SEPARATING CONCERNS IN SCIENTIFIC SOFTWARE USING ASPECT-ORIENTED PROGRAMMING

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF ENGINEERING AND PHYSICAL SCIENCES

2006

By
Bruno Harbulot
School of Computer Science

Contents

List of Figures

7

List of Listings

Abstract

The work reported in this thesis attempts to build a better link between scientific programming and software engineering. Scientific software is concerned with both the mathematical model and the high performance of the computation, particularly via the use of parallel computers. After showing the tangling of code that results from the interaction of these two major concerns, this thesis demonstrates how to improve the engineering of scientific applications by applying a novel technique for separating concerns: Aspect-Oriented Programming (AOP). AOP aims to encapsulate concerns that "crosscut" the main program flow into separate entities, known as aspects.

The most mature aspect-oriented tool that has been available during the course of this project is AspectJ, which is an extension of Java. The Java Grande Forum benchmark suite, originally used for assessing the suitability of Java for High-Performance Computing, is utilised, together with AspectJ, in an attempt to separate the parallelisation concern from the numerical model. AspectJ can be used to write aspects for parallelising applications, but often at the cost of refactoring, the amount of which ranges from minor (extracting a method) to major (redesigning the whole application). The problems lie in the fact that the points where parallelisation should occur are not naturally join points in AspectJ (i.e. points where AspectJ can intervene).

Consequently, a model for a join point capable of handling loops —which are the main target of parallelisation— is proposed. This model goes beyond the present AspectJ models and demonstrates the need to recognise complex behaviours for an effective separation of concerns.

Finally, aspects for implementing parallelisation according to different schemes are presented, together with performance results. This demonstrates the flexibility of aspects for implementing parallelisation, which is always a crosscutting concern with respect to the main concern of high-performance numerical applications.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Copyright

Acknowledgements

Chapter 1

Introduction

1.1   Software engineering in scientific computing

This thesis reports an attempt to build a better link between two disciplines: scientific computing and software engineering. Nowadays, computers are widely used by scientists for both simulations and analysis of results of experiments. Weather forecasting or simulations of fluid flows are examples of scientific applications of computers. Scientists write software that suits the needs of this kind of application. Such applications usually require a large number of numerical calculations to be performed. For this reason, the emphasis has traditionally been placed on high performance, i.e. on computational speed. The techniques utilised for providing high performance rely on software engineering technologies and programming languages —such as Fortran— that have been neglected for a long time in other disciplines of the software industry. However, as these pieces of software become increasingly complex, their programmers have to cope more and more with software development issues, such as correctness, robustness, extensibility, re-usability and compatibility [Mey88, ch. 1].

In the meantime, software engineering, as a research area, has evolved and has had considerable influence on most other software areas. For example, object-oriented programming (OOP) [Mey88] is one of the main technologies used by software developers today. However, OOP has neither solved all the problems in software engineering, nor has it penetrated the scientific community, as yet.

1.2   Aspect-Oriented Programming

Software engineering aims to provide clear processes and mechanisms for designing and maintaining software. These processes are driven by the interactions between diverse concerns. The software engineering mechanisms are meant to provide programmers with a means to abstract and encapsulate these concerns and these interactions. The latest software engineering paradigm, Aspect-Oriented Programming, aims to deal with so-called "crosscutting" concerns, as described in the remainder of this section.

1.2.1   Concerns and software design

As defined in [IEE00, p. 4], " concerns *are those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders*". Concerns can be more generally defined as "*any matter of interest in a software system*" [SR02].

Designing a piece of software consists of abstracting the concerns involved into the appropriate constructs. Using procedural languages, such as C, Pascal or Fortran, designers encapsulate concerns into functions, procedures or subroutines; then these can be grouped into modules, with various degrees of inter-dependency. In object-oriented programming, the use of classes and inheritance provides a further degree of abstraction.

Functional, procedural or object-oriented programming languages have a common way of providing mechanisms for abstraction: concerns placed into functions, procedures or objects can be seen as *functional units* of the system [KLM+97]. Methods, classes and libraries are object-oriented constructs for encapsulating concerns at several degrees of granularity. An object-oriented design aims to make concerns match these constructs. As far as possible, this consists of mapping the concerns involved into a set of objects, with associated actions, linked by inheritance ("*is-a*") or by client ("*has-a*") relationships. As a result, the links between the objects represent relationships between the purposes of the different concerns. However, this kind of decomposition into functional units is not suitable for all kinds of concerns.

1.2.2 Crosscutting concerns and aspects

However well decomposed an object-oriented design may be, some concerns often interact with each other in such a way that they cannot be encapsulated properly within object-oriented constructs. These interactions lead to *code tangling* — when the elements of code for two concerns are in the same unit and cannot be dissociated— or to *code scattering*[1] —when a concern involves code spread across several units. Two concerns that are related in such a way that they imply code scattering or code tangling are said to *crosscut* each other. A concern that crosscuts the main purpose of a unit is a *crosscutting concern* (with respect to that unit's decomposition).

For example, in a system that provides its users with an e-mail service and a file repository service, the two sets of objects that provide these services will have an authentication concern in common. At least two units (sets of objects) will contain statements for this authentication concern. This is an example of code-scattering. If monitoring certain activities in a certain context is required, statements will be added in between the functional code (that performs what is to be monitored). This is an example of code-tangling. More generally, tracing and logging are typical examples of concerns that almost always crosscut the main component's purpose: the concern of tracing the behaviour of a component is different to the main concern that is implemented by this particular component. More examples of code-tangling in the context of scientific applications are presented in Section 1.3.

Aspect-Oriented Programming (AOP) is a recent programming paradigm that is aimed at providing a better separation of concerns in software. In AOP, functional, procedural and object-oriented paradigms are augmented with a means of encapsulating crosscutting concerns separately. In an aspect-oriented design, crosscutting concerns are encapsulated into *aspects*. A further description of aspect-oriented programming and its mechanisms is presented in Chapter 2.

1.3 Code-tangling in scientific software

Scientific applications are often focused on two concerns: the algorithm used by the calculation itself (related to the scientific model) and the high performance of its implementation. These are two separate concerns. Yet, in current procedural

---

[1]In most cases, code scattering implies some code tangling, since some elements of code are located in units that have their own purposes.

or object-oriented implementations, these elements of code are usually interlaced within the same unit of the system design. This is an example of code tangling, and it occurs as a direct consequence of scientific model concerns and high performance concerns crosscutting each other.

One example where crosscutting arises is in code written to be executed in parallel for achieving high performance. Section 1.3.1 gives an overview of the crosscutting that is involved when the parallelisation has to be managed explicitly. Section 1.3.2 shows how compiler directives let the programmer avoid the explicit coding of parallelisation. Section 1.3.3 presents a similar use of compiler directives for sparse matrix optimisation. Both uses of compiler directives still leave some elements of code that crosscut the algorithmic description of the computation. Section 1.3.4 shows, through the analysis of three implementations of the same algorithms, how the concern of parallelisation crosscuts the calculation concern that is the main purpose of the programs. This section aims to show the drawbacks of the design of these examples in terms of modularity.

1.3.1   Explicit parallelisation

Explicit parallelisation could be implemented using languages such as C (for example by explicitly managing the interaction between Unix-style processes). However, this kind of programming is not often used directly for scientific applications, which more commonly implement parallelism using compiler directives.

Java is the language used for most examples in this thesis.[2] The default way of implementing parallelism in Java is a form of explicit parallelisation. Java provides multi-threading natively through its `Thread` class. Concurrent instances of `Thread` can be executed over several processors, using currently available Java Virtual Machines (JVM). The language and API mechanisms used for implementing several concurrent threads rely on either extending the `Thread` class or implementing the `Runnable` interface [CWH00, OW99, ch. 2].

Parallelising a sequential program requires both a specific design using the `Thread`-related classes and the explicit instantiation and synchronisation of those threads. This usually implies refactoring the objects describing the computation [OW99, ch. 9]. As a result, elements of the computation are placed within constructs defining the threads, and statements for managing the threads are placed within the units originally dedicated to the description of the computation, as shown

---

[2] Further details about Java for scientific computing are presented in Section 1.4.

in Listing 1.1. Since the instructions for the original algorithm concern and the multi-threading concern cannot be clearly separated one from the other, this is an example of code-tangling and code-scattering.

Listing 1.1: Code-tangling when parallelising two actions in Java.

```java
public class Example {
    ...
    /** This method executes sequentially action1() and action2() */
    public void sequentialExample() {
        action1 () ;
        action2 () ;
    }
    ...


    /**
     This method executes action1() and action2() in parallel.
     The call to action1() has been relocated in a dedicated class
     implementing the Runnable interface.
     The concern of representing the actions to perform and the
     concern of running them in parallel are tangled with each other.
     */
    public void parallelExample() throws InterruptedException {
        Thread otherThread = new Thread (new Action1Runnable()) ;
        otherThread.start() ;
        action2() ;
        otherThread.join() ;
    }

    /** This class is a Runnable meant to execute action1() */
    class Action1Runnable implements Runnable {
        public void run() {
            action1 () ;
        }
    }
}
```

1.3.2 Compiler directives for parallelism

In general, explicit parallelisation is difficult. In scientific applications, parallelism is more often implemented using compiler directives or using APIs[3] (for example OpenMP [CDK⁺01] or MPI [SOW⁺95]). These techniques allow the programmer

---

[3]API stands for *Application Programming Interface*. It is a set of libraries with a well-defined interface that programmers can re-use.

to parallelise loops with little modification to the original structure, and to hide extra complexity such as that due to the explicit creation of processes and threads.

OpenMP provides a set of compiler directives and library functions that can be used in C, C++ and Fortran for parallelising programs for shared-memory multi-processor machines. There is also an equivalent of OpenMP for Java: JOMP [BK00]. Listing 1.2 shows a basic example of for-loop parallelisation with OpenMP in C. The OpenMP directive (`#pragma omp parallel for`) is used for compiling the for-loop in such a way that it can be executed in parallel on a multi-processor machine.

