY97]. Each of these instructions is one byte long and is called a bytecode. There are approximately 200 bytecode instructions, each of which performs an operation such as a load, a store, an array access, a field access, an arithmetic operation, a logical operation, a method invocation, etc. For some of these operations there is a different bytecode corresponding to operands of different types. For example, the bytecode to load an integer value on stack - *iload*, is different from the bytecode to load a double value on the stack - *dload*.

Bytecode instructions may optionally require some arguments. These are interspersed in the instruction stream. Execution of the program proceeds by interpreting this stream of bytecode instructions. Since every instruction must be interpreted at run-time, it takes longer to execute the program.

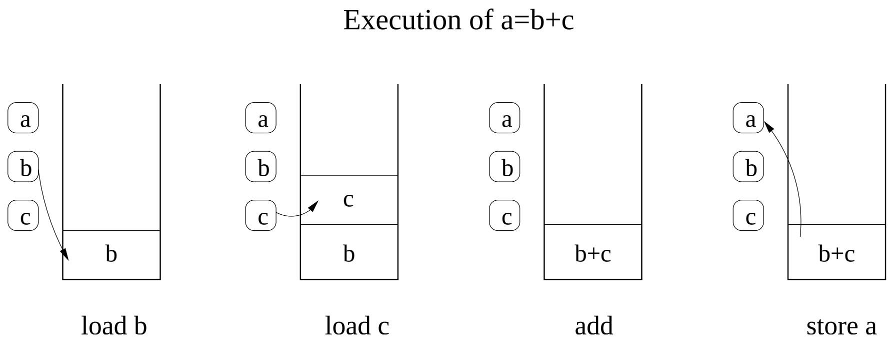


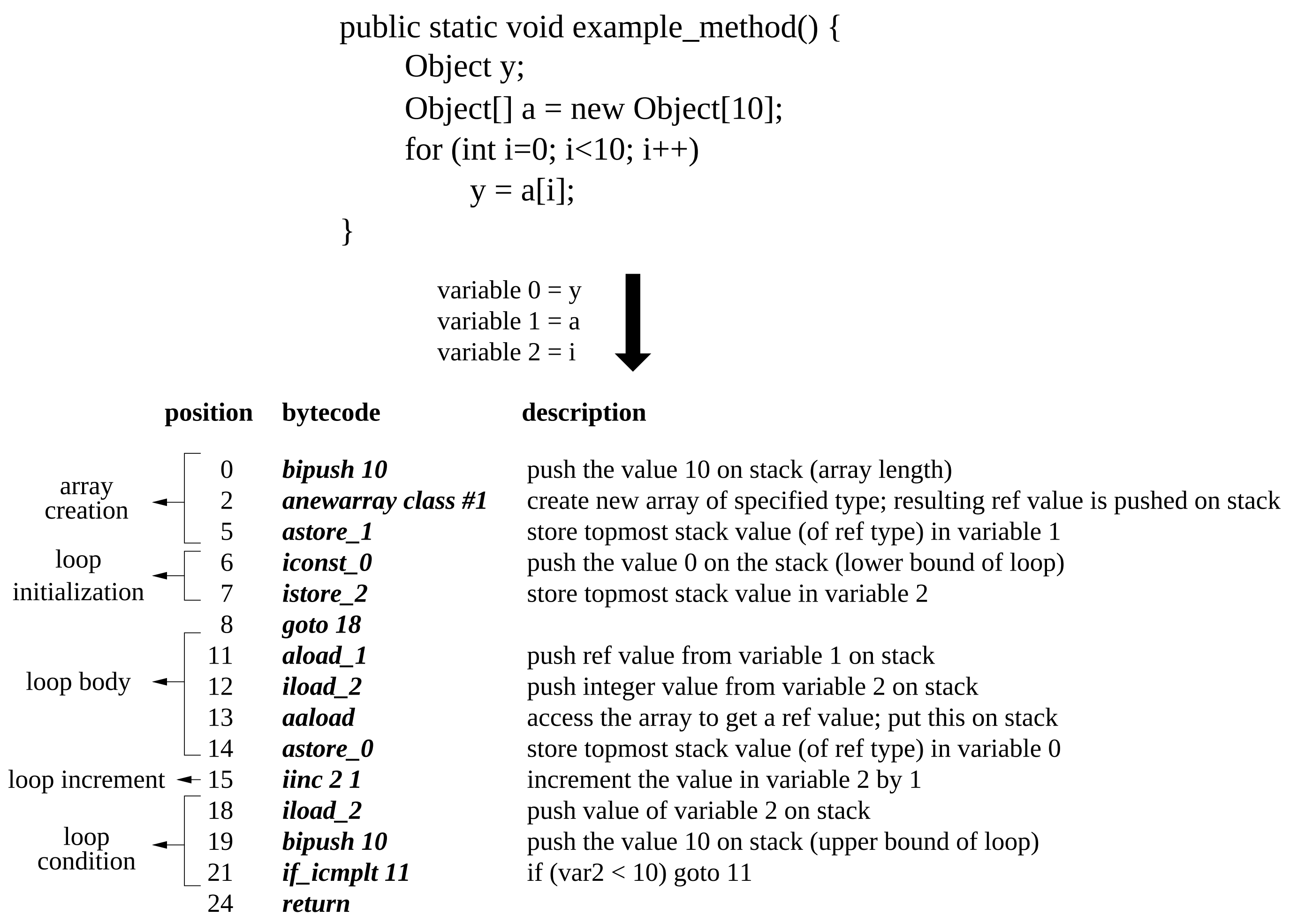
Figure 1.3: Operations on Stack

The JVM follows a stack-execution model. Operands required by an operation are pushed onto a stack. The operation is performed and operand values needed by it are popped from the stack. The result of the operation, if any, is stored on the top of the stack. Figure 1.3 shows how the stack is used for the execution of $a = b + c$.

Figure 1.4 shows an example Java loop and the corresponding bytecode instructions. Next to each bytecode instruction is a description of its purpose.

The bytecode instruction, *aaload*, at position 13 is responsible for performing the array access in the body of the loop. The reference to the array to be accessed is loaded from variable 1, and the index of the element to be accessed is loaded from variable 2 (this is done by the preceding two instructions). The *aaload* instruction implicitly checks to ensure that the array reference value is not *null*, and that the index value is within the bounds of the array.

All the bytecode instructions dealing with loads and stores of array elements implicitly perform this *null* reference check and array bounds check. In addition, *null* reference checks are performed by all instructions that need to access the object pointed to by a reference. These checks add to the execution overhead.



```
public static void example_method() {
        Object y;
        Object[] a = new Object[10];
        for (int i=0; i<10; i++)
                y = a[i];
}
```

variable 0 = y
variable 1 = a
variable 2 = i

	position	bytecode	description
	0	***bipush 10***	push the value 10 on stack (array length)
array creation	2	***anewarray class #1***	create new array of specified type; resulting ref value is pushed on stack
	5	***astore_1***	store topmost stack value (of ref type) in variable 1
loop initialization	6	***iconst_0***	push the value 0 on the stack (lower bound of loop)
	7	***istore_2***	store topmost stack value in variable 2
	8	***goto 18***	
	11	***aload_1***	push ref value from variable 1 on stack
loop body	12	***iload_2***	push integer value from variable 2 on stack
	13	***aaload***	access the array to get a ref value; put this on stack
	14	***astore_0***	store topmost stack value (of ref type) in variable 0
loop increment	15	***iinc 2 1***	increment the value in variable 2 by 1
	18	***iload_2***	push value of variable 2 on stack
loop condition	19	***bipush 10***	push the value 10 on stack (upper bound of loop)
	21	***if_icmplt 11***	if (var2 < 10) goto 11
	24	***return***	

Figure 1.4: Bytecode Example

Chapter 2

Previous Work

Java programs run slower than the corresponding programs in other languages like C or FORTRAN. However, it is possible to make them execute faster using implementation techniques that adapt to the needs of the language. Many different approaches have been explored to improve the performance of Java programs. These include improved bytecode interpretation, just-in-time compilation, bytecode optimization, static compilation, use of libraries, and changing the language specification. We review these in the following sections.

2.1 Improved Bytecode Interpretation

The JVM implementation can use machine specific optimizations and techniques like virtual inlining of method calls to improve performance. Additionally, if multiple processors are available, parallel execution can be used. For example, [YMH98] discusses how bytecodes can be restructured at run-time for speculative parallel execution. In [BVG97], a source to source compiler is used to make parallelism in the code explicit by using Java's multithreading features.

2.2 Just-In-Time (JIT) Compilation

In this approach, bytecodes are not interpreted during execution. Rather, each method is compiled into native code on-the-fly. Inexpensive optimizations are performed to make the code run as fast as possible. These optimizations include common sub-expression elimination, exception checking elimination, register allocation, class hierarchy analysis, and inlining. Several JIT compilers have been studied and implemented [CFM+97][KG97] [ATCL+98].

The amount of time spent on compiler optimizations must be low as it adds to the execution time for the program [IKY+99][CL97b]. Inter-procedural optimizations are not possible in JIT compilation. The optimization of a method can be done either at class load time or at method invocation time. In the former case, some methods which are never called are compiled as well, so it is not preferred.

JIT compilation has been enhanced using the following methods:

- Hotspot Engine [Inc99]: This is implemented by Sun Microsystems, Inc. Instead of compiling all code at run-time, the run-time optimizer works only on those parts of the code that are executed most often. Thus, more time can be spent optimizing a smaller piece of code, and greater benefits can be achieved. Feedback from a run-time profiler is used to decide what the *hotspots* are, i.e. the parts of the code that run most often dynamically, and therefore are the most beneficial to optimize.

- Inserting annotations: When compiling the Java source program to a class file, annotations can be inserted [HAKN97][ANH99]. These annotations carry information that can be used by the run-time JIT compiler to make optimization decisions. For example, static analysis can determine which program variables are most likely to be placed in registers, and encode this information in the annotations. This allows more analysis to be done offline to improve the optimizations. It also reduces the overhead of JIT compilation at run-time.

2.3 Bytecode to Bytecode Optimization

In this approach, the bytecode instructions are analysed and optimizations like dead-code elimination, strength reduction, and loop invariant detection are applied. This can be done in an inter-procedural way. The use of side-effect analysis to drive such optimizations is described in [Cla97]. Bytecode optimization is also illustrated in [CL97a].

Using this method, bytecode portability is maintained. However, there is no run-time context information available as for JIT compilation.

2.4 Static Compilation to Native Code

When Java is used in a high performance execution environment, it is better to compile the whole program into native code. This is because native code is faster. For compute intensive or high performance server applications, fast execution is more important than dynamic class loading or portability features. JVM interpretation is not used at all, and the security and portability constraints imposed by it do not interfere with program optimization.

Static compilation does time-consuming analyses offline and creates binary files. These are persistent code images that are used for run-time execution. The whole program source is made available to the compiler, and it can perform procedure inlining and other inter-procedural optimizations. However, the compiler must adhere to the Java language semantics, and allow for automatic memory management and exception handling.

Examples of static compilation systems are Caffeine [HGH96], Jalapeno [Aea00], and Quicksilver [SBMG00]. In the Quicksilver system, lightweight *stitching* is done to adapt the pre-compiled binaries to the dynamic context during execution. This process ensures that the language semantics with respect to dynamic class loading and virtual method calls are not violated.

The Java source program or the bytecodes can be used as the source for the static compiler. The former requires access to the Java source code and sacrifices portability completely. In the latter approach, analysing low-level bytecode is difficult. Usually, a higher level representation has to be extracted from the bytecode, and this step is expensive.

An alternative method is to convert the Java source code to C program code, and use existing C compilers to obtain native code. This has been implemented in Toba [PTB$^+$97] (from the University of Arizona) and Harissa [MMBC97].

2.5 Use of Libraries

The Java language specification includes a set of standard libraries that provide core functionality. The implementation of these libraries can be tuned to provide better performance as discussed in [HN00].

New libraries have been designed that implement features needed for high performance that are lacking in the Java language. The IBM array library [MMG98a] is an example of this. It implements multidimensional arrays and common operations that are performed on them. Numerical codes written using this library achieve execution speeds that are within 10% of Fortran performance.

Application libraries that address the needs of numerical computing have also been written in Java [BG97][BC98]. These are similar to the LAPACK and BLAS libraries commonly available, though they have been written using Java's object-oriented features [BDP$^+$98]. Our approach uses an application library for numerical computing combined with semantic inlining.

Semantic inlining [WMMG98] makes the compiler aware of a few special classes and methods and the semantics associated with them. It is applied at the bytecode level. The classes and methods chosen for semantic inlining are those that are crucial to performance. These are treated as language primitives. For example, complex numbers are used very often in numerical computations and so the corresponding class may be chosen for semantic inlining. Details of this can be found in [WMMG99]. The compiler can take special action when it encounters an inlined class or method. It can add annotations, generate specialized code, or insert calls to special routines that will result in faster execution.

A repository of application libraries is available on the Internet. These libraries are tuned for good performance. However, they are written in other programming languages. Work has been done to provide an interface and easy access to such libraries from the Java language [CDD97][GHM98].

2.6 Evolving the Language Specification

The Java language specification was critically reviewed in [For98] for applications that need a lot of computing resources. It identifies those aspects of the language that need to be changed to provide a better programming environment and achieve greater performance.