Listing 1.2: Example of loop parallelisation in OpenMP.

```c
#include <stdio.h>
#include <stdlib.h>

int main () {
  int i, n=20 ;
  int *squares = (int*) malloc (n*sizeof(int)) ;

#pragma omp parallel for
  for (i=0; i<n; i++) {
    fprintf (stderr, "Executing on thread %i.\n",omp_get_thread_num()) ;
    squares[i] = i*i ;
  }

  free(squares) ;
  return 0;
}
```

Although implemented with mechanisms different to OpenMP, MPI[4] is a Message Passing Interface standard that is also aimed at parallelising applications. MPI defines a standard API that has to be implemented by an MPI library. The functions of the MPI library are used to hide the complexity of passing data and messages between several threads of a program. Both MPI and OpenMP rely on inserting statements in the functional code. In the case of MPI, these statements are function calls, whereas in the case of OpenMP, these statements may also be compiler directives. They both rely on a library for abstracting some of the complexity of program parallelisation. There are prototype implementations of MPI for Java: mpiJava [BCF+98] and MPJ [CGJ+00].

---

[4]See http://www.mpi-forum.org/.

Other compiler directive approaches for implementing parallelism have been provided in Java, such as `javar` [BVG97], which uses a pre-processor for restructuring programs into multi-threaded programs by using annotations similar to OpenMP compiler directives.

1.3.3 Compiler directives for sparse matrices

Bik *et al.* have produced a way for improving modularity in scientific software that uses sparse matrices [BBKW98]. This consists of an extended Fortran compiler (the "sparse compiler") which selects routines optimised for sparse matrices in code written as if the matrices were dense. A program that represents matrices as two-dimensional arrays is converted automatically into a version that uses the sparse routines. The information regarding the sparse characteristics of the matrices is provided to the compiler by annotations (compiler directives). The example in Listing 1.3 shows compiler directives that describe array `A` as a banded matrix, that is to say, all the a_{ij} values where $1 - N \leq i - j \leq -6$ or $6 \leq i - j \leq N - 1$ are zero (the -6 and 6 values are relative to the width of the band chosen in this example). Programmers can then write operations on array `A` as if it was a dense matrix. The compiler will automatically optimise these operations according to the sparsity described in these directives.

Listing 1.3: Example of dense code for the sparse compiler.

```
       REAL A(N,N)
C_SPARSE(ARRAY(A) , ZERO  (1-N<=I-J<= -6)(1,1))
C_SPARSE(ARRAY(A) , DENSE ( -5<=I-J<=  5)(1,1))
C_SPARSE(ARRAY(A) , ZERO  (  6<=I-J<=N-1)(1,1))
      ... operations on A
```

This technique has many advantages in terms of modularity. It is easier for programmers to try different storage schemes or ways of characterising sparsity (for example a triangular matrix is a particular case of a banded matrix, and both schemes can be tested). The readability of the operations performed on the matrices is also improved: the particular treatment of sparse matrices does not interfere with the actual purpose of the algorithm. Moreover, the resulting code is less error-prone since it is less confusing, better separated, and thus easier to debug.

Having to deal explicitly with sparse routines, without this sparse compiler, would be made difficult by the large amount of tangled code. However, although

the sparse compiler is a first and welcome step towards the elimination of code-tangling, the annotations for dealing with sparse matrices are still interlaced in the algorithm representing the main concern. The sparse properties are not an explicit part of the data structure, and where they apply has to be specified explicitly.

1.3.4   Comparing three versions of the same application

In order to evaluate the performance of Java in scientific applications, the EPCC group [EPC] have developed the *Java Grande benchmark suite* [BSW$^+$00, SB01, SBO01, BSPF01]. Three categories of benchmarks are available on their website: the sequential category, the multi-threaded category—which uses Java threads [OW99]— and the MPJ category—which implements parallelism using mpiJava, a Java implementation of MPI [BCF$^+$98, CGJ$^+$00].

Each category is divided into three sections: section 1 ("Low level operations") is for measuring the performance of low level operations; section 2 ("Kernels") tests short codes that implement specific operations that are frequently used; and section 3 ("Large scale applications") tests code for larger real applications. Whereas programs in section 1 are specific to each category of the benchmark suite, most of the examples in section 2 (Crypt, LUFact, Series, SOR and SparseMatmult) and in section 3 (MolDyn, MonteCarlo and RayTracer) are common to the three categories, each example implementing parallelism with the method appropriate to the suite to which it belongs.

In the sequential category, the parallelism concern is not implemented. The two other categories have extra statements and a slightly different structure for implementing parallelism. Figure 1.1 highlights the statements that implement parallelism in the MPJ version of the RayTracer application. These statements are scattered throughout several units and interlaced with the statements of the functional units (the reference design and implementation for the functional units is the sequential version). The multi-threaded suite is implemented using a design similar to the one shown in Listing 1.1.

This code-tangling makes it difficult to isolate the concerns involved. In this example, the computation concern and the high-performance (by parallel execution) concern cannot be studied separately. Moreover, any evolution of the computational model would require modifications to all three categories (sequential, multi-threaded and MPI). In theory, modifying three versions of the same application should not be a problem. However, this process is error-prone and better
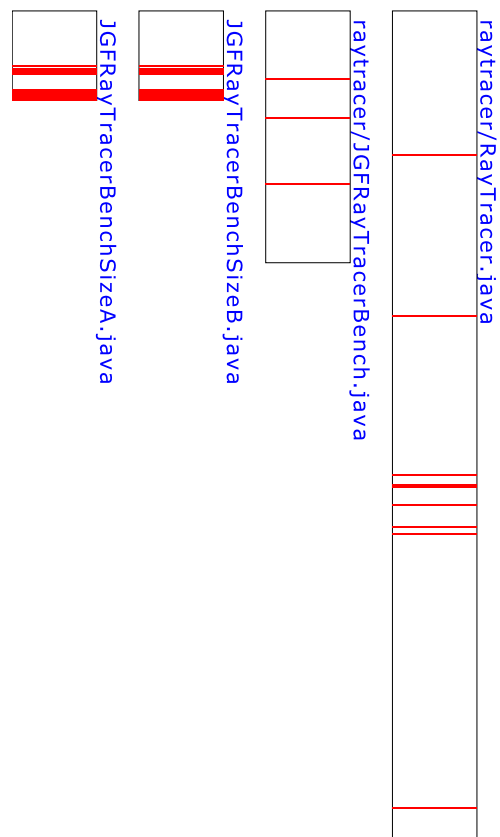
The length of these rectangles is proportional to the number of lines in the files. The highlighted lines are those that contain MPI statements. These statements are spread across four files and located at non-contiguous places. The statements that are not highlighted have not been modified from the sequential version. This example is typical of many of the Java Grande benchmark suite codes.

Figure 1.1: Crosscutting due to MPI statements in the MPJ version of RayTracer.

separation of concerns would have promoted consistency in the evolution of the application.

1.3.5   Separation of concerns in scientific software

There are other examples where scientific code suffers from code-tangling and code-scattering. For instance, many linear algebra systems of equations arise out of scientific problems and are required to be solved. In most cases, these systems involve large matrices that contain many zero-valued coefficients (i.e. sparse matrices). Specific algorithms have been developed to deal with sparse matrices. For linear-algebra applications, two or three concerns can be identified: the means of resolution of the system (by Gaussian pivot, for example), the sparsity of the matrix, and possibly the parallelisation of the algorithm. The resulting piece of software, if written using a procedural or an object-oriented language such as Fortran or C++, will have interlaced statements for these three concerns. Even though this software can be efficient, such a design does not favour adaptation to another kind of sparsity (for example, from upper triangular to banded) or to another kind of parallelisation (for example, suitable for another architecture).

According to Dijkstra, *"the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects"*[5] [Dij76, ch. 27]. This principle, which consists of being able to focus on one facet of a problem at a time, is referred to as the principle of *"separation of concerns"* [Dij76, ch. 27]. The general aim of this thesis is to provide computer-literate scientists with a means of enforcing a better separation of the concerns related to the mathematical models from the concerns related to performance, by providing adequate abstraction mechanisms for implementing concerns. The context in which this is attempted is that of the object-oriented Java programming language.

1.4   Java-based numerical computing

"Java" consists of two parts: the Java language [GJSB05] and the Java virtual machine (JVM) [LY99]. In general, programs written in the Java language are executed on the JVM, and the JVM is mainly used for running programs written

---

[5]The term *aspect* is not used here as part of the Aspect-Oriented Programming terminology.

in Java. Their respective designs have influenced each other; however, they do not necessarily need each other. On the one hand, Java code can be compiled into native code for a particular architecture, for example with GCJ—the GNU Compiler for Java [gcj]. On the other hand, Java byte-code can be generated from other languages and can be run on the JVM [Tol, GC00]. In particular, one of these languages is AspectJ (an aspect-oriented extension to Java[6]).

Java technology, including both language and virtual machine, has become increasingly popular over the last ten years. The JVM and its standard API[7] have made Java applications highly portable. Moreover, the programming language is relatively easy to use, especially since it relies implicitly on the virtual machine for memory management and security. These advantages have made Java the technology of choice for a large number of applications, in a wide range of domains.

Several standard features, such as portability, the networking API and built-in multi-threading, have made Java particularly attractive for scientific programming. The question of its suitability as both a programming language and a running environment for numerical computation has been raised several times [CCFL98, Art00, MMG+00, BMPP01, Thi02]. This has become more pertinent since the appearance of virtual machines with *just-in-time* (JIT) compilers, such as HotSpot [Hot], which have increased the performance to an "acceptable" level.

The Java Grande Forum (JGF) [FSS99, GM00, CRP01, MFG02] "*[aims to] develop community consensus and recommendations for either changes to Java or establishment of standards (frameworks) for Grande libraries and services*" [JGF]. A *Grande* application "*is any application, scientific or industrial, that requires a large number of computing resources*" [Pan98]. The JGF has identified five critical issues related to the use of Java for large computational problems:

- *Multidimensional arrays* are currently represented as arrays of arrays;

- *Complex arithmetic* is not supported with a primitive type;

- Lack of *lightweight classes*;

- Using *floating-point hardware* is at odds with Java's portability;

---

[6]See Section 2.1.3 and Appendix A

[7] The Java development kit (JDK) and runtime environment (JRE) provide the standard Java API.

- *Operator overloading* is not possible.

These issues have been discussed [MMG⁺00, BMPP01], and some solutions have been proposed. Since Java 1.2, floating point operations can take some advantage of the underlying hardware.

Evolution towards a Java environment for high-performance computing, in both the language and specifications, and the JVM implementations, is encouraging. On the one hand, the Java Community Process[8] has been set up to address the evolution of the specifications of Java (for both the language and the virtual machine), via Java Specification Reviews (JSRs); but each review often takes several months. On the other hand, there has been a clear improvement in performance each time a new JVM has been released, especially on PCs [BMPP01].

However imperfect Java is for high-performance computing, this is the platform of choice for most of the supported aspect-oriented tools.[9] Therefore, for practical reasons, the experiments, the tools and the methods developed throughout this thesis are Java-based. The performance results presented in the thesis depend heavily on the JVM environment within which the test-cases have been executed.

1.5 Challenges and contributions

1.5.1 Aspects for parallel computing

Decoupling numerical models from the expression of their parallelisation is the main challenge addressed throughout this thesis. The first main contribution of the thesis is to provide methods that make aspects capable of achieving this goal, as presented in Chapters 3 and 4. More practically, aspects that implement diverse parallelisation strategies are shown in Chapter 5. The flexibility induced by these aspects makes parallelisation a pluggable unit, re-usable across applications.

1.5.2 AspectJ and beyond: join points for complex behaviours

The main problem encountered for applying aspect-orientation to scientific software is to identify the adequate abstractions for the domain. In AOP, Filman *et al.* define the *abstractness constraint* as follows. "*The constructs of an aspect-oriented*

---

[8]`http://www.jcp.org/`

[9]During the investigation that has led to the thesis, AspectC++ (`http://www.aspectc.org/`, [SGP02]) has gained momentum and support, and version 1.0 is about to be released.

programming language must be abstract enough to match the natural abstractions of the problem domain. However, they also must be concrete enough to match the realization of the implementation platform. This constraint aims to minimize the implementation effort and enable efficiency" [FECA04, Part 1].

The abstractions that AspectJ can match and at which it can intervene are its *join points*. In the domain of scientific computing, one natural abstraction is the *loop*. However, the current AspectJ join points are limited to relatively simple Java behaviour and, in particular, do not include loops. The second main contribution of this thesis is to provide AspectJ with a join point that correspond to a complex behaviour in the AspectJ model: the join point for loops. Chapter 4 provides AspectJ with a new join point that enables aspects to intervene directly at loop-level.

1.6   Outline

Chapter 2 gives an introduction to aspect-oriented programming, which is the approach used to address the problem, and presents the background work, prior to this thesis, about aspects and performance.

After this, Chapter 3 presents techniques for using AspectJ for writing aspects capable of parallelising Java applications; this is done by means of refactorings. Chapter 4 goes a step further and provides AspectJ with a join point for loops, so as to avoid any refactorings. Application examples and an evaluation of the performance of the two approaches are presented in Chapter 5. Finally, Chapter 6 concludes.

Chapter 2

Aspect-Oriented Programming

This chapter introduces Aspect-Oriented Programming (in Section 2.1) and presents background work, prior to this thesis, about aspects and performance (in Section 2.2).

2.1   Introduction to Aspect-Oriented Programming

This section is an introduction to Aspect-Oriented Programming (AOP). First, Section 2.1.1 presents the motivation for this relatively new technology. Section 2.1.2 presents the main concepts behind AOP and their novelty. Finally, Section 2.1.3 gives an overview of current AOP tools and implementations.

2.1.1   Motivation

Object-Oriented Programming (OOP) [Mey88] is probably the most commonly used programming paradigm today. The evolution from assembler language to current software engineering paradigms reflects the will for better readability and re-usability—or, more generally, better organisation— in designing applications.

As discussed earlier, functional, procedural and object-oriented programming languages have a common way of abstracting and separating out concerns: they rely on explicitly calling subprograms (subroutines, procedure, methods, *etc.*) that represent *functional units* of the system [KLM⁺97]. However, not all concerns can be encapsulated properly in a functional decomposition. For example, tracing and logging are concerns that are usually distinct from the functional units they are related to (i.e. the units whose behaviour is traced or logged). As a result, they must be coordinated with other functional units and they usually involve code scattered

throughout several of these functional units. Aspect-Oriented Programming aims at better separation of concerns by providing the *aspect* as a means to encapsulate such crosscutting concerns.[1]

(Figure source: [EAK⁺01])
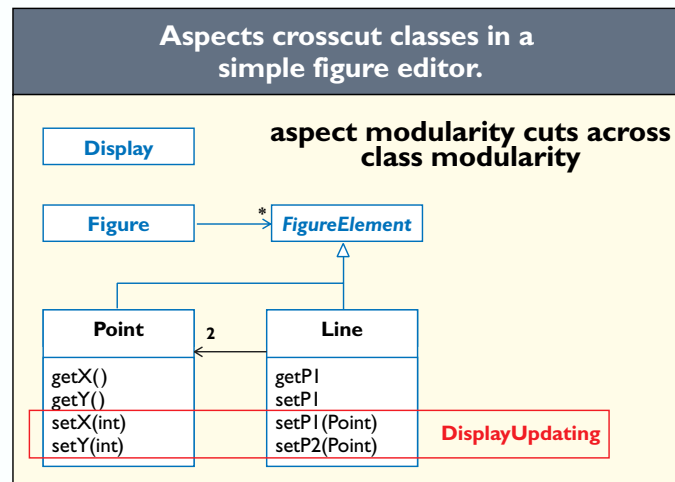
Figure 2.1: Example of aspects in a figure editor.

A short example of crosscutting concerns and use of AOP is shown in Figure 2.1.[2] The figure shows the classes used in a simple figure editor: *"a Figure consists of a number of FigureElements, which can be either Points or Lines. [...] There is a single Display on which figure elements are drawn"* [KHH⁺01a]. Methods setX and setY of the FigureElements involve two separate actions: they must update the coordinates in their target object and they must trigger the redrawing of the display. Updating coordinates X or Y of the target is intrinsic to the object and clearly corresponds to, respectively, methods setX and setY. However, the concern of updating the display has to be handled after executions of either setX or setY. Thus, the display-update concern crosscuts two methods within the same class. This concern must also be applied in several classes. Updating the display is a concern that is not directly related to the FigureElement model. One can imagine another system, using the same FigureElement model, which would render elements of this model on a printer instead of a screen. It is possible for all the statements related to the display to be embedded in the code for

---

[1]As explained in Section 1.2.2, the notion of crosscutting is relative to the decomposition of the other concerns. In particular, a crosscutting concern is one that crosscuts with respect to at least one other concern.

[2]This example is taken from [KHH⁺01a] and [EAK⁺01].

the `FigureElement` model. However, in that case, transforming the application so that it uses a printer would require modifications in diverse parts of the code, because the concern of updating the display would crosscut the model and would not be encapsulated in its own entity. An aspect-oriented implementation of this application would encapsulate these kind of concerns into aspects, making it easier to read, re-use and adapt the code in other contexts. The principles of such an implementation are described in the next section.

2.1.2   Concepts

Aspect-Oriented Programming aims to be able to link together concerns that cut across each other, and yet encapsulate them transparently as separate program entities. The general style of programming that arises out of this aim consists of program statements of the form:

> *"In programs P, whenever condition C arises, perform action A."* [FF00]

This leads to the following definitions: a *join point* is a point in the structure or in the execution of a program where a concern crosscutting that part of the program might intervene. Join points are the points that can be used to express potential conditions C in programs, according to the above formulation. Join points can be seen as hooks in a program where other program parts can be conditionally attached and executed.

A *pointcut* is a subset of all possible join points. The expression of a pointcut is the *pointcut descriptor* (often, the term "pointcut" is used in place of "pointcut descriptor"). A pointcut descriptor defines the condition C in the above formulation. This condition matches a subset of join points which is the pointcut. The piece of code A that is to be executed when condition C arises (i.e. at a join point of the pointcut) is called the *advice*.

The unit of code that defines the pointcuts and the advice related to the same concern is called the *aspect*. An aspect can also be more generally defined as a unit that encapsulates a crosscutting concern[3].

---

[3]Although it would be possible to encapsulate in the same unit sets of pointcuts and their associated advice that are not related to the same concern, this would be against the main principle of AOP, which aims to make it possible to separate concerns by encapsulating crosscutting concerns each in their own entity.

The counterparts of aspects are *components* or *base code*, which are the functional units of code that do not contain aspect-oriented statements but only base actions. Components are units of code as written using functional, procedural or object-oriented languages. To some extent, the advice in aspects could be considered as a component, within which aspects could intervene. However, the problems that can arise out of interaction between aspects lie beyond this introduction; they are the subject of continuing research.

In the example shown in Figure 2.1 (described at the end of Section 2.1.1) the `FigureElement` model is an example of a component. The update of the display is implemented in an aspect. A pointcut descriptor in this aspect expresses the points in the execution flow after returning from methods `setX` or `setY`, and the piece of advice associated with this pointcut then updates the display.

Mixing components and aspects together, so that the behaviour specified by the aspects occurs where and when it is supposed to, is the process of *weaving*. Some implementations use a specific type of compiler, called a *weaver*, to generate an executable from components and aspects. Other implementations perform the weaving at runtime or load-time, using mechanisms equivalent to a runtime form of compilation. The details of these implementations are out of the scope of this introduction. However, some examples of aspect-oriented languages and frameworks are given in the next section.

AOP is not about writing macros or inserting code at some given line number. Rather, it is about applying certain actions when a specifiable behaviour happens. Thus, mechanisms for aspect orientation rest on three pillars[4]:

- a model of the behaviours that can be recognised and exploited (the join points),

- a means of characterising a subset of these possible behaviours (the ability to define pointcuts),

- a means of implementing the behaviour defined in the aspects at the place and at the time that the expected behaviour defined in the pointcuts happens (the weaving of the advice).

The components need not be aware of the effect of the aspects to which they are subject. In particular, components may be prepared so as to be subjects to aspects,

---

[4]These three pillars are what Kiczales *et al.* describe in [KHH+01a] as the "*three critical elements [that] AO languages have*".

for example via annotations or refactorings, but should not be prepared in a manner that would couple them tightly with the behaviour of any potential aspect to which they might be subject.[5] This notion of *obliviousness* is one of the main assets of AOP for improving the flexibility of software development. This means that, in some cases, the integration of certain aspects into a final version of the code is optional. However, failure to integrate some aspects might completely change the behaviour of the application concerned. For example, an aspect that would check the consistency of some data may be necessary to prevent faults, whereas an aspect that would be used by the programmer for debugging rarely needs to be integrated into the final version of a project.