The specification for the Java language defines strict floating point semantics that prevent the use of special hardware on some machines [KD98]. For example, it cannot take advantage of the fused-multiply-add (FMA) instruction of the PowerPC architecture. It allows only a subset of the functionality defined in the IEEE 754 floating-point specification. For example, only one mode of rounding (*round-to-nearest*) is included.

The language was revised [Inc] and the definitions of floating-point operators changed such that, instead of exactly one correct result, a set of correct results may be returned.

This allows for the use of extended precision in intermediate expressions and the more efficient extended floating point registers in the x86 architecture. It is argued that the non-deterministic results due to this does not affect practical applications. In cases where it is imperative that the result be deterministic, a keyword *strictfp* can be used to specify that original Java floating point semantics should be followed.

There have been proposals (known as Java Specification Requests[1]) to add support for multidimensional arrays. These are commonly used in scientific applications. In order to process array elements in the most efficient way, the developer must know the layout of the array in memory. For example, adjacent elements of the row of an array can be defined to be in physically contiguous memory locations. In this case, an algorithm that accesses the array in row-major order is more efficient than the corresponding algorithm that accesses the array in column-major order. However, under the current scheme of automatic transparent memory management, the memory layout is not known to the user.

There is a proposal to allow classes to be declared as immutable. Objects of these classes can be passed-by-value. They can be treated as lightweight objects and can be inlined for better performance. Operator overloading and parametric polymorphism are two other additions that are proposed. These do not directly improve performance, but make the interface to libraries simpler and more natural. This makes Java programs easier to write and maintain.

In addition to these techniques, there are experimental implementations of variations of the Java language. For example, Titanium [YSP+98] uses the Java language as a base and adds features needed for high performance, such as multidimensional arrays, immutable classes, explicit parallelism with a global address space, etc.

[1]A list of all the Java Specification Requests (JSRs) can be found in [jsr]

Chapter 3

Overview

In this chapter, we describe our approach to improve the performance of Java numerical programs.

We observe that there is a small set of basic operations that frequently occur in numerical algorithms, such as vector dot-product, matrix row transformations, and initialization of array elements. A collection of such operations can be pre-compiled into a library of native code that is integrated with the interpreter. These operations are mostly loops that manipulate vectors or matrices, and in many cases a significant fraction of the computation of the program is covered by them. Moreover, the codes usually do not have invalid reference accesses or out-of-bounds array accesses. This is important because illegal program accesses can raise exceptions. These exceptions will inhibit code transformation due to the strict exception semantics of Java (described in section 4.3.1).

The basic operations can be statically recognized by the compiler. To do this correctly, the compiler must ensure that there are no side-effects due to aliasing between reference values. Loop distribution must also be performed to uncover as many instances of such operations as possible, and ensure that a greater coverage is achieved. When code is reordered for loop distribution, it is important to ensure that precise Java exception

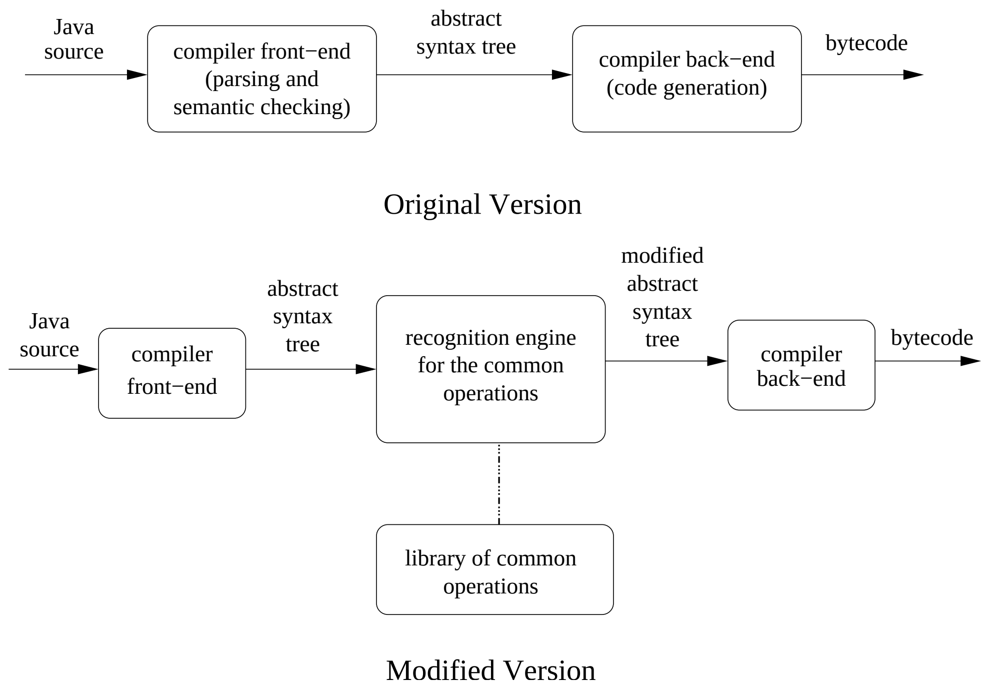

Java source
compiler front−end (parsing and semantic checking)
abstract syntax tree
compiler back−end (code generation)
bytecode
Original Version
Java source
compiler front−end
abstract syntax tree
recognition engine for the common operations
modified abstract syntax tree
compiler back−end
bytecode
library of common operations
Modified Version

Figure 3.1: Modified Compiler

semantics are not violated. For this, the code has to be analysed to determine the exceptions that can be thrown.

Figure 3.1 shows the modifications required in the compiler. After parsing and semantic checking, the analyses required to recognize the common basic operations are performed. These are clubbed together in the box labeled '*recognition engine for the common operations*'. Finally, code generation is done, and the bytecode instructions are written to the output *class* file.

The compiler can insert special methods for the common basic operations so that later, at run-time, the interpreter can identify them as special cases. The interpreter calls the corresponding native code from the library with appropriate parameters. The modified bytecode sequence, shown in Figure 3.2, has special calls inserted for the basic operations recognized. The bytecode sequence that is functionally equivalent to the inserted method call is retained as part of an exception handler. This is needed for compatibility across all interpreters as discussed in section 6.2.

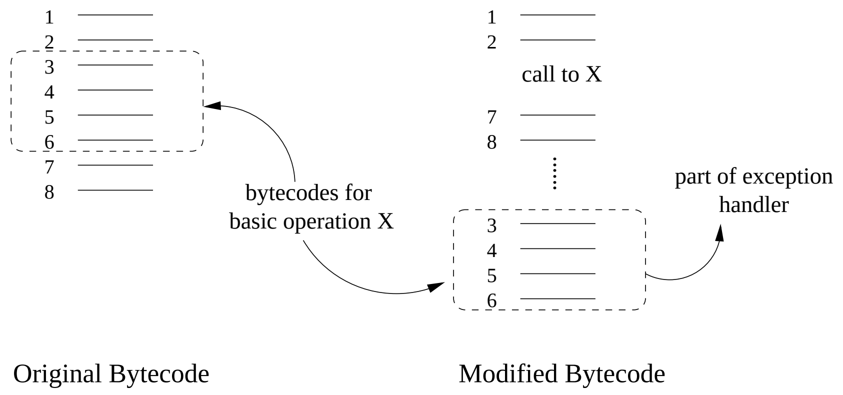
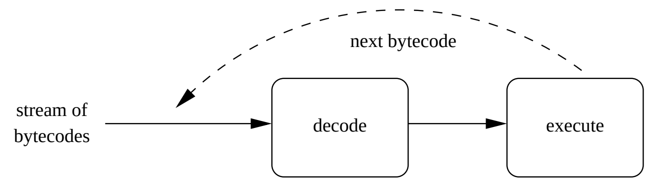


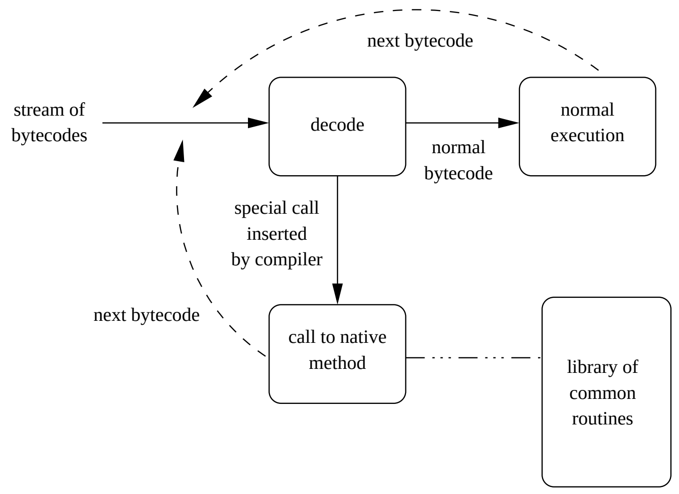
Figure 3.2: Modified Bytecode Sequence

Figure 3.3(a) shows the normal interpretation of a bytecode sequence without any of our modifications. Figure 3.3(b) shows the changes introduced when the interpreter is modified to incorporate our native library of common basic operations. When a bytecode instruction is decoded to be a special one that is inserted due to the modifications in the compiler, the corresponding native method is called before the next bytecode instruction is interpreted. For other bytecode instructions, the interpretation is done exactly as before. Figure 3.3(c) shows how the bytecode stream modified by our compiler is executed by an interpreter that is not aware of the native library. When this interpreter encounters a special call to a basic operation, it will not recognize it and an exception will be thrown. The modified code defines the exception handler that will catch this exception and execute a bytecode sequence that corresponds to the basic operation for which the special call was inserted. The exception handler is defined in the modified Java source code by a catch statement, and its scope is restricted to exceptions thrown by statements in the corresponding try-block. This try-block will contain the special native method call only.

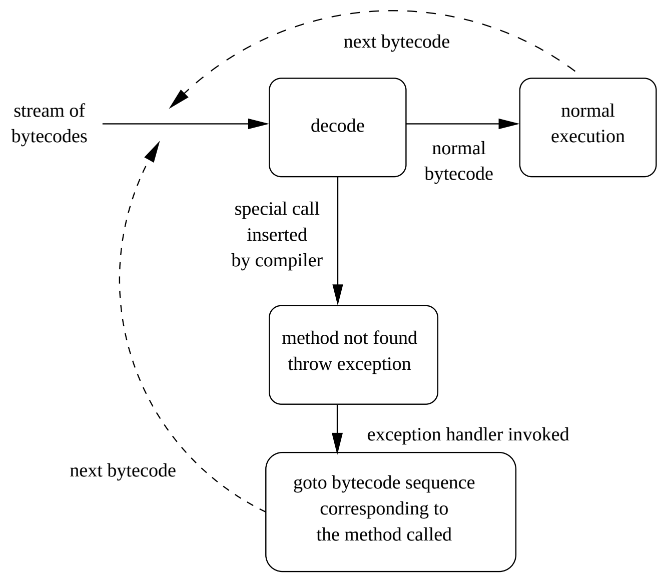
Thus, many parts of the Java program can be executed using native code, while Java semantics are still adhered to. Since native code is the way to achieve fastest execution, this improves performance as desired.



(a) Original Interpreter With Unmodified Code



(b) Modified Interpreter With Modified Code


next bytecode
stream of bytecodes
decode
normal execution
normal bytecode
special call inserted by compiler
method not found throw exception
exception handler invoked
next bytecode
goto bytecode sequence corresponding to the method called

(c) Original Interpreter With Modified Code

Figure 3.3: Modified Interpreter

Chapter 4

Program Analysis

Our technique to improve performance makes use of native code rather than interpretation for the execution of commonly-used basic operations. The first step needed to apply this technique is to recognize the maximum number of these basic operations in the source program. This is done statically by the compiler.

The source program must be analysed and transformed for us to be able to recognize as many instances of the basic operations as possible. There are four main steps involved in this transformation process: data dependence analysis, reference analysis, exception checking, and loop distribution. The following sections describe each of these steps and their implementation. The last section talks about recognizing the common basic operations, and transforming the program to insert special calls for them. These special calls are later used by the run-time virtual machine to enable faster execution.

4.1 Data Dependence Analysis

Dependence analysis is needed to determine if the order of execution of statements can be changed without changing the flow of values in the program. It is used during loop distribution. Since we use the results of the analysis for loop transformation, we are concerned with finding dependences within loop nests only.

4.1.1 Program Loops

For the analysis, we consider only those program loops that are in the normalized form. A normalized loop, like a FORTRAN do-loop, has an associated **loop variable**, a **lower bound**, a loop invariant **upper bound**, and an **increment**. The lower bound is fixed (has a value of zero), and the increment is always unity. The bounds may be specified as absolute or symbolic values. The loop variable serves as a counter for the iterations of the loop. The lower bound is the initial value for this counter, the upper bound is its final value before the execution of the loop is complete, and the increment is the value that is added to the counter at the end of each iteration.

All program loops that cannot be converted to the normalized form must be treated conservatively, and loop distribution cannot be applied to them. This is because the implementations of alias analysis and dependence analysis handle such loops very conservatively.

4.1.1.1 Loop Nests

A loop nest is a sequence of loops contained within each other. The **nesting depth** gives the number of loops in the sequence. The **loop level** gives the position of a loop in a loop nest, with the outermost loop at level one. The loop iteration vector can be used to uniquely identify any iteration of a loop within a loop nest.

Definition 4.1.1: For a given iteration of a loop nest with nesting depth n, the loop **iteration vector** is an n-tuple, such that the i-th element of the tuple is the value of the loop variable for the loop at level i in this iteration.