Reasoning about aspects is still an open problem. Filman and Friedman proposed that *"better AOP systems [should be] more oblivious. They minimize the degree to which programmers (particularly the programmers of the primary functionality) have to change their behavior to realize the benefits of AOP"* [FF00]. However, full obliviousness has proven to be difficult to achieve in practice. Decoupling crosscutting concerns from the base system gives benefits in term of readability, but full obliviousness can prevent the programmer of the advised units from knowing what will happen when these units are utilised, and in most cases will not be free of undesired side-effects. Even in mainstream AOP languages such as AspectJ (see Section 2.1.3), tools have been developed to assist programmers in knowing the interactions between the aspects and the components. More recently, Kiczales and Mezini [KM05] proposed a different way of reasoning about aspects, in which *"aspects cut new interfaces through the primary decomposition of a system. This implies that in the presence of aspects, the complete interface of a module can only be determined once the complete configuration of modules in the system is known. While this may seem anti-modular, it is an inherent property of crosscutting concerns [...]"*. Aspects change the concepts of modules as they are used in procedural and object-oriented languages, but provide the ability to view and to reason about cross-sections of the system. Related work on aspects and modularity includes [Cli05], [Ald05] and [SGS⁺05].

---

[5]When using annotations, these should reflect characteristics of the components rather than implementation details regarding the aspects. The latter would imply code-tangling equivalent to that of using compiler directives, as shown in Section 1.3.

2.1.3 Languages and tools

Several aspect-oriented languages or frameworks have been developed. Aspects must use a language to describe the actions to be performed by their advice. Obviously, the components into which the aspects are woven are also written using a language. The language used for both the components and the advice is usually the same, and this is usually a general language such as C or Java. Most of the aspect-oriented languages and tools are based on procedural, functional or object-oriented languages that exist outside AOP. A list of aspect-oriented languages and tools can be found on the Aspect-Oriented Software Development (AOSD) web-site [AOS].

One of the most popular and most mature aspect-oriented languages is AspectJ [asp, KHH+01a]. The AspectJ project started at the Xerox Palo Alto Research Center. Its leading researchers published some of the founding articles on Aspect-Oriented Programming [KLM+97]. AspectJ is an aspect-oriented extension to Java. It uses regular Java statements to write the advice, but it defines a few specific constructs for encapsulating aspects and for writing pointcuts. A summary of the syntax can be found in Appendix A. AspectJ uses a specific compiler (a weaver), which produces standard bytecode that can be executed on any Java virtual machine.

AspectJ mainly works on the interfaces of the classes. The basic join points that AspectJ can use are calls to methods, executions of methods, accesses to fields, instantiations of objects and executions of exception handlers. Pointcut descriptors are then written as logical expressions defining which of these join points have to be picked out. The selection is based on the name of the objects and methods involved and on their signature, using regular expressions. It is also possible to refine the conditions, for example by selecting certain calls only if they are called from within the control flow of a particular method. The advice can be executed before, after or around these pointcuts.

Aspects in AspectJ can be compared to classes in Java. AspectJ defines the `aspect` keyword for declaring aspects. These aspects can contain pointcut descriptors, advice and even regular Java fields and methods (which can be used by the advice). Two examples of AspectJ's syntax are shown in Listings 2.1 and 2.2. The full reference and programming guide on AspectJ can be found on the official AspectJ web-site [asp].

Listing 2.2 shows an example of *around*-advice. This kind of advice will be used for parallelisation in the following chapters. An *around* piece of advice replaces the execution of the intercepted join point. One of the principal AspectJ constructs for *around*-advice is "`proceed`". This keyword can only be used in *around* pieces of advice. As its name indicates, it proceeds with the execution of the join point that was intercepted. The main feature of `proceed` is that it takes arguments that must match the parameters of the piece of advice. These arguments can be modified before going ahead with the execution of `proceed`. In this example, `proceed(v)` will execute the join point that consists of setting `Test.value`. However, `proceed` replaces the values of the original pointcut arguments by its own arguments. In this example, if `v` > `AUTHORISED_MAX`, the value `AUTHORISED_MAX` is going to be passed to `proceed`, and the execution of the join point that corresponds to setting the value of `Test.value` is going to use `AUTHORISED_MAX` for the new value, instead of the original value of `v`. Moreover, the use of `proceed` is optional; not using it implies that the join point advised is not going to be executed.

Along with compilers such as that for AspectJ, there exists a set of tools to assist developers and designers in using aspect-oriented technologies. Eclipse[6] is a development environment that fully supports AspectJ via the AspectJ Development Tool (AJDT) [CCHW05], an optional plug-in. It can assist the development of AspectJ projects with various features, such as showing graphically where the aspects are woven in. Since development environments are often a matter of taste, there exist similar AspectJ plug-ins for Emacs, JBuilder and Netbeans. Other tools, such as "Aspect Browser"[7], can help find crosscutting in existing Java projects by producing representations similar to Figure 1.1.

2.2   Performance as an aspect

The aspect-oriented programming community has often cited performance as an example crosscutting concern that could be encapsulated in an aspect. Most of the recent work on performance with AOP consists of intervening at a coarse level, for example by caching network transactions. This approach is too coarse for most problems in scientific computation. This section reviews the few publications related to the use of aspects for improving performance while achieving good

---

[6]Eclipse is an open-source project and can be downloaded from `http://www.eclipse.org/`.

[7]Aspect Browser can be obtained from `http://www-cse.ucsd.edu/users/wgg/Software/AB/`.

Listing 2.1: Example of aspect, using AspectJ (*before* and *after* advice).

```
/**
 *  This simple class only contains a field.
 *  It can be used in various places in a larger program.
 */
public class Test {
   public int value = 0 ;
}



/**
 *  This short aspect prints out ''The value is going to be modified.''
 *  each time the ''value'' field of any instance of ''Test'' is about
 *  to be assigned.
 */
aspect SimpleTracing {
   /* This pointcut picks out all the points where the ''value''
      field of any instance of ''Test'' is modified.
    */
   pointcut modifyingValue(): set(int Test.value) ;

   /* This piece of advice is executed before each join point picked
      out by the pointcut defined above. The body of the piece of
      advice is written as the body of a Java method would be.
    */
   before(): modifyingValue() {
      System.err.println ("The value is going to be modified.") ;
   }
}



/**
 *  This aspect is similar to the aspect above, but it gets the
 *  context and prints out the new value as well.
 */
aspect TracingWithContext {
   pointcut modifyingValue(): set(int Test.value) ;

   /* This piece of advice also gets the argument to the setting of
      the new value. This element of dynamic context can be used
      from within the piece of advice.
    */
   after(int v): modifyingValue() && args(v) {
      System.err.println ("The new value is: "+v) ;
   }
}
```

Listing 2.2: Example of aspect, using AspectJ (*around* advice).

```
/**
 *  This simple class only contains a field.
 *  It can be used in various places in a larger program.
 */
public class Test {
   public int value = 0 ;
}


/**
 *  This aspect enforces a maximum value for the ''value''
 *  field of any instance of ''Test''.
 */
aspect SaturateValue {
   pointcut modifyingValue(): set(int Test.value) ;

   /* This piece of advice is executed instead of the action
    * of setting the ''value'' field of any instance of ''Test''.
    * ''proceed'' executes the intercepted join point, but
    * replaces the value of the argument to ''set''.
    */
   void around(int v): modifyingValue() && args(v) {
      if (v > AUTHORISED_MAX)
         proceed(AUTHORISED_MAX) ;
      else
         proceed(v) ;
   }
}
```

program readability, in the context of numerical computing. Section 2.2.1 and Section 2.2.2 present the publications relative to the use of aspects, respectively, for loop fusions and for sparse matrix codes.

2.2.1 Aspects for loop fusion

An image processing system is described in the first major paper on AOP [KLM$^+$97] (and in more detail in [MKL97]). The goal—or the main concern—of this application is to apply transformations (filters) to black-and-white images. These images can be large, and so a subsidiary goal is performance, achieved by using the memory efficiently. The aim is to design software that will both satisfy these two concerns and be easy to develop and maintain. Using the memory efficiently (thereby improving performance) can be done by merging loops. This merging optimisation can occur, for example, when several simple filters are combined to form a complex filter: if possible, the loops that iterate through the pixels of the images are merged.

This image processing application, in both non-aspect-oriented and aspect-oriented versions, was written in Lisp. In the non-aspect-oriented version, the simple filters were written as procedures (see Figure 2.2) and the complex filters were optimised by merging the loops manually. The complex filters are described in Figure 2.3. This manual loop-merging optimisation gives rise to tangled code (as shown in Figure 2.4), which makes it more difficult to understand how these complex filters have been composed from simple filters.

```
(defun or! (a b)                              loop over all
   (let ((result (new-image)))              the pixels in the
      (loop for i from 1 to width do           input images
         (loop for j from 1 to height do
the operation     (set-pixel result i j
to perform on ——(or (get-pixel a i j)
the pixels              (get-pixel b i j)))))    storing pixels in
         result))                            the result image
```

(Listing source: [KLM$^+$97].)
This listing represents the or! operation (which takes two black-and-white images and creates a new image that represents their pixel-wise logical *or*) implemented as a procedure, in the simple version of the application.

Figure 2.2: Procedural implementation of a simple filter.

functionality	implementation
pixelwise logical operations	written using loop primitive as above
shift image up, down	written using loop primitive; slightly different loop structure
difference of two images	`(defun remove! (a b)` `  (and! a (not! b)))`
pixels at top edge of a region	`(defun top-edge! (a)` `  (remove! a (down! a)))`
pixels at bottom edge of a region	`(defun bottom-edge! (a)` `  (remove! a (up! a)))`

horizontal edge pixels	`(defun horizontal-edge! (a)`
	`  (or! (top-edge! a)`
	`       (bottom-edge! a)))`

(Source: [KLM+97].)

This table describes a few complex filters in terms of simple filters. This is the non-optimised version of the procedural implementation. In this version it is easier to read the composition of the `horizontal-edge!` filter than in the optimised version shown in Listing 2.4.