Figure 4.1 shows a loop nest with a depth of 3, and the valid iteration vectors for it.

loop level	***loop nest***	***valid iteration vectors (i, j, k)***		
1	`for (int i=0; i<=1; i++)`	(0, 0, 0)	(0, 0, 1)	(0, 1, 0)
2	`    for (int j=0; j<=1; j++)`	(0, 1, 1)	(1, 0, 0)	(1, 0, 1)
3	`        for (int k=0; k<=1; k++)`	(1, 1, 0)	(1, 1, 1)	
	`            ......`			

Figure 4.1: An Example Loop Nest

4.1.2 Statement Instances

Statements contained in a loop are executed once for every iteration of the loop.

Definition 4.1.2: Given a statement S that is contained within a loop, and an iteration vector x, a **statement instance** *S(x)* is the execution of the statement S for the loop iteration identified by x.

4.1.3 Execution Order

For statements in a program that are not within the same loop nest, the lexicographic order defines their execution order. For statement instances within the same loop nest, relations between loop iteration vectors are also used to determine the execution order.

Definition 4.1.3: For two iteration vectors, $(v_1, v_2, \ldots, v_n)$ and $(w_1, w_2, \ldots, w_n)$, $(v_1, v_2, \ldots, v_n) = (w_1, w_2, \ldots, w_n)$ iff $v_j = w_j$ for all j such that $1 <= j <= n$.

Definition 4.1.4: For two iteration vectors, $(v_1, v_2, \ldots, v_n)$ and $(w_1, w_2, \ldots, w_m)$ where $n <= m$, $(v_1, v_2, \ldots, v_n) < (w_1, w_2, \ldots, w_m)$ iff there exists i such that $1 <= i <= n$ and $v_i < w_i$, and $v_j = w_j$ for all j such that $1 <= j < i$.

Given two statement instances *S1(x)* and *S2(y)* which are in the same loop nest, and x and y correspond to the iteration vectors for loops that are common between statements *S1* and *S2*, *S1(x)* precedes *S2(y)* in the execution order iff:

$x < y$, or

$x = y$ and *S1* comes before *S2* in the lexicographic order.

4.1.4 What is a Dependence

Data dependences are caused by accesses to the same memory location at different points in the program. An access to a memory location may either read the contents of the memory location or write a value to the memory location[1].

Definition 4.1.5: For a statement S, **USE**(S) is the set of all memory locations that may be read during the execution of that statement.

Definition 4.1.6: For a statement S, **DEF**(S) is the set of all memory locations that may be written to during the execution of that statement.

There are three kinds of data dependences:

1. **Flow dependence**: Occurs from a write access to a subsequent read access. It indicates that the value read by the second access must be the same as the value that was written by the first access. A flow dependence will exist from statement *S1* to *S2* if $DEF(S1) \cap USE(S2) \neq \Phi$.

2. **Anti dependence**: Occurs from a read access to a subsequent write access. It indicates that the second access must not overwrite the original value until it has been read by the first access. An anti dependence will exist from statement *S1* to *S2* if $USE(S1) \cap DEF(S2) \neq \Phi$.

3. **Output dependence**: Occurs from a write access to a subsequent write access. It indicates that the value contained in the memory location after both accesses must

[1]Update accesses which both read and write a memory location must be counted as both a read access and a write access.

be the one that was written by the second access. An output dependence will exist from statement *S1* to *S2* if $DEF(S1) \cap DEF(S2) \neq \Phi$.

A data dependence exists from statement instance *S1* to statement instance *S2* if all of the following conditions hold:

1. *S1* comes before *S2* in the program execution order.
2. There is some common element, say x, between the sets USE(*S1*) and DEF(*S2*) (*anti dependence*), or between the sets DEF(*S1*) and USE(*S2*) (*flow dependence*), or between the sets DEF(*S1*) and DEF(*S2*) (*output dependence*), i.e.
$$(USE(S1) \cap DEF(S2)) \cup (DEF(S1) \cap USE(S2)) \cup (DEF(S1) \cap DEF(S2)) \neq \Phi \quad .$$
3. There is no statement instance *S3* such that *S3* comes between *S1* and *S2* in the program execution order, and DEF(*S3*)=$\{x\}$.

Program control flow is also used to determine if a dependence may exist between two accesses. There is no dependence between an access in the if-branch and an access in the else-branch of the same conditional statement.

4.1.5 Direction Vectors

When the dependence is between statement instances contained in a loop nest, it can be represented as a direction vector.

Definition 4.1.7: Given a dependence from statement instance *S1(x)* to statement instance *S2(y)*, where $x=(x_1, x_2, \ldots, x_n)$ and $y=(y_1, y_2, \ldots, y_m)$, the **direction vector**, v, for this dependence is a p-tuple, where p is the minimum of n and m. Each element v_i, for $1 <= i <= p$, is one of the following:

- $<$, if $x_i < y_i$.
- $=$, if $x_i = y_i$.
- $>$, if $x_i > y_i$.

For example, consider the following doubly nested loop:

```
for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
        1: a[i][j] = ...;
        2: a[i-1][j] = ...;
    }
}
```

For the output dependence from statement 1 to statement 2, the direction vector is $(<, =)$. This tells us between which instances of the two statements the dependence is valid. The dependence exists only when the i value of the instance of statement 1 is less than the i value of the instance of statement 2, and the j values of the two instances are equal.

Definition 4.1.8: A dependence between two statement instances which are within the same loop nest is **loop-carried** if the iteration vectors corresponding to the two statement instances are not equal. The dependence is said to be carried at level c, where c is the lowest loop level for which the corresponding element of the direction vector tuple is not $=$.

Thus, a loop-carried dependence can exist only between different iterations of a loop.

Definition 4.1.9: A dependence between two statement instances *S1(x)* and *S2(y)*, which are within the same loop nest, is **loop-independent** if $x=y$.

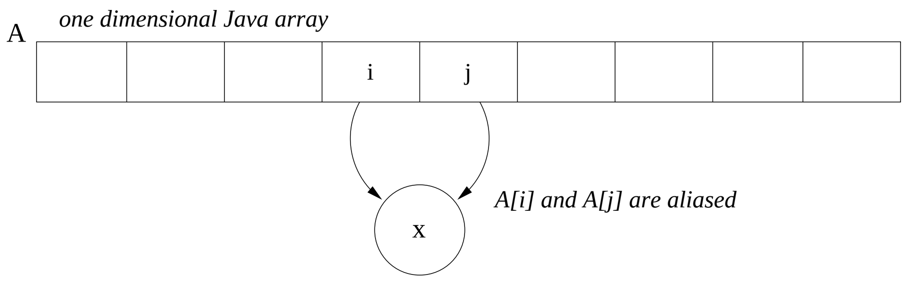


Figure 4.2: Distinct Locations of Java 1-D Array Elements

4.1.6 Array Accesses

We have discussed data dependences in terms of memory locations. These locations are specified in a program using variable names, optionally combined with some field access operations and array subscript expressions.

According to Java language semantics, different elements within the same array must be distinct memory locations when the array is one-dimensional. Two accesses to the same array, *A[i]* and *A[j]*, will access different locations if i and j have different values. This is illustrated in Figure 4.2 where *A[i]* and *A[j]* have the same values, but different locations. Thus, array subscript expressions can be used to disambiguate memory locations in dependence analysis.

4.1.7 Dependence Tests

A dependence test checks to see if a given pair of program accesses can refer to the same memory location. There are several different tests to do this, such as the GCD test, Banerjee's test, and the Omega test. We discuss each of these tests in the following sections. In our implementation, we use the Omega test.

4.1.7.1 The GCD Test

Consider the following loop:

```
for (int i=0; i<n; i++) {
    ... a[$x_1 * i + x_0$] ...
    ... a[$y_1 * i + y_0$] ...
}
```

The two subscripts will have the same value when there is an integer solution for the equation:

$$x_1 * i_1 - y_1 * i_2 = x_0 - y_0$$

In the above equation, i_1 represents the value of the loop variable in the iteration in which the first access is made, and i_2 represents the value of the loop variable in the iteration in which the second access is made. If there is no integer solution for the equation, then there will be no dependence between the two accesses.

The GCD test checks for the possibility of a dependence using the property that a solution will exist if and only if the greatest common denominator of x_1 and y_1 divides $(x_0 - y_0)$.

4.1.7.2 Banerjee's Test

The GCD test does not take into account the bounds on the values of the loop variable. Banerjee's test does this.

It finds the upper bound (U) and the lower bound (L) for the expression $x_1 * i_1 - y_1 * i_2$ under the constraints that i_1 and i_2 should be values within the bounds of the loop. Then, if $U < (x_0 - y_0)$ or $L > (x_0 - y_0)$, the two subscript expressions cannot have the same value, and no dependence exists.

Banerjee's test gives an inexact result. The result can conservatively indicate that a dependence might exist in some cases where there is no dependence in fact. This is because:

- The test checks for real solutions, whereas loop variables take integer values only.
- When an access has multiple subscripts, the test checks for a solution for each pair of subscripts individually. However, the intersection of all the individual solutions might be empty, which will mean that no dependence exists.

4.1.7.3 Omega Test

Unlike the previous two tests, the Omega test [Pug91] provides an exact solution.

The conditions for a dependence to exist between two program accesses are expressed in terms of constraints over integer tuple relations and sets. This includes specifying the loop variables for the loops enclosing the accesses, the bounds of these loop variables, and the coefficients of these loop variables in subscript expressions (if the accesses are elements of an array). These constraints can be specified using relational and logic operators, free and bound variables, numbers, and Boolean values. The system of constraints is solved to determine if a dependence exists.

The Omega test can be implemented using the Omega library [KMP+96]. Using this test, direction vectors can be obtained for a dependence that exists between two accesses in a loop nest. The implementation can handle only those array subscripts that are affine expressions of the loop variables, i.e. linear expressions of the form $a_0 + a_1 * x_1 + a_2 * x_2 + ... + a_n * x_n$, where $a_0 ... a_n$ are constants and $x_1 ... x_n$ are loop variables. A conservative result is returned for other cases.

4.2 Reference Analysis

Reference analysis is needed to disambiguate the reference accesses in a program. In fact, it can be regarded as a form of dependence analysis that specifically works with program accesses that hold reference values. Reference analysis also helps to detect exception-free loops.

Definition 4.2.1: A program **reference access** can be of the form p, $p[\dots]$ (i.e. an array access), or $p.e$ (i.e. a field access), where p is either a stack variable containing a reference value or another reference access.

Definition 4.2.2: A **reference assignment statement** is an assignment statement whose left hand side is assigned a reference value.

In this analysis, we map program reference accesses to the heap objects they refer to. We track these mappings in a flow sensitive manner, i.e. we keep a separate set of mappings for each reference assignment statement in the program. Thus we can determine when two program accesses refer to the same memory location, and dependences between program accesses can be tracked.

4.2.1 Basic Concepts

References in Java can only denote objects allocated in the heap by instances of *new* statements in the program. Invalid references have the special value *null*. An access will refer to some heap object, and the object that it refers to can change dynamically as the program executes.

Ideally, each program access should map to a single heap object, but this is not possible because of conditional assignments to variables in the program. Also, deeply nested aggregate data structures increase the amount of information that needs to be tracked beyond what is feasible in practice. Therefore, some approximations are used, and mappings are maintained from program accesses to a set of heap objects.

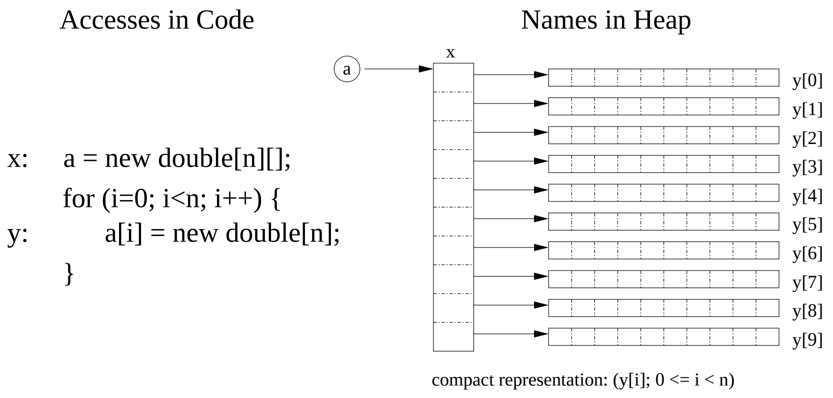


Figure 4.3: Mapping Program Accesses to Heap Names

Reference analysis is implemented using the algorithm described in [WFPS00]. This technique uniquely identifies each heap object by the instance of the *new* statement that allocated it. Statement instances are named using the position at which they occur in the program source and the loop iteration vector (if the statement is within a loop nest). For handling *new* statements within while-loops and do-while loops, a dummy loop variable is introduced. Thus, every object has a unique *heap name*.

The algorithm does an element-wise analysis, i.e. each array element has its own mapping to a set of heap objects. To enable a compact representation, mappings for array elements are in terms of functions over the array indexes. Figure 4.3 shows an example where statement instances x and *y(i)* are used to name the heap objects allocated by them. Precise information for array elements is important in the case of Java. This is because Java multidimensional arrays are arrays of arrays and one dimensional array objects are reference values that can be manipulated just like other reference values. This is of greater significance for numerical programs which typically involve many array operations.

4.2.2 The Algorithm

The reference analysis is done by an iterative algorithm that traverses the program control flow graph. It invokes a transfer function for each reference assignment statement. These transfer functions update the set of mappings from program accesses to heap object names based on the effect of the corresponding statement. This update affects the mappings specific to a particular statement point in the program. Statements in loops must be iterated over, and the set of mappings is updated each time until a fixed point solution is reached. At each program point where two execution paths meet (end of conditional statements and loop statements), the union of the results obtained on each path is taken.

4.2.3 Transfer Functions

The algorithm in [WFPS00] defines a transfer function for each reference assignment statement. All reference assignment statements can be expressed in terms of seven primary ones. The transfer functions for these seven statements are described below. p and q are stack variables of reference type. a is a field name for the Java object that is referenced, and e is the array subscript expression.

1. p = *null*:

   The set of heap objects that p can refer to is set to be the *null* object only.

2. p = *new javaObject()*:

   The set of heap objects that p can refer to is updated to be the single object that is allocated by this statement instance.

3. $p = q$:

   The set of heap objects that p can refer to is the same set that q can refer to.

4. $p = q.a$:

   The set of heap objects that p can refer to is the same set that $q.a$ can refer to. Since the mappings for $q.a$ are approximated by the mappings for q, the new mapping for p is the same as that for q. Using the type information for p, this set can be pruned to exclude all those heap objects that are not of compatible type.

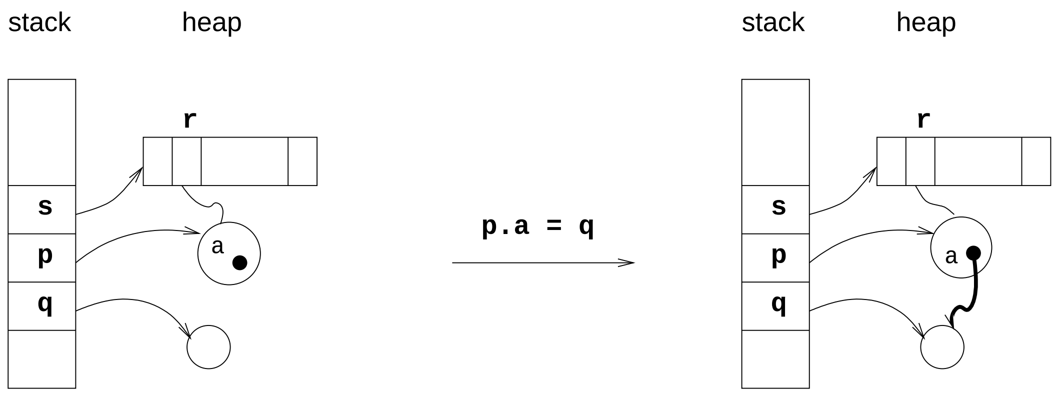


Figure 4.4: Effect of $p.a = q$ on Memory Locations

5. *p = q[e]*:

   The set of heap objects that p can refer to is the same set that *q[e]* can refer to. Also, if p is of array type, then *p[x]* will map to the same set as *q[e][x]*, *p[x][y]* will map to the same set as *q[e][x][y]*, and so on, up to the last dimension of p.

6. $p.a = q$:

   Figure 4.4 shows how references to memory locations are modified by the execution of this statement.

   The access $p.a$ will be mapped to a set of heap objects that can be referred to by it, say the set **P** of heap names. Similarly, the access q will be mapped to a set of heap objects, say **Q**. The assignment causes some heap object in **P** to refer to some heap object in **Q**. To be conservative, we must assume that any member of **P** can now refer to any member of **Q**.

   This assignment can reflect on program accesses other than $p.a$. This is because there can be another program access r that refers to one of the heap objects in **P**. Since the objects referred to by members of **P** has now changed, the set of objects referred to by r should also change accordingly (i.e. **Q** should be added to the set of mappings for r).

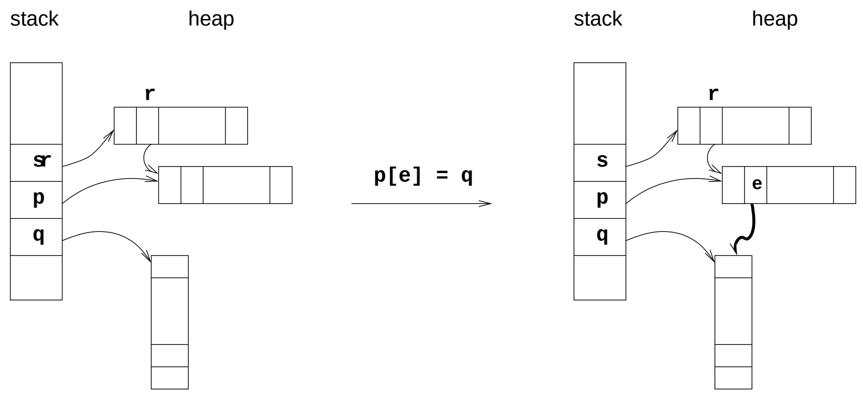


Figure 4.5: Effect of *p[e]* $=q$ on Memory Locations

Thus, the transfer function for the statement $p.a = q$ does the following:
For each program access r whose mapping includes some subset of **P**, update the mapping of r to include the set of heap objects reachable from **Q**.

7. *p[e]* $= q$:

   Figure 4.5 shows how references to memory locations are modified by the execution of this statement.

   In this case, p is of array type, and the access *p[e]* is to an array element. The set **P** of heap objects that p can refer to must also be of array type. The location assigned to by the statement is at an offset e inside some heap object in **P**. This location can be reached by any other access r, only if that access has some subset of **P** in its mapping (i.e. in the set of heap objects it can refer to).

   Case 1: The access r is of array type.
   Updating the mapping of an array access affects the mappings for elements at deeper levels of the array as well (i.e., what *r[i][j]* refers to depends on what *r[i]* refers to). Thus, if the program access r is of array type, *r[e]* can refer to what q refers to, and its mapping should include **Q** (the mapping for q). Also, *r[e][x]* can refer to what *q[x]* refers to, *r[e][x][y]* can refer to what *q[x][y]* refers to, and so on, up to the highest dimension. Therefore, we add the mappings for elements of q to

the mappings for the corresponding elements of *r[e]* [2]. Java's strong typing ensures that the dimension of r is one more than q.

Case 2: The access r is not of array type.

The mappings should be updated as in the case of $p.a = q$, i.e. we add the set of heap objects reachable from **Q** to the mapping for r.

4.3 Exception Checking

Checking for exceptions is important when we try to reorder statements for loop distribution. This is because code motion is restricted across statements which can throw exceptions.

4.3.1 Java Exception Semantics

Precise exception semantics defined for the Java language require that the user-visible state of a program, on termination due to an exception, should be exactly the same as if all the statements before the exception point (and only those statements) have been executed. This implies that if there is a possibility for a statement to throw an exception, there must be no code motion across the boundary of that statement.

Many common operations in numerical programs involve array accesses. The bytecode instruction that performs an array access implicitly checks for the following two exceptions:

- *NullPointerException*: This is thrown when the reference value for the array accessed is *null*.

[2]Since mappings take subscripts as parameters, constraints are used to specify that the heap objects added are valid only when the first subscript of the access to r is e.

- *ArrayIndexOutOfBoundsException*: This is thrown when the subscript expression for the array element accessed evaluates to a value that is beyond the bounds of the array.

Additionally, there are other run-time exceptions that are implicitly checked by byte-code instructions. Examples of these are *ArithmeticException* (thrown, for example, when division by zero is attempted) and *ClassCastException* (thrown when trying to cast a reference to an incompatible type)[3]. Beside run-time exceptions, there are other user-defined exceptions which are explicitly thrown and checked in the Java source program.

If we want to change the relative order of any two program statements, we must ensure that the execution of these two statements and all the statements between them will never throw an exception. Thus, static analysis is done to detect redundant exceptions and determine which statements are exception free. Since *NullPointerExceptions* and *ArrayIndexOutOfBoundsExceptions* are most common in numerical programs, we concentrate on them in our analysis. Any statement that can throw an exception other than these two exceptions will not be exception-free by default.

4.3.2 Checking For *NullPointerExceptions*

We can determine if a reference access may have the *null* value or not. This is done by extending the reference analysis.

The original reference analysis uses a simplistic approach and each mapping from a program reference access to a set of heap names includes *null* by default. Another analysis phase is added to remove redundant *nulls* from these mappings. A single sequential scan of the program is done, traversing it statement by statement in lexical order. The reference mappings at each statement point in the program are modified based on the

[3]Refer to [LY97] for a complete list of exceptions.

effect of the statement and the values of the mappings at the immediately preceding point in the program. For each reference assignment statement:

- If the left hand side maps to a single heap object, and the right hand side is a *new* statement, or the mapping for the right hand side at the immediately preceding program point does not include *null* in its set of heap objects, then the *null* value is removed from the left hand side heap object for this program point. For example, consider the assignment $a = b$. Let A be the set of heap objects corresponding to a and B be the set of heap objects corresponding to b at the program point immediately preceding the assignment statement. Then, for this case to apply, A must be a singleton set and B must not include *null*. As a result, A is modified so that it does not include *null* for the program point after the assignment statement.
- If the mapping for the right hand side at the immediately preceding program point includes *null* in its set of heap objects, then *null* is included in the mapping for the left hand side for this program point. In the previous example, if set B includes *null*, then *null* is added to the set A for the program point after the assignment statement.

If each element in the set of mapped heap object names for a given program access is marked as non-null, then a *NullPointerException* cannot occur when accessing a field or element of that access. The exception is therefore redundant, and need not be checked during program execution.

4.3.3 Checking For *ArrayIndexOutOfBoundsException*

In reference analysis, each heap object is given a name when it is allocated by an instance of a *new* statement. For heap objects that correspond to arrays, we store the length of the array as part of the heap name for that object. This length can be obtained from the arguments to the *new* statement.

Thus, for an array access in the program, we can get its length information using the mapping from program accesses to heap object names. The array length and the subscript expression for the access can then be compared to determine if an *ArrayIndex-OutOfBoundsException* can be thrown.

The subscript expression and the array bound must be either constants or symbolic values that are statically comparable. Otherwise, the exception will not be redundant and it must be checked during execution.

More detailed analyses for checking redundant *ArrayIndexOutOfBoundsExceptions* is presented in [BGS00] and [MMS98]. These are not implemented in our system, but can be incorporated to obtain better precision for the detection of redundant exceptions.

4.4 Loop Distribution

Loop distribution splits a loop with many statements into two or more loops with fewer statements each. We want to apply this transformation because the basic operations in our pre-compiled library are all of the simplest possible form. Smaller loops give us the opportunity to recognize more basic operations that might be embedded in the source code, thus resulting in greater coverage for the optimization.

Also, loop distribution is the main transformation needed for vectorization [ZC91, Chapter 6]. When running programs on a vector machine, performance is improved using vectorization.

After basic operations are recognized, a loop fusion transformation can be added[4]. Loop fusion combines two or more program loops into a single one wherever possible without violating data dependences. This is important because having more loops than necessary degrades performance due to the loop overheads of testing the loop condition, branching back, and incrementing the loop variable.

[4]This is not implemented in our system.

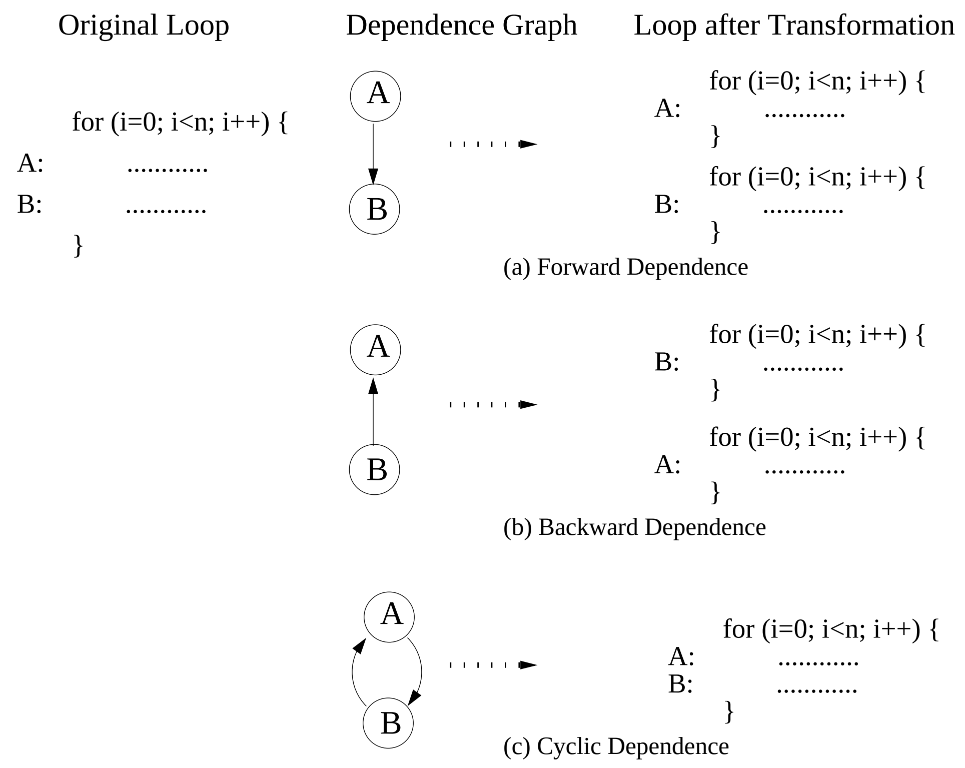


Figure 4.6: Loop Distribution

4.4.1 Concept

Statements in a loop can be split into two different loop partitions if no dependences are violated and if no instance of a statement in the first loop partition depends on any statement in the second loop partition.