Figure 2.3: Description of non-optimised complex filters.


```
(defun horizontal-edge! (a)                         only three result
  (let ((result (new-image))                      images are created
        (a-up (up! a))
        (a-down (down! a)))
    (loop for i from 1 to width do            one loop structure
      (loop for j from 1 to height do          shared by many
        (set-pixel result i j                  component filters
          (or (and (get-pixel a i j)
                   (not (get-pixel a-up i j)))
              (and (get-pixel a i j)
                   (not (get-pixel a-down i j)))))))))
    result))
```

operations from many sub-filters
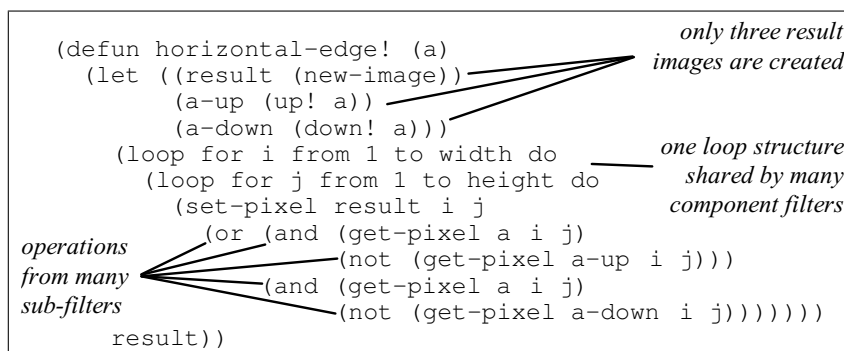
(Source: [KLM+97].)

This is an optimised version of the `horizontal-edge!` filter described in Figure 2.3. This version is tangled, and it is difficult to see the composition of this complex filter.

Figure 2.4: Optimised version of a complex filter.

In the aspect-oriented version, the expression of complex filters is kept as simple as possible and does not include their optimisation. Instead, fusing the loops for improving performance is expressed as an aspect. The expression of filters in the aspect-oriented version presents a few differences to the procedural version. "*First, filters are no longer explicitly procedures. Second, the primitive loops are written in a way that makes their loop structure as explicit as possible*" [KLM+97].

```
(define-filter or!  (a b)
(pixelwise (a b) (aa bb) (or aa bb)))
```

(Source: [KLM+97].)

This listing represents the same operation as in Figure 2.2 implemented for the aspect-oriented version. This second implementation uses the separately defined construct "pixelwise", which is an iterator that "*walks through images* a *and* b *in lockstep, binding* aa *and* bb *to the pixel values, and returning an image comprised of the results*" produced by the operation defined in its third argument [KLM+97].

Figure 2.5: Implementation of a simple filter in the aspect-oriented version.

This change of abstraction, illustrated in Figure 2.5, gives rise to shorter pieces of code by using powerful keywords. These keywords are easier to recognise while encompassing a complex meaning. For example, "pixelwise" is another abstraction for the two nested loops that represents the iteration through every pixel of an image (such as in Figure 2.2). It is easier to recognise the "pixelwise" construct (see Figure 2.5) than to analyse a block of instructions and to recognise these two nested loops. This allows the weaver to analyse the components (in this case, the filters) and apply the code transformations encapsulated in the aspects, which are also written in Lisp. The readability is improved on two fronts: the functional units are written using a simpler (but semantically richer) abstraction, and the aspects represent the optimisation separately and thus avoid the tangling of the code.

Although this example is not really a scientific application, the loop fusion technique is relevant to the kind of optimisations that can be generally envisaged in scientific software. The key to the success of this Lisp-based aspect-oriented example (originally presented in [KLM+97, MKL97]) is the combination between the adequate abstraction of the components and the ability that the aspects and the compiler have for efficiently exploiting this abstraction. This satisfies the three

pillars presented in Section 2.1.2. In this AOP prototype, the fact that Lisp is an interpreted language that can delay the evaluation of some parts of the program makes it easier for the aspects to be woven in the components.

2.2.2   Aspects for sparse matrix code

Aspect-oriented programming has also been used for performance in another branch of numerical software, in a project related to sparse matrix codes [ILG$^+$97]. This project relies on its own specific language: AML (annotated Matlab[8]). This language is similar to Matlab, but also allows the programmer to write annotations that represent properties of sparse matrices. AML also extends the original Matlab syntax with a new "for"-like construct (`for nzs`) that is meant to iterate only through non-zero values of vectors. Two "aspects" can be used in AML: the data representation format (to hide the complexity of the sparse matrix algorithms) and the implicit permutations (since some operations require matrices to be presented in a certain form, which can be obtained by multiplying by a permutation matrix). These aspects facilitate the writing of sparse matrix algorithms as if these matrices were dense. They are related to performance to the extent that sparse matrix code avoids unnecessary computation, and they thus improve the performance of the application.

Figure 2.6 shows an example of AML code that performs the LU factorisation. This consists of decomposing matrix A into matrices L and U, so that $A = LU$, with L and U being respectively lower and upper triangular matrices. Extra constructs are introduced to declare matrices as sparse and to use implicit permutations. These annotations make it possible to hide the complexity of the code that would have been required for dealing explicitly with the sparsity of the matrices. Here, for example, "`view A through p`" is an abstraction that hides the fact that A has to be permuted from its sparse storage form into its usable form. This is done implicitly between "`view`" and "`end view`".

This work was originally presented as being aspect-oriented [ILG$^+$97]. However, the use of annotations embedded in the functional units, and the fact that the "aspects" (the permutations and the data storage format) are not flexible but pre-defined in the weaver, conflict with the principle of transparency promoted by AOP. Thus, this project has since been rejected as not being aspect-oriented by

---

[8]Matlab is a language dedicated to mathematics, and more particularly to matrices.

the group at Xerox PARC [Lop02, Lop04], which includes some of the authors themselves.

The use of annotations for sparse matrices can actually be compared to (non-aspect-oriented) compiler directive techniques, such as those presented in Section 1.3, in particular to the use of compiler directives in Fortran proposed by Bik *et al.* [BBKW98] and presented in Section 1.3.3.

```
function [L,U,p] = lu(A);
declare real sparse matrix A, L, U;
declare real        scalar v;
declare int         scalar m, n, j;
[m,n] = size(A);
L = zeros(n,n);
U = zeros(n,n);
declare permutation p;
p=[1:n];
view A,L,U through (p,:)
    for j = 1:n
        declare SPA t;
        view t through p
            t = A(:,j);
            for nzs k in order in t(1:j-1)
                t = t - t(k)*L(:,k);
            end;
            [v,piv] = max(abs(t(j:n)));
            piv = piv+j-1;
            p([j,piv]) = p([piv,j]);
            U(1:j,j) = t(1:j);
            L(j+1:n,j) = t(j+1:n)/t(j);
        end view;
    end;
end view;
```

The data representation declarations have been marked with a single bar to the left. Double bars mark code that deals with implicit permutations, including code in the component language that adjusts the permutation.

(Source: [ILG+97].)

Figure 2.6: Example of AML code: LU factorisation.

Even if it is not aspect-oriented, the "`for nzs`" construct in AML illustrates the need for an appropriate abstraction in the base code, recognisable via the "aspects".

2.3 Summary

This chapter has introduced the motivation for and main concepts of Aspect-Oriented Programming. It has also presented AspectJ, an aspect-oriented extension to Java, illustrated with a short example. Further information about AOP and related tools can be found on the AOSD web-site [AOS] and in the special issue of *the Communications of the ACM* dated October 2001 [EFB01, EAK⁺01, LOO01, OT01, BA01, KHH⁺01a, PC01, MWB⁺01, CKF⁺01, NEF01, GBNT01, Sul01, NEF01, GBNT01, Sul01]. During the course of this project, several books or book chapters on AOSD have been published. Some are specific to AspectJ [CCHW05, Lad03], and others are more general, such as [PRS04]⁹, [FECA04], and [Mon05, ch. 11].

The ability to characterise and recognise certain behaviours lies at the root of AOP. Aspect-oriented implementations can be successful if the join points, at which the aspects intervene, can be clearly expressed and characterised. This characterisation is in terms of behaviour and abstraction of the components, rather than by means of explicit mark-up annotations which are tightly coupled to the implementation.

Explicit mark-up annotations or compiler directives, as shown in Section 2.2.2, are more a means for expressing complex concerns in compact abstractions rather than a means to separate concerns. Annotations neither entail the automatic recognition of certain behaviour nor possess the flexibility of the aspect program.

The two examples presented in Section 2.2 come from articles published in 1997. Since then, little work on numerical optimisation has been done in the aspect-oriented software community. Performance has been mentioned in several publications as an example of a type of aspect. However, most of the more recent work on performance consists of improving performance at a coarser level, for example by writing aspects to handle caching of network transactions, or caching successive requests to complex methods [DHS⁺03]. AOP has also been used for (coarse-grain) performance monitoring [DHS⁺03, Bod05] and profiling [PWBK05].

The loop fusion example (in Section 2.2.1) shows that AOP can be used in the context of numerically intensive applications such as scientific software. Unfortunately, it seems unrealistic to envisage that programmers of scientific software will move towards a language such as Lisp.

---

⁹ This book is in French; a similar book by the same authors is available in English [PSR05].

Chapter 3

Join points for parallelism in AspectJ

The work presented in this chapter has led to the publication of an article presented at the *3rd International Conference on Aspect-Oriented Software Development (AOSD'2004)* [HG04].

The join points that can be exploited by AspectJ (one of the most popular aspect-oriented extensions to Java, see Section 2.1.3) are located at the interface of the components (packages, classes or methods). They consist mainly of method calls (and variations, such as calls to constructors) or field accesses. However, the constructs that are usually interesting for parallelising an algorithm are "for" loops and array accesses. Unfortunately, AspectJ does not recognise these constructs as join points.

This chapter investigates how to circumvent this limitation. Section 3.1 shows how AspectJ can be used to implement parallelism in scientific applications using example codes from the *Java Grande Forum benchmark suite* [BSW$^+$00, SB01, SBO01, BSPF01]. It is found that the underlying abstractions for describing the numerical concern can cause significant problems when writing aspects aimed at parallelisation, and so Section 3.2 proposes object-oriented models for describing loops, within which aspects can be woven.

3.1   Aspects for the Java-Grande Forum benchmark suite

As has been shown in Section 1.3.4, parallelising Java applications can be achieved by various means, but often leads to code-tangling. The Java Grande Forum

benchmark suite (see Section 1.3.4) comprises a set of applications that come in three versions:

- the *sequential version*, which is aimed at single-processor machines;

- the *multi-threaded version*, which is parallelised using Java threads; and

- the *MPJ version*, which is parallelised using a Java version of MPI.

Some applications are provided in all three versions; the only difference between the three are due to the extra statements introduced in the parallelised versions. As has been shown in Figure 1.1, these extra statements are interlaced in the code that describes what is to be computed, which corresponds to the sequential version.

This section investigates how AspectJ might be used to encapsulate the means of parallelising the application in each version. The aim is to provide two aspects that, if woven into the sequential version, would produce, respectively, the MPJ version or the multi-threaded version.

The amount of refactoring required in the sequential version varies across the benchmark suite codes, ranging from minor to major; three cases are taken as representative of the extremes (Sections 3.1.1 and 3.1.2) and the middle ground (Section 3.1.3).

3.1.1   Minor refactoring

The first chosen test case is the *Crypt* application (in Section 2 of the benchmark suite). This consists of 5 Java files: `JGFCryptBenchSizeA`, `JGFCryptBench-SizeB`, `JGFCryptBenchSizeC`, `IDEATest.java` and `JGFCryptBench.java`, and uses classes of the provided `jgfutil` package, such as `JGFInstrumentor` (which handles the timers and displays the results). It requires little modification before aspects can be used for parallelisation.

The computationally intensive part of the program is the loop in method `void cipher_idea(byte[] text1, byte[] text2, byte[] key)` in class `IDEATest`. This method takes the data in parameter `text1`, enciphers it with the key in parameter `key` and stores the results in the array defined by parameter `text2`. The method in the sequential version is written in the form shown in Listing 3.1.

Since each iteration of this loop is independent from all others (i.e. the loop is *embarrassingly parallel*), it is possible to spread the computation across several

processors by splitting the range of the loop index `i` into blocks. For example, the multi-threaded version of *Crypt* is written so that there can be several instances of the loop that start with $i = i_{low}$ and stop when $i \geq i_{upper}$, for appropriate (distinct) values of i_{low} and i_{upper}, as shown in Listing 3.2.

Listing 3.1: Implementation of `cipher_idea` in the sequential version.

```java
private void cipher_idea(byte[] text1, byte[] text2, int[] key) {
    /* Declaration of local variables and initialisations */
    ...

    for (int i = 0; i < text1.length; i += 8) {
        /* Body of the loop */
        ...
    }
}
```

In order to make it easier to write an aspect that can intercept the original calls to `cipher_idea` and partition the loops, the sequential version, into which aspects for parallelism may be woven, is refactored so as to allow both the original use (from 0 to the total length of the text) and the use of blocks (from i_{low} to i_{upper}). The *iteration space* of the algorithm is thus accessible and modifiable from the object interface, and can be partitioned by a parallelism aspect. This small refactorisation is shown in Listing 3.3. This is the only modification that needs making to the sequential version, and it neither breaks the original decomposition nor introduces any code-tangling. After this, it is possible to write an aspect for either parallelisation scheme (MPJ or multiple threads). These aspects implement parallelism transparently with respect to the components that implement the calculation. Starting from a sequential version as a basis, the parallelisation concern can be successfully encapsulated in aspects. This concern is no longer tangled within the computation code, and the application is more flexible, since it can be compiled so as to implement parallelism using whichever of the two techniques is desired.

The aspect for multi-threading, in Listing 3.4, uses an `around` piece of advice for intercepting calls to `cipher_idea(*, *, *, int, int)` made from within the original `cipher_idea(*, *, *)`. These original calls are not executed, but the advice proceeds with the execution of the refactored `cipher_idea` method via inner instances of `Runnable`, each run in a Java Thread.

The aspect for the MPI-based parallelisation, in Listing 3.5, intercepts accesses to portions of these arrays (divided across several processes using the Message

Listing 3.2: Implementation of `cipher_idea` in the multi-threaded version.

```
    ...

    /* The cipher_idea method does not actually exist in the
       multi-threaded version, but its equivalent is implemented thus:
     */
    private void cipher_idea(byte[] text1, byte[] text2, int[] key) {
        /* Declaration of local variables and initialisations */
        ...

        for (int i = 1; i < /* Total number of threads */; i++) {
            thobjects[i] = new IDEARunner(i, text1, text2, key);
            th[i] = new Thread(thobjects[i]);
            th[i].start();
        }
        thobjects[0] = new IDEARunner(0, text1, text2, key);
        thobjects[0].run();
    }
    ...

/**
  The multi-threaded version uses class IDEARunner, implementing
  Runnable, to split the loop across several Threads. The content of
  this class is the same as the content of the cipher_idea method in
  the sequential version, except that the bounds of the loops are
  defined according to the thread id given to the constructor.
 */
class IDEARunner implements Runnable {
    int id, key[];
    byte text1[], text2[];

    public IDEARunner(int id, byte[] text1, byte[] text2, int[] key) {
        this.id = id;
        this.text1 = text1;
        this.text2 = text2;
        this.key = key;
    }

    private void run () {
        /* Declaration of local variables and initialisations */
        ...

        ilow = id * slice;
        iupper = (id + 1) * slice;
        if (iupper > text1.length)
            iupper = text1.length;

        for (int i = ilow; i < iupper; i += 8) {
            /* Body of the loop */
            ...
        }
    }
}
```

Listing 3.3: Re-factoring of `cipher_idea` for aspects.

```
...

/**
   This is a new method whose behaviour matches the behaviour of
   the original version. It is merely a call to the following method
   with default values for the bounds.
 */
private void cipher_idea(byte[] text1, byte[] text2, int[] key) {
    cipher_idea (text1, text2, key, 0, text1.length) ;
}


/**
   This is an overridden method, with the original content (apart
   from the loop bounds) and two extra parameters (the loop
   bounds)
 */
private void cipher_idea(byte[] text1, byte[] text2, int[] key,
                         int ilow, int iupper) {
   /* Declaration of local variables and initialisations */
   ...

   /* Instead of ''for (int i = 0; i < text1.length; i += 8)'' */
   for (int i = ilow; i < iupper; i += 8) {
       /* Body of the loop from the original version*/
       ...
   }
}
...
```

Listing 3.4: Example aspect for parallelising *Crypt* using multiple Java Threads.

```
public privileged aspect MultiThreadsCrypt {
   private final int NUM_THREADS;

   public MultiThreadsCrypt () {
      NUM_THREADS = Integer.parseInt(System.getProperty("threads","1"));
   }

   void around(int ilow, int iupper) :
               call(void IDEATest.cipher_idea(..))
           && args(*, *, *, ilow, iupper)
           && withincode(void IDEATest.cipher_idea(*, *, *)) {

      Runnable[] runnables = new Runnable[NUM_THREADS];
      Thread[] threads = new Thread[NUM_THREADS];

      int tslice = (iupper - ilow) / 8;
      int ttslice = (tslice + NUM_THREADS - 1) / NUM_THREADS;
      int slice = ttslice * 8;

      for (int k = 0; k < NUM_THREADS; k++) {
         final int localilow = k * slice;
         int iuppertemp = (k + 1) * slice;
         if (iuppertemp > iupper)
            iuppertemp = iupper;
         final int localiupper = iuppertemp ;

            runnables[k] = new Runnable() {
               public void run() {
                  proceed(localilow, localiupper) ;
               }
            } ;
      }

      for (int k = 1; k < NUM_THREADS; k++) {
         threads[k] = new Thread(runnables[k]);
         threads[k].start();
      }

      runnables[0].run();

      for (int k = 1; k < NUM_THREADS; k++) {
         try {
            threads[k].join();
         } catch (InterruptedException e) {
         }
      }
   }
}
```

Listing 3.5: Example aspect for parallelising *Crypt* using MPI.

```
public privileged aspect MPICrypt {
   private int NUM_PROC;
   private int rank;


   pointcut mainMethodExecution():
                  execution(void JGFCryptBenchSize*.main(..));

   void around(String[] arg): mainMethodExecution() && args(arg) {
      try {
         MPI.Init(arg);
         rank = MPI.COMM_WORLD.Rank();
         nprocess = MPI.COMM_WORLD.Size();

         proceed(arg);

         MPI.Finalize();
      } catch (MPIException e) {
         e.printStackTrace();
      }
   }

   void around(): (call(* JGFInstrumentor.*(..)) ||
                  execution(* JGFCryptBench.JGFvalidate()))
               && cflow(mainMethodExecution()) {
      if (rank == 0) {
         proceed();
      }
   }

   int p_array_rows;
   int ref_p_array_rows;
   int rem_p_array_rows;

   byte[] p_plain1 = null;
   byte[] p_crypt1 = null;
   byte[] p_plain2 = null;

   before(IDEATest idea): call(* IDEATest+.buildTestData())
         && withincode(* JGFCryptBench.JGFinitialise())
         && target(idea) {
      p_array_rows = (((idea.array_rows/8)+NUM_PROC-1)/NUM_PROC)*8;
      ref_p_array_rows = p_array_rows;
      rem_p_array_rows =
         p_array_rows - ((p_array_rows*NUM_PROC)-idea.array_rows);
      if (rank == (NUM_PROC - 1)) {
         if ((p_array_rows * (rank + 1)) > idea.array_rows) {
            p_array_rows = rem_p_array_rows;
         }
      }
   }
```

Listing 3.5 continued: Example aspect for parallelising *Crypt* using MPI.

```
before(): execution(void IDEATest+.buildTestData()) {
   p_plain1 = new byte[p_array_rows];
   p_crypt1 = new byte[p_array_rows];
   p_plain2 = new byte[p_array_rows];
}

before(): execution(void IDEATest+.freeTestData(..)) {
   p_plain1 = null ;
   p_crypt1 = null ;
   p_plain2 = null ;
}


pointcut doMethodExecution() : execution(void IDEATest +.Do());

byte[] around(): get(* IDEATest+.plain1)
      && withincode(void IDEATest+.Do()) {
   return p_plain1;
}

byte[] around(): get(* IDEATest+.plain2)
      && withincode(void IDEATest+.Do()) {
   return p_plain2;
}

byte[] around(): get(* IDEATest+.crypt1)
      && withincode(void IDEATest+.Do()) {
   return p_crypt1;
}

void around(IDEATest idea): doMethodExecution() && target(idea) {
   try {
      int m_length;

      MPI.COMM_WORLD.Barrier();

      if (rank == 0) {
         for (int i = 0; i < p_array_rows; i++) {
            p_plain1[i] = idea.plain1[i];
         }
         for (int k = 1; k < NUM_PROC; k++) {
            if (k == NUM_PROC - 1) {
               m_length = rem_p_array_rows;
            } else {
               m_length = p_array_rows;
            }
            MPI.COMM_WORLD.Ssend(
               idea.plain1,
               (p_array_rows * k),
               m_length,
               MPI.BYTE,
               k,
               k);
         }
```

Listing 3.5 continued: Example aspect for parallelising *Crypt* using MPI.

```
      } else {
        MPI.COMM_WORLD.Recv(p_plain1, 0, p_array_rows,
            MPI.BYTE, 0, rank);
      }

      proceed(idea);

      MPI.COMM_WORLD.Barrier();

      if (rank == 0) {
        for (int k = 0; k < p_array_rows; k++) {
          idea.plain2[k] = p_plain2[k];
        }

        for (int k = 1; k < NUM_PROC; k++) {
          MPI.COMM_WORLD.Recv(
            idea.plain2,
            (p_array_rows * k),
            p_array_rows,
            MPI.BYTE,
            k,
            k);
        }
      } else {
        MPI.COMM_WORLD.Ssend(p_plain2, 0, p_array_rows,
            MPI.BYTE, 0, rank);
      }

      MPI.COMM_WORLD.Barrier();
    } catch (MPIException e) {
      e.printStackTrace() ;
    }
  }
}
```

Passing Interface). This could have been effected in the original code, but it works the same way in the refactored code, and thus allows the latter to be used as a multi-purpose expression of the main computational concern, which can be targetted at shared memory multiprocessors or distributed memory multicomputers, as desired.