For example, consider the loop shown in Figure 4.6, with two statements A and B, where A lexically precedes B. If there is a dependence from A to B (forward dependence), the loop can be split as shown in Figure 4.6(a). If there is a dependence from B to A (backward dependence), the loop can still be safely split as shown in Figure 4.6(b). However, if there is a dependence from A to B and from B to A, i.e. there is a dependence cycle as in Figure 4.6(c), then the loop cannot be split without violating dependences.

In general, all statements involved in a dependence cycle must be in the same loop, and statements not in a common dependence cycle with one another can be split into separate loops.

4.4.2 Distribution Using Dependence Graphs

Loop distribution can be done by building a dependence graph for all the statements contained within the loop.

Definition 4.4.1: A **dependence graph** is a directed graph where each node represents a program statement. There is an edge from one node to another if a dependence exists from the statement corresponding to the first node to the statement corresponding to the second node.

For loop distribution, we must find all the strongly connected components in the dependence graph.

Definition 4.4.2: A **strongly connected component (SCC)** of a graph is a maximal subset of the nodes in the graph such that for any pair of nodes A and B from this subset, there is a path from A to B and a path from B to A.

Once we have identified the strongly connected components, we can get the acyclic condensation of the dependence graph [ZC91, pages 353-355].

Definition 4.4.3: The **acyclic condensation** of a graph is an acyclic graph obtained by replacing each strongly connected component of the original graph by a single node. There will be an edge from node A to node B in the condensed graph if the original graph had an edge from any of the nodes in the SCC corresponding to node A to any of the nodes in the SCC corresponding to node B.

The loop can be split into as many parts as there are nodes in the acyclic condensed graph. Each of the loop partitions contains the statements that correspond to the nodes in an SCC of the original graph. The order in which these partitions appear in the program is determined by a topological sort of the acyclic condensed graph. This ensures that no dependences are violated.

When there is a loop nest and an inner loop is considered for distribution, dependences that are loop carried (Definition 4.1.8) on the outer loops only (i.e. all the loops enclosing the one that is to be distributed) are not to be taken into account. A loop carried dependence exists only between different iterations of the loop it is carried on. Splitting an inner loop will not affect the manner of execution of the outer loops, so it is safe to ignore dependences between different iterations of the outer loops.

4.4.3 Distributing Loops With Conditional Statements

Conditional statements have an associated conditional expression which determines the flow of control, and a body of statements whose execution depends on the value of the conditional expression. This dependency is called a **control dependency**. Conditional statements in a loop can also be split between different loop partitions to obtain a finer granularity of distribution. The algorithm described in [KM90] can be used to do this. It replaces control dependences between partitions of a distributed loop with data dependences.

The method works as follows:

1. Initially, the loop is distributed disregarding the control dependences that exist due to conditional if-then-else statements. Each conditional expression is also considered to be a statement node in the dependence graph built for distribution.

2. After the loop statements are partitioned, all the conditional statements are checked to determine which ones have been split up between two or more partitions of the distributed loop.

3. For each conditional statement identified in the previous step, a new *execution variable* is introduced for its conditional expression. An execution variable is an

array variable with as many elements as there are loop iterations. Each element will record the value that the conditional expression evaluates to (i.e. true or false) in the corresponding iteration.

4. All execution variables are initialized to a special *undefined* value at the start of the loop partition that contains the corresponding conditional expression. The *undefined* value indicates that the conditional statement has not been encountered on the program execution path.

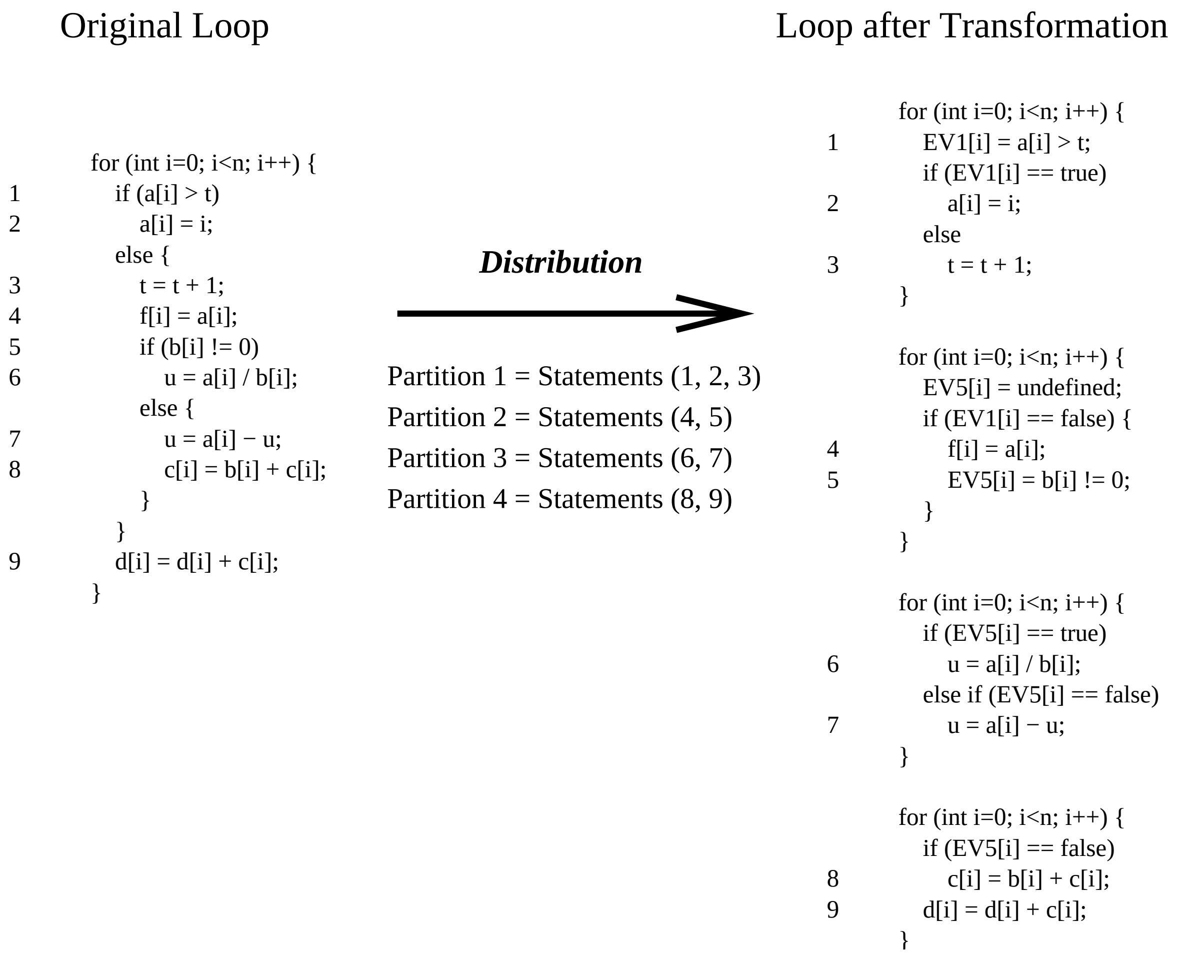
5. The execution variables are assigned values at those points where the corresponding conditional expressions are placed (note that these were also considered to be separate statements when performing loop distribution). This initialization of the execution variable replaces the conditional expression in the transformed code. If the assignment of the *undefined* value that was done earlier immediately precedes this initialization, then that assignment is redundant and can be omitted.

6. Guards are introduced for the execution of all statements that were originally contained within one of the conditional statements identified in step 2. The guard expressions are defined in terms of the execution variable corresponding to that conditional statement. The statement is executed only if the guard expression evaluates to true.

7. The data and control flow dependences are updated to account for the execution variables and guard expressions inserted.

The method outlined above can be generalized to conditional statements other than if-then-else statements (for example switch statements) by allowing the execution variable to assume an arbitrary number of values.

Figure 4.7 shows the distribution for an example loop taken from [KM90]. Two execution variables, *EV1* and *EV5*, are introduced that correspond to the conditional expressions at lines 1 and 5. The loop is distributed into four parts as shown.



Original Loop

```
     for (int i=0; i<n; i++) {
1        if (a[i] > t)
2            a[i] = i;
         else {
3            t = t + 1;
4            f[i] = a[i];
5            if (b[i] != 0)
6                u = a[i] / b[i];
             else {
7                u = a[i] − u;
8                c[i] = b[i] + c[i];
             }
         }
9        d[i] = d[i] + c[i];
     }
```

Partition 1 = Statements (1, 2, 3)
Partition 2 = Statements (4, 5)
Partition 3 = Statements (6, 7)
Partition 4 = Statements (8, 9)

Loop after Transformation

```
     for (int i=0; i<n; i++) {
1        EV1[i] = a[i] > t;
         if (EV1[i] == true)
2            a[i] = i;
         else
3            t = t + 1;
     }

     for (int i=0; i<n; i++) {
         EV5[i] = undefined;
         if (EV1[i] == false) {
4            f[i] = a[i];
5            EV5[i] = b[i] != 0;
         }
     }

     for (int i=0; i<n; i++) {
         if (EV5[i] == true)
6            u = a[i] / b[i];
         else if (EV5[i] == false)
7            u = a[i] − u;
     }

     for (int i=0; i<n; i++) {
         if (EV5[i] == false)
8            c[i] = b[i] + c[i];
9        d[i] = d[i] + c[i];
     }
```

Figure 4.7: Distribution of a Loop With Conditional Statements

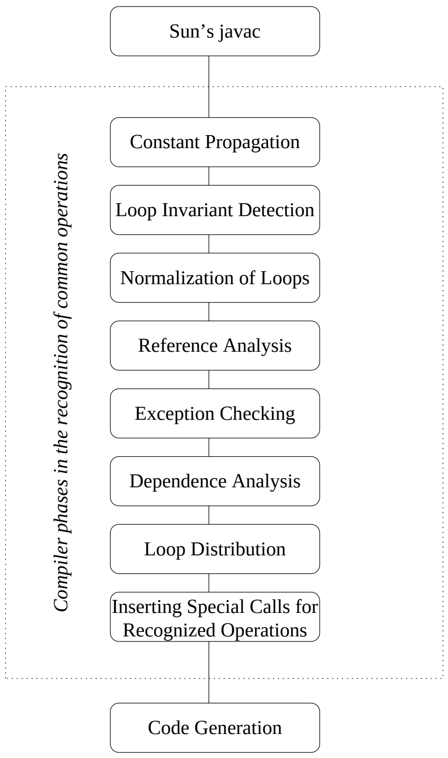


Figure 4.8: The Recognition Engine

4.5 Implementation

An existing Java parser (Sun's *javac* from JDK 1.2.2) is used as a starting point and modified to incorporate the analyses and transformations discussed. Figure 4.8 shows the different phases added in the compiler.

Sun's *javac* parser does some basic optimizations like constant folding and dead code elimination. We add constant propagation and loop invariant detection to improve the accuracy of the dependence tests.

Since only loops that are in the normalized form (as discussed in section 4.1.1) are analysed for dependences and loop distribution, a loop normalization phase is included that normalizes loops not in the standard form. It is not possible to normalize all loops as for some of them, the upper bound cannot be inferred from the code.

Next, reference analysis, exception checking, dependence analysis, and loop distribution are done in that order.

Finally, the basic operations can be recognized, and the corresponding code is transformed. This is discussed in section 4.6.

The following sections describe the reference analysis, dependence analysis, and loop distribution implementations in greater detail.

4.5.1 Reference Analysis

The reference analysis uses the iterative algorithm described in section 4.2. The abstract syntax tree previously created by the parser is used to traverse the program control flow for each iteration. The transfer functions used by this algorithm are written in Java. The implementation for the transfer functions was done by Peng Wu. At the end of the analysis, every reference access at each statement point in the program is mapped to a set of heap locations that may be referenced by it.

After the iterative algorithm has reached its fixed point, a single sequential traversal of the program removes redundant *nulls* from the reference mappings. Information about the bounds of array references is also stored along with their reference mappings. This helps in exception checking as described in section 4.3.

4.5.2 Dependence Analysis

First, we collect a read-set and a write-set of all accesses to memory locations for each loop nest in the program. The read-set is the union of the sets USE(S), where S is any statement in the loop nest. Similarly, the write-set is the union of the sets DEF(S), where S is any statement in the loop nest. USE and DEF are defined in section 4.1.4.

The reference accesses within the read-set and the write-set are then translated to sets of heap objects. This is done using the mappings obtained from the reference analysis.

Heap object names resulting from these mappings are identifiers which can be followed by some subscript expressions. These subscripts correspond to the iteration vector for the *new* statement that allocates the heap object.

For an array access *a[i]*, the heap object will be the reference mapping for a subscripted by i. For a field access $a.i$, the heap object will also be the reference mapping for a subscripted by i. This subscripting applies only to the last dimension of an access. Thus, for an access *a[i][j]*, the heap object will be the reference mapping for *a[i]* subscripted by j.

Finally, the heap object names for each pair of accesses of compatible Java types are tested for dependence. At least one access of each pair must be a write access. A dependence exists if:

- The names of both the accesses are identical, and
- Both the accesses are followed by the same number of subscripts, and
- Each pair of corresponding subscript expressions has the same value.