3.1.2 Major refactoring

The next attempt at using aspects for encapsulating parallelisation is the legacy code *LUFact* in the JGF benchmark suite. This application is based on a Java implementation of Linpack, which is originally "*a collection of Fortran subroutines that analyze and solve linear equations and linear least-squares problem*" [DBMS]. Substantial refactoring is necessary simply to put this code in an object-oriented form. Thereafter, a fundamental incompatibility is found between the mechanisms for defining join points in AspectJ and the needs of the application.

As shown in its header comments, the Java version of *LUFact* is an adaptation of a C version, which had itself been translated from Fortran. The result still reflects the original coding style of the Fortran subroutines. Matrices and vectors are represented by arrays of `doubles`; the dimensions of the matrices are not part of their data representation, but an extra parameter given to the functions (it would have been better to use the `length` of the Java array, or a field if the representation was a class). Thus, although this example is valid for testing the performance of JVMs against this kind of code, it does not provide a genuinely object-oriented application; turning a function into a method does not make an object-oriented design. As a result, the potential join points that would have been interesting for parallelisation are `for`-loops and array accesses. However, these points are not join points in AspectJ, which aims to use aspects only in truly object-oriented designs, that is to say, when it is possible to work at object or interface level.

The possibility of having a join point for loops is investigated in Chapter 4. However, the problem of identifying and selecting loops would remain: the sequences of instructions within Linpack procedures are too low-level to help identify the purpose of each instruction that uses the array representation. Unlike the *Crypt* example, the multi-threaded *LUFact* implementation requires more than identifying a tile in the iteration space; it also involves synchronising between selected instructions that make accesses to the arrays. Because it is impossible to distinguish between the various reasons for accessing arrays, it is impossible to characterise which accesses are to be party to which synchronisations. Without a

meaningful abstraction, it is practically impossible accurately to recognise the sub-operations and therefore to plan their parallelisation using aspects. Thus, without completely re-writing *LUFact* with an adequate object-oriented abstraction, there is nothing more that can be done using AspectJ.

3.1.3   Moderate refactoring

A third application, which appears at first sight to have an appropriate object-oriented structure, is *RayTracer*. With a moderate amount of refactoring, this might be expected to be amenable to parallelisation via aspects. However, it reveals other problems.

RayTracer produces 2D bitmap images from 3D display models. The important part for parallelisation is located in its method `void render (Interval interval)`; this is where the methods for computing pixel values are called, and where the results are stored. The `interval` parameter is an object which defines the area of the picture that is to be computed. Unfortunately, even in the multi-threaded version, the data structure in which it was chosen to store the results is an array declared as a local variable. Since the `render` method is of type `void`, and its results are stored in a local variable, the multi-threaded version never gathers the results together so as to form the final picture. As a result, the synchronisation that should have happened at the end of the computation is avoided. Both functional behaviour and timing results are therefore questionable.

This problem can be solved by creating a new class, `Image`, which contains an array field to store a picture, and by modifying the `render` method to become `void render (Interval interval, Image image)`, and have it save the results in the `image` object. It would then be possible to create an aspect that would replace the original calls to method `render`, and execute portions of the original `interval` in multiple threads. The original multi-threaded version could have used a data structure such as `Interval` to define the iteration space (and therefore the partitioning and the scheduling of the tasks in the parallel versions) in both sequential and parallel implementations. Conveniently, a tile of the iteration space would also be a subset of pixels of the image rendered, since the computation is embarrassingly parallel with respect to each pixel. Instead, the implementation of the scheduling is scattered in the multi-threaded version: `interval` contains a thread-id field which is used by `render` along with the total number of threads obtained from a static field belonging to another class.

This application would need a better object-oriented design to both clarify the sequential design and to improve the validity of its assessment of object-oriented Java. It would then be possible to write aspects for implementing the two parallel versions (using threads or MPI) in much the same way as shown in Section 3.1.1.

3.1.4   Aspects for the JGF benchmarks: summary

The underlying design of an application is crucial for allowing appropriate aspects to be written. AspectJ is a general-purpose aspect-oriented extension to Java, and it expects to work mostly on object or class interfaces. Unfortunately, not all Java code has well-designed object and class interfaces, and many of the applications in the Java Grande Forum benchmark suite are deficient in this respect.

Most C programs can be compiled by C++ compilers, but putting C statements into a big C++ class does not turn a procedural implementation into an object-oriented one. The same phenomenon happens in Java. Java was originally designed to be an object-oriented language, but it is possible to program large methods that are direct translations of C functions. Such programs are best described as "procedural" Java programs. Measuring the performance of JVMs using this kind of application is biased towards a programming style inherited from procedural languages. Benchmarking programs using abstractions that are more object-oriented would test the suitability not only of the JVM but also of Java as an object-oriented language for numerical computing.

For it to be feasible to parallelise using AspectJ, large tasks must make information about their sub-division visible in their object interfaces. This requirement entails only a minor modification to *Crypt*, making the iteration space parameter accessible from the interface of one large method. A similar tiling of the iteration space might have been achieved in *RayTracer*, after a few modifications. However, "procedural" Java programs, like *LUFact*, that freely use loop-constructs and array accesses, and that do not encapsulate small tasks into meaningful methods, make it very difficult to encapsulate parallelisation using AspectJ aspects. The difficulties encountered also suggest that the idea of writing aspects for implementing performance should be envisaged from the early stages of the application design.

Where truly object-oriented designs have been developed for scientific code, evidently there will be a need to traverse multi-dimensional structures using nested `for`-loop constructs. Given that AspectJ will have difficulty intercepting the iterations of such constructs, this warrants further investigation. Note that the

successful refactoring of *Crypt* involved *tiling* of a 1-dimensional iteration space. This is a special case of a classical technique for managing data parallelism over multi-dimensional iteration spaces, and it suggests a generalisation that might readily be put into truly object-oriented form. The next section thus investigates alternative ways of representing 2-dimensional `for`-loops whose iterations can be intercepted using AspectJ. Such alternative forms could potentially be used to solve the problem found in *RayTracer*. Chapter 4 goes a step further and provides an extension to AspectJ for a loop join point.

3.2   An object-oriented model for loops

This section proposes new representations for nested `for`-loops, which have a structure appropriate to aspect-oriented parallelisation using AspectJ. The models are designed in such a way that the points of the loops where the parallelisation can happen are also valid AspectJ join points. The examples focus on rectangular double-nested loops, with fixed bounds, which would usually be written in the form shown in Listing 3.6. It is assumed that the different instances of the body of the loop can be executed in no particular order; thus, executing the loop body with certain values of i and j does not impact upon the execution of the same loop body with other values of i and j (i.e. the loop nest is *embarrassingly parallel*).

Listing 3.6: Rectangular double loop nest in Java.

```
for (int i = minI; i <= maxI; i++) {
    for (int j = minJ; j <=maxJ; j++) {
        /* Loop body. Functions of i and j. */
    }
}
```

Three object models for `for`-loops are introduced: RectangleLoopA (Section 3.2.1), RectangleLoopB (Section 3.2.2) and RectangleLoopC (Section 3.2.3). The evolution from form A to form C attempts to improve performance, based on expectations about the JVM and any just-in-time optimisations. Actual performance results are presented in Chapter 5, in particular in Section 5.6.

3.2.1   Model RectangleLoopA

The first model is the `RectangleLoopA` design pattern presented in Figure 3.1. This consists of a *delegation* relationship between the following:

- the `Runnable2DLoopBody` interface for representing a double-nested loop body;

- the `RectangleLoopA` class that is in charge of executing the iterations.
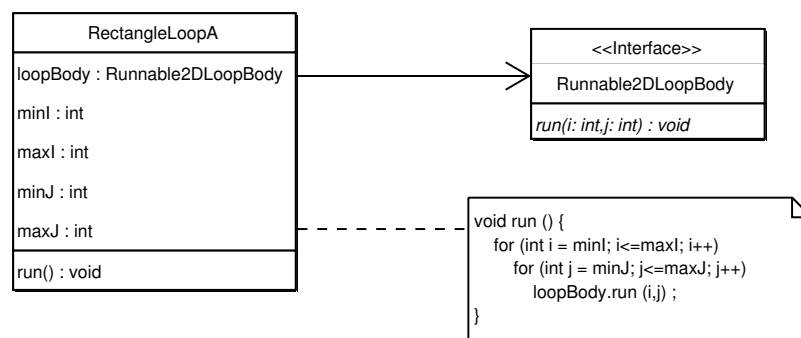
Figure 3.1: UML class diagram for model `RectangleLoopA`.

In this model, the body of the loop appears in the `void run(int i, int j)` method of a class that implements the `Runnable2DLoopBody` interface. The `i` and `j` parameters to the method are the loop indices of the associated double-nested `for`-loop. An instance of `RectangleLoopA` holds an instance of `Runnable2DLoopBody` (its `loopBody` attribute) and the bounds (`minI`, `maxI`, `minJ` and `maxJ`) of the rectangle in the iteration space for which it is responsible. The `run()` method of `RectangleLoopA` is a double-nested loop that executes the `run(int i, int j)` in its `loopBody` attribute over the part of the iteration space defined by the bounds. As an example of this scheme, the loop described in Listing 3.6 would be refactored as shown in Listing 3.7.

Creation of an instance of `RectangleLoopA`, or a call to its `run()` method, together with the dynamic context (in this case, the arguments to the constructor), can be intercepted by an aspect in AspectJ. As a result, parallelism can be implemented transparently in the nested loop structure.