For the subscript expressions, a conventional dependence test can be applied to determine if they may have the same value. In the implementation, the Omega test described in section 4.1.7.3 is used.

The interface to the Omega library [KMP+96] is modified so that it takes as parameters:

- A set of variables that correspond to the loop variables for the loops common to the two accesses being considered.
- A set of constraints on the bounds of these variables.
- The subscript expressions for the two accesses to the same array. Each subscript can be an affine expression of the loop variables or a special *unknown* value.

The returned value is a direction vector that gives the dependence relation for the common loops enclosing the two accesses.

Thus, we obtain direction vectors for all data dependences between the accesses in loop nests.

4.5.3 Loop Distribution

The exception checking and dependence information obtained previously is used for loop distribution.

First, the access-wise dependences are converted to a statement-wise dependence graph. Each node in the graph corresponds to a statement contained in the loop that is to be distributed. There is an edge from node A to node B in the graph if there is a dependence from some access in statement A to some access in statement B.

Next, we find the strongly connected components in the dependence graph using Tarjan's algorithm. This algorithm is given in the Appendix.

Then, we obtain the acyclic condensation of the graph. We also reorder statements such that all statements belonging to the same strongly connected component are adjacent to one another in program order. During this reordering, we have to be careful not to move code across the boundary of any statement that can throw an exception. If reordering is not possible because of statements that can throw exceptions, the loop cannot be distributed.

Finally, a topological traversal of the nodes in the acyclic condensed graph is done, and the statement(s) contained in each node are put in a separate loop. Thus, the original loop is distributed into as many loops as there are strongly connected components.

Loop distribution is applied to each loop in the loop nest beginning with the outermost loop. Each time, the set of dependences is pruned to exclude those dependences that are loop-carried on the outer loops only.

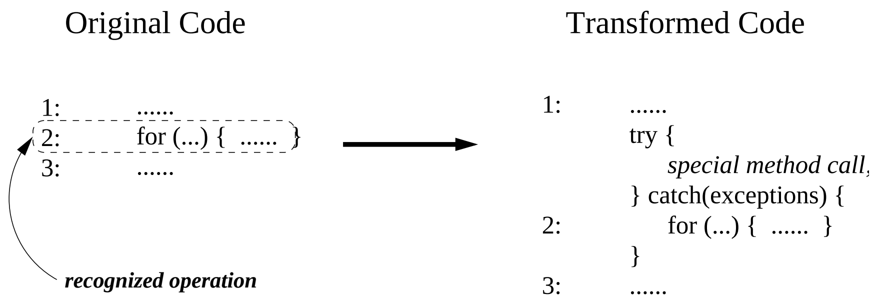


Figure 4.9: Insertion of Special Method Calls

4.6 Recognizing Operations and Inserting Special Calls

We compile a database of basic operations commonly found in the computation kernels of numerical programs. These are typically simple operations on vectors or matrices done within a single or double-nested loop. After loop distribution, we can identify the maximum number of instances of these basic operations in the source code.

The recognition is implemented using the program abstract syntax tree. Each basic routine is represented in the form of a tree pattern. Recognition of the routines can be done by tree pattern matching. It is possible to make this process simple and efficient using a canonical ordering of the abstract syntax tree [MW00].

When a match is performed, it is important to ensure that the statements involved cannot throw any exceptions other than *ArrayIndexOutOfBoundsException* and *Null-PointerException*. These two exceptions are explicitly checked by our native implementation since they are the most common in numerical codes. If it is possible for the statement to throw any other exception, it cannot be replaced by native code. In this case, the native implementation will not be able to handle the exception and therefore will not adhere to Java's precise exception semantics (defined in section 4.3.1).

Whenever a match is found, the corresponding sequence of statements is replaced by a try-catch statement as shown in Figure 4.9. The try block contains a call to a special static method with the appropriate parameters. The exceptions caught by the catch-block are *NoClassDefFoundError*[5], *NullPointerException*, and *ArrayIndexOutOfBoundsException*. The body of the catch-block contains the original sequence of statements that matched the basic operation. This ensures that the bytecode instructions generated are portable, as discussed in section 6.2.

Chapter 6 talks in greater detail about run-time interpretation and what happens when these specially inserted try-catch statements are encountered during execution.

[5]Errors are exceptions that usually cannot be recovered from, and therefore they are not caught in general. This is not the case in our implementation though.

Chapter 5

Library of Common Basic Operations

We list the basic operations that are found commonly in the computational kernels of numerical codes. These operations are optimized by our technique and replaced by native method calls. Since the set of sample codes studied is limited, this list is not comprehensive. We also identify those operations performed commonly in loops that are not recognized by our implementation. These are not replaced by native method calls.

In the implementation, we write our own native code library for the basic operations. The native code must take care to check for exceptions, and keep the same numerical precision defined by Java semantics. It must be transactional in nature, i.e. either the native call completes successfully, or it does not make any modifications to the user-visible program state if an exception is thrown. It is possible to use readily available, highly optimized library calls[1] for the actual computation. These calls will be wrapped around by code that makes sure there are no exceptions and takes care of any other functionality specific to the Java interface.

[1]For example, the BLAS library [JgJ] can be used.

5.1 Common Operations Recognized

The commonly occurring basic operations are enumerated below. In the following examples, a and b represent matrices, c and d represent vectors, m, n, x, y, and t represent loop invariants, and i and j represent loop variables. *f()*, *f'()*, *g()*, and *g'()* denote affine functions of loop variables.

1. Assigning to the elements of a matrix. The values assigned can be:

   (a) a constant:

   ```
   for (i=0; i<m; i++)
     for (j=0; j<n; j++)
       a[f(i,j)][f'(i,j)] = t;
   ```

   (b) elements of a matrix:

   ```
   for (i=0; i<m; i++)
     for (j=0; j<n; j++)
       a[f(i,j)][f'(i,j)] =
                 b[g(i,j)][g'(i,j)];
   ```

   (c) an affine expression of the loop variables:

   ```
   for (i=0; i<m; i++)
     for (j=0; j<n; j++)
       a[f(i,j)][f'(i,j)] = g(i,j);
   ```

2. Assigning to the elements of a matrix row or column. The values assigned can be:

   (a) a constant:

   ```
   for (i=0; i<m; i++)
     a[a1][a2] = t;

   where: a1=f(i) and a2=x OR a1=x and a2=f(i)
   ```

   (b) elements of a vector:

   ```
   for (i=0; i<m; i++)
     a[a1][a2] = d[g(i)];

   where: a1=f(i) and a2=x OR a1=x and a2=f(i)
   ```

(c) elements of a matrix row or column:

```
for (i=0; i<m; i++)
  a[a1][a2] = b[b1][b2];

where: a1=f(i) and a2=x OR a1=x and a2=f(i)
       b1=g(i) and b2=y OR b1=y and b2=g(i)
```

3. Assigning to the elements of a vector. The values assigned can be:

(a) a constant:

```
for (i=0; i<m; i++)
  c[f(i)] = t;
```

(b) elements of a vector:

```
for (i=0; i<m; i++)
  c[f(i)] = d[g(i)];
```

(c) elements of a matrix row or column:

```
for (i=0; i<m; i++)
  c[f(i)] = b[b1][b2];

where: b1=g(i) and b2=y OR b1=y and b2=g(i)
```

4. Assign-op expression with the l-values as the elements of a matrix. The operator can be +, -, * or /, though + and * are more common. The right hand side expression can be:

(a) a constant:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    a[f(i,j)][f'(i,j)] op= t;
```

(b) a matrix element:

```
for (i=0; i<m; i++)
  for (j=0; j<n; j++)
    a[f(i,j)][f'(i,j)] op=
              b[g(i,j)][g'(i,j)];
```

5. Assign-op expression with the l-values as the elements of a matrix row or column. The right hand side of the expression can be:

   (a) a constant:

   ```
   for (i=0; i<m; i++)
     a[a1][a2] op= t;

   where: a1=f(i) and a2=x OR a1=x and a2=f(i)
   ```

   (b) a vector element:

   ```
   for (i=0; i<m; i++)
     a[a1][a2] op= d[g(i)];

   where: a1=f(i) and a2=x OR a1=x and a2=f(i)
   ```

6. Assign-op expression with the l-values as the elements of a vector. The right hand side of the expression can be:

   (a) a constant:

   ```
   for (i=0; i<m; i++)
     c[f(i)] op= t;
   ```

   (b) a vector element:

   ```
   for (i=0; i<m; i++)
     c[f(i)] op= d[g(i)];
   ```

7. Assign-op expression with the l-values as the elements of a matrix row or column. The right hand side of the expression is a vector element multiplied by a constant:

   ```
   for (i=0; i<m; i++)
     a[a1][a2] op= t * d[g(i)];

   where: a1=f(i) and a2=x OR a1=x and a2=f(i)
   ```

8. Assign-op expression with the l-values as the elements of a vector. The right hand side of the expression is a vector element multiplied by a constant:

   ```
   for (i=0; i<m; i++)
     c[f(i)] op= t * d[g(i)];
   ```

9. Vector dot product, where the two vectors are matrix rows or columns:

```
for (i=0; i<m; i++) {
  ......
  dotprod = t;
  for (j=0; j<n; j++)
    dotprod += a[a1][a2] * b[b1][b2];
}

where: a1=j and a2=x OR a1=x and a2=j
       b1=j and b2=y OR b1=y and b2=j
```

10. Vector dot product, where the two vectors are one dimensional arrays by themselves:

```
for (i=0; i<m; i++)
  dotprod += c[i] * d[i];
```

11. Adding a multiple of one row or column of a matrix to another row or column of a matrix:

```
for (i=0; i<m; i++)
  a[a1][a2] += t * b[b1][b2];

where: a1=f(i) and a2=x OR a1=x and a2=f(i)
       b1=g(i) and b2=y OR b1=y and b2=g(i)
```

12. Swapping elements of a vector:

```
for (i=0; i<m; i++) {
  tmp = c[f(i)];
  c[f(i)] = c[g(i)];
  c[g(i)] = tmp;
}
```

Table 5.1 summarizes the number of occurrences of each of these operations in the set of benchmark programs used by us. The benchmark programs are described in section 7.1. The first column of the table gives the absolute count of the number of instances of the operation recognized across all benchmarks, and the second column gives the number of distinct programs (out of a total of eight benchmark programs) in which the operation occurred.

This shows that the distribution of the operations in the library is not concentrated in single programs, but is spread over the benchmarks.

No.	Routine	Total Occurrences	Number of benchmarks
1a	MatrixAssignConst	4	4
1b	MatrixAssignMatrix	7	3
1c	MatrixAssignAffine	7	5
2a	MatrixRowAssignConst	9	3
2b	MatrixRowAssignVect	7	2
2c	MatrixRowAssignMatrixRow	2	2
3a	VectAssignConst	4	4
3b	VectAssignVect	4	2
3c	VectAssignMatrixRow	5	3
4a	MatrixOpConst	1	1
4b	MatrixOpMatrix	4	1
5a	MatrixRowOpConst	6	2
5b	MatrixRowOpVect	2	2
6a	VectOpConst	6	2
6b	VectOpVect	3	2
7	MatrixRowOpMulExpr	8	4
8	VectOpMulExpr	12	4
9	MatrixRowDotProduct	12	5
10	VectDotProduct	5	2
11	MatrixRowTransform	2	2
12	SwapVectElements	8	2

Table 5.1: Number of Occurrences of Basic Operations

5.2 Operations Not Recognized

Our implementation does not recognize all operations in loops that can be replaced with native code. In particular, the following operations are not supported yet:

- Operations involving conditional expressions. Examples of these are found in the *idamax* and *dgesl* routines of the standard BLAS library [JgJ]. The form of these operations is:

```
for (int i=0; i<n; i++) {
        if (conditional expression) {
                    . . .
        }
}
```

- Operations involving method calls or arbitrarily complex expressions. These include the use of mathematical functions like *abs* and *sqrt*, and file operations like *read* and *write*. A very common pattern is a loop that is used to initialize array elements with values read from a data file.

There are a few loops in the benchmark programs studied (described in section 7.1) that have *break* or *continue* statements. Also, the upper bound is not known at compile time for some loops. These loops cannot be normalized, and our optimization technique is not applied to them.

Chapter 6

Revised Interpretation

Here, we discuss the modifications we make in the run-time interpreter to realize the benefits of the static analysis. We also discuss why this will not interfere with the semantics defined for the Java language [GJS96] and the Java Virtual Machine (JVM) [LY97].

6.1 Modifications to the Interpreter

The compiler inserts special method calls for the basic operations that it recognizes. These are calls to static methods of a special Java class, which we call the *BasicOperations* class. The implementation body of these methods is empty. They only serve as handles to identify a particular operation.

The modified interpreter must be provided with two things:

1. the pre-compiled library of native code for the frequently occurring basic operations.
2. the *BasicOperations* class. The special methods inserted previously by the modified compiler belong to this class.

At run-time, calls to these special methods result in an invocation of the corresponding routine in the native library.

The run-time interpreter (the Java Virtual Machine) is based on a stack model. A program is run by decoding and executing a stream of bytecode instructions. When the first call to a method of the special *BasicOperations* class is encountered, the class is loaded and initialized. As part of this process, a method table is created. This maps each static method of the class to a native method in the pre-compiled library, rather than a Java method. Thereafter, each call to a method of the *BasicOperations* class is efficiently translated to the corresponding native method call using this table.

By using method calls as the mechanism to let the interpreter know that a basic operation has been identified, it becomes easy to specify the parameters that must be passed to the corresponding library call. This is done via the normal method invocation mechanism. The parameters for a method call are automatically pushed on stack before the bytecode instruction for method invocation is executed.

Though it appears similar to the Java Native Interface (JNI) method invocation, our technique is significantly different. We use internal knowledge of the Java stack and interpreter implementation which gives us greater efficiency. This is illustrated by the data presented in section 7.2.3.2.

6.2 Validity of the Transformation

Our modification to the interpreter implementation does not violate the definition of the Java Virtual Machine. The native method implements the exact same functionality contained in the original source code. We also account for all the exceptions that may be thrown. This is done as follows:

If user-defined exceptions exist, the native code cannot check to see whether they may or may not be thrown during execution. In this case, the compiler does not insert the special method call, and the original code remains the same. If there are run-time exceptions (like *ArrayIndexOutOfBoundsException* and *NullPointerException*), they are all checked to see if they might be thrown during execution. This checking is done at the beginning of the native method before any program state is modified. This can be done efficiently as the native methods are integrated with the Java Virtual Machine implementation and are aware of the internal data layouts. They can access information, such as length of objects allocated by the memory manager, in the same way that the interpreter does. If the method determines that an exception may be thrown, then it does not perform any computation, but directly throws the exception. The native method terminates abruptly and control is transferred to the catch-block for the exception. This catch-block is the one that is introduced specially by our compiler. The catch-block contains the original code that corresponds to the native method call. This original code is executed, and the exception is thrown again at the point in the computation where it is actually raised. Thus, precise exception semantics are maintained.

The bytecode instructions generated by the compiler are portable across all conforming virtual machine implementations. The call to the special method is inserted within a try block. If our modified interpreter is not used, the *BasicOperations* class will not exist. In this case, a *NoClassDefFoundError* will be thrown by the class loader. This will be caught, and will cause the catch block to execute. The catch block contains the sequence of code that corresponds to the method call, so execution of the program will be exactly the same as the original Java source program. The exception that was thrown is not visible to the user. Thus, the program will execute correctly irrespective of which interpreter is used.

Our implementation will not cause a name conflict if users attempt to define a class of their own named *BasicOperations*. This is because a class in Java can be qualified by its location in the directory structure of the file system. We make sure the class is placed in a directory specific to where the virtual machine is installed on the system. Whenever the compiler inserts a special static method call, it uses the fully qualified class name. This ensures that our newly introduced class does not interfere with any user classes. However, the CLASSPATH variable set by the user must not include the directory where our *BasicOperations* class is placed. Since this directory is specific to the virtual machine installation, users should not be aware of it, and it should not be included in the CLASSPATH setting.

Chapter 7

Experimental Evaluation

In this chapter, we describe the benchmark programs used for testing our implementation, and the performance results obtained.

7.1 The Benchmark Programs

We used all the five programs in the Jama benchmark [MwJ]: Cholesky decomposition, LU factorization, QR decomposition, singular value decomposition, and eigenvalue decomposition. In addition, we have three other benchmark programs: matrix multiplication, a successive over-relaxation routine[1], and multidimensional minimization of a function using the downhill simplex method (amoeba)[2].

Table 7.1 lists these programs and the number of lines of source code for each.

The programs are modified to inline method calls that may change any reference values reachable from the parameters passed to the method. This is needed because the reference analysis algorithm does not handle inter-procedural analysis.

[1]from the Java Grande Benchmark suite.

[2]adapted from the FORTRAN source in [PFTV86, pgs 292-293].

Program	Description	Lines of code
matmul	matrix multiplication	65
sor	successive over-relaxation routine	62
amoeba	downhill simplex minimization of a function	247
cholesky	cholesky decomposition of a matrix	58
lufact	LU factorization of a matrix	178
qrd	QR decomposition of a matrix	218
svd	singular value decomposition of a matrix	531
evd	eigenvalue decomposition of a matrix	948

Table 7.1: Benchmark Programs Used

Our benchmark programs are representative of a subset of numerical computations. Even though we use a small number of codes, the results are valid for the purpose of evaluating our approach. This is because our library of commonly used basic operations is small too, and can be expanded for a wider range of applications.

7.2 Results

We present the effectiveness of the exception checking analysis and the loop distribution transformation. We also compare the run-time performance of our technique versus plain interpretation, JIT compilation, and an approach that uses the Java Native Interface (JNI).

7.2.1 Redundant Exception Checking

Table 7.2 gives the statistics for exception checking. For each benchmark, it shows the fraction of redundant exceptions detected by the actual number of redundant exceptions in the program. It does this for both *ArrayIndexOutOfBoundsException* and *NullPointerException*. The last column lists the number of loops found to be exception-free and the actual number of exception-free loops in the program. Statements can be reordered within exception-free loops.

Program	ArrayIndexOutOfBounds (redundant/total)	NullPointer (redundant/total)	Exception-free loops/Total loops
matmul	1.00	0.69	5/7
sor	1.00	0.59	5/7
amoeba	0.85	0.82	13/25
cholesky	1.00	0.53	4/8
lufact	0.44	0.68	6/15
qrd	0.88	0.70	13/21
svd	0.22	0.71	9/45
evd	0.28	0.63	17/78

Table 7.2: Redundant Exceptions and Exception-free Loops

Some exceptions are not detected as redundant. This is because symbolic expressions are not compared for array bounds in order to keep the implementation simple. However, this does not affect the analysis for our benchmarks, because exceptions do not inhibit code motion anywhere in the set of programs used.

7.2.2 Loop Distribution

Table 7.3 shows the number of new loops that are introduced in the transformed programs after loop distribution. The last column shows how many of these new loops recognize basic operations. We see that across all the benchmarks, a total of 33 routines are recognized due to this transformation. Thus, loop distribution has a significant impact on the effectiveness of our technique.

Program	Number of new loops	Extra routines recognized
matmul	3	2
sor	1	1
amoeba	12	11
cholesky	0	0
lufact	1	1
qrd	3	3
svd	3	3
evd	13	13

Table 7.3: Effect of Loop Distribution

7.2.3 Performance

All measurements were done on a SUN Ultra Sparc 5 running under SunOS 5.7. Different sizes were used in the experiments for the dimensions of the matrices or vectors in the programs. Execution times were obtained for each of these problem sizes.

7.2.3.1 Comparison with Plain Interpretation and JIT Compilation

We measure the execution time for each of the benchmark programs for the following three systems:

- Compiling with Sun's *javac*, and using the normal run-time interpreter.
- Compiling with Sun's *javac*, and using the JIT compiler at run-time.
- Compiling and executing with our modified compiler and interpreter.

The results are shown in the graphs in Figure 7.1. Each graph shows the data for a single benchmark. There are three curves in each graph; each curve represents one of the three systems listed above. The execution times are plotted for different sizes of the problem.

We observe that the plain interpreted version executes much slower than the other two systems for all cases. For two benchmarks, namely *matmul* and *cholesky*, our modified interpreted version performs better than the JIT version. For one benchmark, *sor*, it is much slower than the JIT version. For all the remaining benchmarks, the performance of our version of the interpreter is very close to the JIT version. For the problem size of 1000, the JIT version executed 1.18 times faster than our version on average.

Our technique is limited by Amdahl's Law since it does not apply to the entire program. The cases where our modified interpreter performs the best of all the three systems are those where almost the entire code comprises of a sequence of basic computational operations. These cases show that even when using an interpreter, there is scope for improvement over the JIT compiler.

7.2.3.2 Comparison with JNI

The Java Native Interface (JNI) is a mechanism that the Java language defines to allow programmers to interact with native methods from Java code. For the JNI approach, the native method calls that our compiler inserts for the basic operations it recognizes are invoked using the JNI mechanism.

Table 7.4 shows the ratio of the execution time using JNI and the execution time using our modified interpreter. The last column gives the average of the results for all problem sizes.

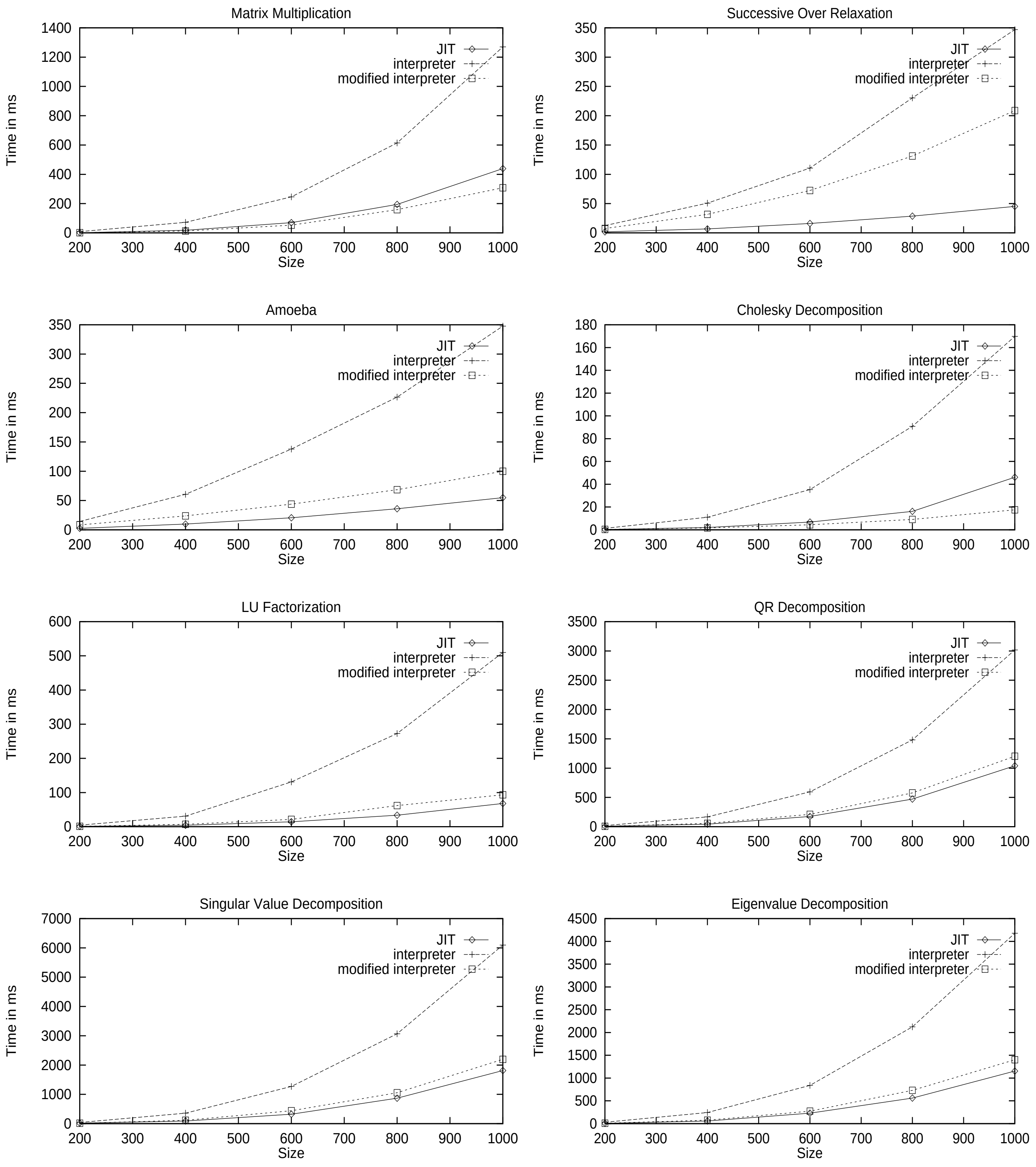


Figure 7.1: Execution Times for the Benchmarks

Program	Problem Size					
	200	400	600	800	1000	average
matmul	14.8	14.1	12.9	9.6	10.0	12.3
sor	3.0	2.8	2.7	2.7	2.6	2.8
amoeba	2.2	2.5	2.6	2.6	2.6	2.5
cholesky	2.5	2.8	2.9	3.1	2.9	2.8
lufact	2.3	2.4	2.8	1.9	2.6	2.4
qrd	13.3	13.9	13.4	11.1	10.6	12.5
svd	12.6	13.8	12.4	11.8	11.3	12.4
evd	7.0	8.1	7.7	6.7	6.7	7.2

Table 7.4: Comparison with JNI Performance

We see that our system always performs better than JNI and in many cases the difference is significantly large.

Chapter 8

Conclusions

In our work we target a specific set of applications, viz. those that are in the domain of numerical computing. We improve the run-time performance of Java programs written for these applications. This is done by analyzing the programs to find all instances of a predetermined set of commonly occurring basic operations. These operations are expressed in the form of loops or parts of a loop that are separable. The Java interpreter is modified to incorporate a library of native code for these operations. We show that the use of native code to execute these operations improves performance by a factor of 7.6 times maximum and 4 times on the average.

Our technique allows time-consuming analyses to be done off-line. It optimizes program loop nests. This makes it effective because loop nests usually contribute more to the dynamic execution time than other parts of the program. The technique needs support from both the static compiler and the run-time interpreter.

Since our technique does not exclude other methods, it can be combined with other approaches to achieve better results. For example, it can be used in conjunction with a just-in-time (JIT) compiler. Section 7.2.3 shows that our system outperforms JIT compilation for programs comprising mostly of the basic operations. Use of our technique together with JIT compilation can give the JIT compiler more time to optimize other parts of the code, since it will then be working on a smaller subset of the program.

Thus, the method presented is a simple, effective way to enhance the performance of numerical Java programs.

Chapter 9

Future Work

A wide range of numerical codes must be studied to assemble a larger library of basic operations that occur commonly. Operations that use conditional expressions and mathematical functions like *abs* and *sqrt* need to be handled.

Our technique must be integrated with a JIT compiler to see how effective the unified approach is.

Currently, we handle arbitrary arithmetic expressions by breaking them down into a sequence of simpler expressions, which are then recognized as basic operations. For example, consider the code segment shown below:

```
for (int i=0; i<n; i++) {
    a[i] = x * a[i + 1] + a[i];
}
```

It is broken down into several expressions like this:

```
for (int i=0; i<n; i++) {
    tmp[i] = 0;
    tmp[i]  += x * a[i + 1];
    tmp[i]  += a[i];
    a[i] = tmp[i];
}
```

The array *tmp* is a temporary array that is introduced by us. The sub-expressions correspond to basic operations, and they can be evaluated using native method calls. The benchmark program *sor* has such expressions in loop statements. The performance we get in this case is much better than the plain interpreted version, but lags far behind the JIT version. In the present scheme, each intermediate result causes the creation of a new temporary array object. This needs to be optimized.

A better implementation using symbolic manipulations is needed to improve the detection of redundant exceptions. Techniques discussed in [BGS00] can be used too. If we detect more redundant exceptions, we can determine more statements to be exception free. Since statements that could possibly throw exceptions inhibit code motion, we will then have greater flexibility in reordering code. This can allow more loops to be distributed and make it possible for a greater number of basic operations to be identified.

Currently, we recognize only specific syntactic patterns as basic operations. The technique can be extended to recognize algorithms. For example, matrix multiplication can be recognized as such, irrespective of whether it is implemented in a straightforward manner, or using blocking, or loop unrolling. Thus to some extent, the optimization can be made independent of the particular implementation used by the programmer.

Appendix: Tarjan's Algorithm

Here, we give Tarjan's algorithm [Tar72] to determine the strongly connected components of a directed graph. It uses a variant of depth-first search and solves the problem in linear time.

Input: A directed graph G = (N,E). N is the set of graph nodes, and E is the set of graph edges.
Output: A list of the strongly connected components of G.
Method:
/* The algorithm uses a push-down stack. With every node $n \in N$, three variables are associated: *visited(n)*, *dfn(n)*, and *link(n)*. Furthermore, the algorithm uses counters *df-count* and *r*.

The number *link(n)* computed in search(n) can be characterized as follows:

link(n) = min({*dfn(w)*: $w = n$ or there is a descendant *n'* of n and an edge (n',w) with *dfn(n')* >= *dfn(w)* such that the root of the strongly connected component containing w is an ancestor of n})

According to this definition, *link(n)* <= *dfn(n)* $\forall n \in N$.

A node n is the root of a strongly connected component iff *link(n)* = *n*.

*/

```
procedure search(n);
begin
init_stack;
visited(n) := true;
```

```
dfn(n)      := df_count;
link(n)     := df_count;
df_count plus 1;
push(n);
for every n' in succ(n) do
   if not(visited(n'))
      then search(n');  link(n) := min(link(n),link(n'));
      else  /* visited(n') */
         if dfn(n') < dfn(n) and in_stack(n')
            then  /* n' is the same strongly connected component as an
                     ancestor of n
                  */
               link(n) := min(link(n),dfn(n'));
         fi
   fi
end for;

if link(n) = dfn(n)
   then /* n is the root of a strongly connected component whose nodes
           are the topmost elements of the stack, up to and including n.
           These elements are output and popped off the stack.
        */
      r plus 1;  write(`Strongly connected component number: ', r);
      repeat
         x := top_of_stack;  pop;  write(x);
      until x=n
fi
end search


/* MAIN PROGRAM */

begin
df_count := 1;
r := 0;
for every node n in the graph do
   visited(n) := false
end for;
while there exists a node n in the graph such that not(visited(n)) do
   search(n)
end while
end
```

Bibliography

[Aea00] B. Alpern and et al. The Jalapeno virtual machine. *Special issue of IBM Systems journal on Java performance, 39(1)*, 2000.

[AGMM99] Pedro Artigas, Manish Gupta, Samuel Midkiff, and José Moreira. High performance numerical computing in Java: Language and compiler issues. In *Languages and Compilers for Parallel Computing*, pages 1–17, 1999.

[ANH99] Ana Azevedo, Alex Nicolau, and Joseph Hummel. Java annotation-aware just-in-time (AJIT) compilation system. In *Proceedings of the ACM 1999 Java Grande Conference*, San Francisco, June 1999.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles Techniques and Tools.* Addison-Wesley, 1986.

[ATCL+98] A. Adl-Tabatabai, M. Cierniak, G. Lueh, V. Parikh, and J. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In *Proceedings of the ACM SIGPLAN '98 conference on Programming Language Design and Implementation*, pages 280–290, Montreal, Canada, June 1998.

[BC98] Brian Blount and Siddhartha Chatterjee. An evaluation of Java for numerical computing. In *ISCOPE*, pages 35–46, 1998.

[BDP+98] Ronald F. Boisvert, Jack J. Dongarra, Roldan Pozo, Karin A. Remington, and G. W. Stewart. Developing numerical libraries in Java. *Concurrency: Practice and Experience*, 10(11–13):1117–1129, 1998.

[BG97] Aart J. C. Bik and Dennis B. Gannon. A note on native level 1 BLAS in Java. *Concurrency: Practice and Experience*, 9(11):1091–1099, 1997.

[BGS00] Rastislav Bodik, Ragiv Gupta, and Vivek Sarkar. ABCD: Eliminating array bounds checks on demand. In *Proceedings of the SIGPLAN '00 Conference on Program Language Design and Implementation*, Vancouver, Canada, June 2000.

[BK97] Z. Budimlic and K. Kennedy. Optimizing Java: Theory and practice. In *Software—Practice and Experience 9, 6*, pages 445–463, June 1997.

[BVG97] Aart J. C. Bik, Juan E. Villacis, and Dennis B. Gannon. javar: A prototype Java restructuring compiler. *Concurrency: Practice and Experience*, 9(11):1181–1191, 1997.

[CDD97] Henri Casanova, Jack Dongarra, and David M. Doolin. Java access to numerical libraries. *Concurrency: Practice and Experience*, 9(11):1279–1291, 1997.

[CFM$^+$97] Timothy Cramer, Richard Friedman, Terrence Miller, David Seherger, Robert Wilson, and Mario Wolczko. Compiling Java just in time: Using runtime compilation to improve Java program performance. *IEEE Micro*, 17(3):36–43, 1997.

[CL97a] Michael Cierniak and Wei Li. Optimizing Java bytecodes. *Concurrency: Practice and Experience*, 9(6):427–444, June 1997.

[CL97b] Michał Cierniak and Wei Li. Just-in-time optimizations for high-performance Java programs. *Concurrency: Practice and Experience*, 9(11):1063–1073, 1997.

[Cla97] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[CLS00] Michal Cierniak, Guei-Yuan Lueh, and James M. Stichnoth. Practicing JUDO: Java under dynamic optimizations. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 13–26, 2000.

[CPS$^+$99] C. Chambers, I. Pechtchanski, V. Sarkar, M. Serrano, and H . Srinivasan. Dependence analysis for Java. In *In 12th International Workshop on Languages and Compilers for P arallel Computing*, August 1999.

[For] Java Grande Forum. The Java Grande Forum Benchmark Suite. http://www.javagrande.org/.

[For98] Java Grande Forum. Java grande forum panel report: Making Java work for high-end computing. Orlando, Florida, November 1998.

[GHM98] Vladimir Getov, Susan Flynn Hummel, and Sava Mintchev. High-performance parallel programming in Java: exploiting native libraries. *Concurrency: Practice and Experience*, 10(11–13):863–872, 1998.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java$^{(TM)}$ Language Specification.* Addison-Wesley, 1996.

[HAKN97] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, 1997.

[HGH96] Cheng-Hsueh A. Hsieh, John C. Gyllenhaal, and Wen-mei W. Hwu. Java bytecode to native code translation: The Caffeine prototype and preliminary results. In *International Symposium on Microarchitecture*, pages 90–99, 1996.

[HN00] Allan Heydon and Marc Najork. Performance limitations of the Java core libraries. *Concurrency - Practice and Experience*, 12(6):363–373, 2000.

[IKY$^+$99] K. Ishizaki, M. Kawahito, T. Yasue, M. Takeuchi, T. Ogasawara, T. Suganuma, T. Onodera, H. Komatsu, and T. Nakatani. Design, implementation, and evaluation of optimizations in a just-in-time compiler. *ACM 1999 Java Grande Conference*, pages 119–128, June 1999.

[Inc] Sun Microsystems Inc. Updates to the Java$^{(TM)}$ language specification for JDK$^{(TM)}$ release 1.2 floating point. Available at: http://java.sun.com/docs/books/jls/strictfp-changes.pdf.

[Inc99] Sun Microsystems Inc. The Java Hotspot$^{(TM)}$ performance engine architecture. http://java.sun.com/products/hotspot/whitepaper.html, April 1999.

[JgJ] BLAS (Basic Linear Algebra Subprograms). http://www.netlib.org/blas/.

[jsr] JSRs: Java Specification Requests. http://jcp.org/jsr/stage/jsr.jsp.

[KD98] W. Kahan and Joseph D. Darcy. How Java's floating-point hurts everyone everywhere, 1998.

[KG97] A. Krall and R. Grafl. Cacao - a 64-bit Java VM just-in-time compiler. In *PPoPP '97 Workshop on Java for Science and Engineering Compuation*, 1997.

[KM90] K. Kennedy and K. S. McKinley. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, New York, NY, 1990.

[KMP$^+$96] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library, Version 1.1.0 Interface Guide. url http://www.cs.umd.edu/projects/omega, November 1996.

[LY97] Tim Lindholm and Frank Yellin. *The Java$^{(TM)}$ Virtual Machine Specification.* Addison-Wesley, 1997.

[MMBC97] G. Muller, B. Moura, F. Bellard, and C. Consel. Harissa: A flexible and efficient Java environment mixing bytecode and compiled code. In *Third*

Usenix Conference on Object-Oriented Technologies and Systems, Portland, Oregon, June 1997.

[MMG98a] José Moreira, Samuel Midkiff, and Manish Gupta. A standard Java array package for technical computing. *IBM Research Report RC 21369(96233)*, December 1998.

[MMG98b] José Moreira, Samuel Midkiff, and Manish Gupta. From flop to megaflops: Java for technical computing. In *Proceedings of the 11th International Workshop on Languages and Compilers for Parallel Computing*, 1998. IBM Research Report 21166.

[MMGL98] José Moreira, Samuel Midkiff, Manish Gupta, and R.D. Lawrence. Parallel data mining in Java. Technical Report RC 21326, IBM T. J. Watson Research Center, 1998.

[MMS98] Samuel Midkiff, José Moreira, and Marc Snir. Optimizing bounds checking in Java programs. *IBM Systems Journal*, 37(3):409–453, August 1998.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufman, 1997.

[MW00] Robert Metzger and Zhaofang Wen. *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization.* MIT Press, 2000.

[MwJ] JAMA: A Java Matrix Package. The MathWorks and NIST. http://math.nist.gov/javanumerics/jama/.

[MY01] Hidehiko Masuhara and Akinori Yonezawa. Run-time bytecode specialization: A portable approach to generating optimized specialized code. Aarhus, Denmark, 2001.

[PFTV86] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes: The Art of Scientific Computing.* Cambridge University Press, 1986.

[PTB$^+$97] T. Proebsting, G. Townsend, P. Bridges, J. Hartman, T. Newsham, and S. Watterson. Toba: Java for Applications—A Way Ahead of Time (WAT) Compiler. In *Technical report, Department of Computer Science*, University of Arizona, Tucson, 1997.

[Pug91] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Supercomputing*, 1991.

[SBMG00] Mauricio Serrano, Rajesh Bordawekar, Samuel Midkiff, and Manish Gupta. Quicksilver: a quasi-static compiler for Java. In *Proceedings of OOPSLA*, pages 66–82, Minneapolis, MN, October 2000.

[SS98] M. Schwab and J. Schroeder. Algebraic Java classes for numerical optimization. In *ACM SIGPLAN Workshop on Java for High-Performance Network Computing*, 1998. URL: http://www.cs.ucsb.edu/ conferences/java98.

[Tar72] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[WFPS00] Peng Wu, Paul Feautrier, David Padua, and Zehra Sura. Points-to Analysis for Java with Applications to Loop Optimizations, 2000.

[WMMG98] Peng Wu, Samuel Midkiff, José Moreira, and Manish Gupta. Improving Java performance through semantic inlining. IBM Technical Report RC21313, 1998.

[WMMG99] Peng Wu, Samuel Midkiff, José Moreira, and Manish Gupta. Efficient support for complex numbers in Java. In *ACM 1999 Java Grande Conference*, CA, 1999.

[YMH98] K. Yoshizoe, T. Matsumoto, and K. Hiraki. Speculative parallel execution on JVM. *EuroPar Workshop on Java for High Performance Network Computing*, September 1998.

[YSP$^+$98] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, New York, NY 10036, USA, 1998. ACM Press.

[ZC91] Hans Zima and Barbara Chapman. *Supercompilers for parallel and vector computers.* ACM Press, 1991.