Listing 3.7: Implementation of a double `for`-loop nest using `RectangleLoopA`.

```
class ConcreteLoopBody implements Runnable2DLoopBody {
    ...
    public void run (int i, int j) {
        /* Loop body. Functions of i and j. */
    }
}


    ...
    /* Where the regular for-loop would be */
    Runnable2DLoopBody loopBody = new ConcreteLoopBody (...) ;
    RectangleLoopA loop =
            new RectangleLoopA (loopBody, minI, maxI, minJ, maxJ) ;
    loop.run () ;
    ...
```

The `RectangleLoopA` implementation[1] makes one call per iteration to a method belonging to an external object (`loopBody.run(i,j)`). This is assumed to damage performance, since it is doubtful that the JVM can optimise it.

3.2.2 Model RectangleLoopB

In order to address the above performance problem, the second model, `RectangleLoopB`, does not use an external class, but defines its own abstract method `loopBody(int i, int j)`, which must be overridden in a sub-class so as to contain the body of the loop. The UML class diagram is shown in Figure 3.2. The *delegation* relationship of the previous model is replaced by an *inheritance* relationship. According to this model, the implementation of a double-nested `for`-loop should be of the form shown in Listing 3.8.
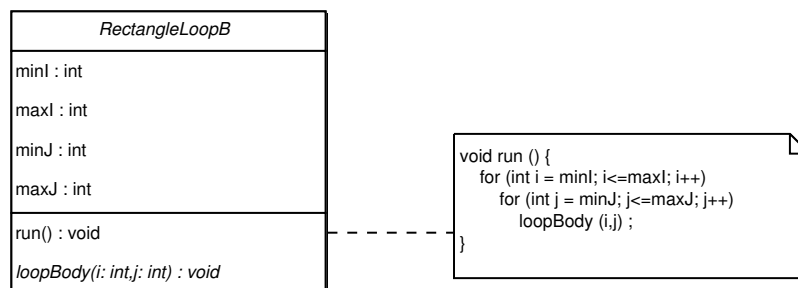
Figure 3.2: UML class diagram for model `RectangleLoopB`.

---

[1]See Listing C.1, in Appendix C.1.

Listing 3.8: Implementation of a double `for`-loop nest using `RectangleLoopB`.

```
class ConcreteLoopB extends RectangleLoopB {
    ...
    public void loopBody (int i, int j) {
        /* Loop body. Functions of i and j. */
    }
}


    ...
    /* Where the regular for-loop would be */
    RectangleLoopB loop = new ConcreteLoopB (minI, maxI, minJ, maxJ) ;
    loop.run () ;
    ...
```

The `RectangleLoopB` implementation[2] makes one call to one of its own methods (`loopBody(i,j)`) per iteration. Again, the JVM optimisation is not predictable, but it is assumed that there is still some associated performance penalty.

3.2.3   Model RectangleLoopC

It is now assumed that both the above models prevent the JVM from performing certain optimisations on loops, such as loop unrolling. Thus, in the third model, `RectangleLoopC`, the regular structure of the inner (`j`) loop is maintained.[3] The UML class diagram is shown in Figure 3.3. According to this model, the implementation of a double-nested `for`-loop should be of the form shown in Listing 3.9. In this model, the `loopDoJRange(int i, int minJ, int maxJ)` method has to iterate through one "line" of the rectangle, from `minJ` to `maxJ`, using index `i`.
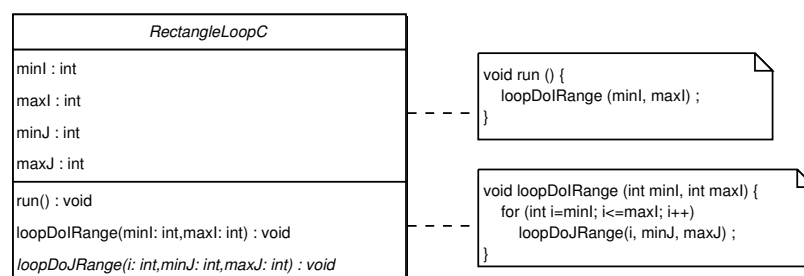
Figure 3.3: UML class diagram for model `RectangleLoopC`.

---

[2]See Listing C.3.

[3]See Listing C.4.

Listing 3.9: Implementation of a double `for`-loop nest using `RectangleLoopC`.

```
class ConcreteLoopC extends RectangleLoopC {
    ...
    public void loopDoJRange (int i, int minJ, int maxJ) {
       for (int j=minJ; j <= maxJ; j++) {
          /* Loop body. Functions of i and j. */
       }
    }
}


    ...
    /* Where the regular for-loop would be */
    RectangleLoopC loop = new ConcreteLoopC (minI, maxI, minJ, maxJ) ;
    loop.run () ;
    ...
```

3.2.4   Object-oriented loops: summary

The above three models have been designed so that the iteration space and the loop body of an embarrassingly parallelisable, double-nested `for`-loop can be encapsulated into an object. Creation and manipulation of such objects can then be recognised in AspectJ aspects. Thus, AspectJ can be used to define a tile of the iteration space originally defined in a sequential implementation, and parallelise the loop accordingly.

Although model `RectangleLoopA` enforces a better separation of concerns by using two classes to describe the boundaries and loop body, performance results featured in Chapter 5 show that the required method calls to an external object are too expensive. Models `RectangleLoopB` and `RectangleLoopC` both encapsulate loop characteristics into a single class. However, although `RectangleLoopC` is expected to give better timing results, the programmer has to write the inner loop explicitly each time. Thus, model `RectangleLoopB` is arguably the best reusable object-oriented pattern. As shown in Section 5.6.1.1, models `RectangleLoopB` and `RectangleLoopC` produce similar performance results on the IBM JVM, but model `RectangleLoopC` performs substantially better than `RectangleLoopB` on the Sun JVM. (In both cases, the IBM JVM is faster than the Sun JVM).

3.3   Summary

This chapter has proposed schemes for using AspectJ for encapsulating parallelism. Section 3.1 has shown that this objective cannot always be achieved, in particular if the underlying numerical model is not implemented in a object-oriented way. Section 3.2 has proposed design patterns for describing loops in an object-oriented manner, that can be handled by AspectJ. In both cases, the key idea is to make the iteration space (and possibly the data) correspond to AspectJ join points, that is, to make this information visible and modifiable in the interface of the components, by refactoring (parts of) the application. Chapter 5 presents performance evaluations of such refactorings and subsequent advising of test-case applications.

The next chapter, Chapter 4, provides AspectJ with a model of join point capable of addressing loops directly in the code, so that refactorings can be avoided.

Chapter 4

A join point for loops in AspectJ

The work presented in this chapter has led to the publication of two articles presented, respectively, at the *Foundations of Aspect-Oriented Languages workshop*[1] *(FOAL'2005)* [HG05] and at the *5th International Conference on Aspect-Oriented Software Development (AOSD'2006)* [HG06].

When parallelising code in order to improve performance, loops are the natural places to make changes. There are sometimes several alternative ways of parallelising the same loop, depending on various parameters, such as the nature of the data being processed, or the architecture on which the application is going to be executed. In certain cases, it is possible to use aspects for parallelising loops, in particular for choosing a method of parallelisation, as shown in Chapter 3. However, since there is currently no join point for loops in AspectJ [KHH$^+$01b], the method proposed in Chapter 3 resorts to refactoring the base-code. In order to eliminate this inconvenience, this chapter proposes a loop join point model for AspectJ which allows direct parallelisation of loops, without refactoring of the base-code.

Section 4.1 presents the loop join point model and the kind of loops it aims to recognise. Although it is based on Java and AspectJ, the model can potentially be applied to other languages. Section 4.2 explains why the approach is based on the bytecode and not on the source code. Section 4.3 presents the model from the point of view of an aspect compiler, and explains the join point *shadow*. Section 4.4 enhances the join point model with a relation to the data handled by the loops. Section 4.5 explains the specific requirements for loop selection, and describes the associated difficulties, compared with other kinds of join points. Section 4.6

---

[1]FOAL'2005 was held in conjunction with the *4th International Conference on Aspect-Oriented Software Development (AOSD'2005)*.

Listing 4.1: Example of Java `for`-loops iterating over a `Collection`.

```
/* Since Java 5 */
Collection<ExampleClass> c ;
for (ExampleClass obj: c) {
    /* Do something with obj */
}


/* Before Java 5 */
Collection c ;
for (Iterator it = c.iterator() ; it.hasNext(); ) {
    ExampleClass obj = (ExampleClass)it.next() ;
    /* Do something with obj */
}
```

describes some of the resulting problems related to base-code which throws exceptions. Section 4.7 introduces LoopsAJ, a prototype implementation, based on abc,[2] of a weaver capable of handling the loop join point model. Section 4.8 embellishes the join point model with reflective capabilities, so that further information about the loop can be obtained by the aspect. Section 4.9 shows how to write aspects for parallelisation using the loop join point. Section 4.10 briefly introduces ideas for other potential fine-grained join points, namely a "loop-body" join point and an "if-then-else" join point.

4.1   The loop join point model

This section presents the objective of the loop join point model. This objective consists of defining: (a) the behaviour the model aims to recognise, and (b) its dynamic (run-time) characteristics, that is, providing the join point with an execution context. The resulting model could be applied to various aspect-oriented systems, but the presentation focusses on AspectJ.

Java 5 offers an *enhanced* `for` *statement*, similar to "for-each" constructs in other languages [BB04, GJSB05], as shown in Listings 4.1 and 4.2. Although this is a general purpose construct, the behaviour it defines is ideal for parallelisation because this construct encapsulates both the iterative structure and the associated data that are to be processed. This "for-each" abstraction is used below as a basis for the loop join point model.

---

[2]http://www.aspectbench.org/

Listing 4.2: Example of Java `for`-loops iterating over an array.

```
/* Since Java 5 */
ExampleClass[] a ;
for (ExampleClass obj: a) {
    /* Do something with obj */
}


/* Before Java 5 */
ExampleClass[] a ;
for (int i = 0 ; i < a.length ; i++) {
    ExampleClass obj = a[i] ;
    /* Do something with obj */
}
```

For iterating over the elements of a `Collection` or of an array, the loop constructs prior to Java 5 rely on the abstraction provided by an `Iterator` or by an array index, respectively. The enhanced `for` statement in Java 5 puts emphasis on the data processed by the loop (the `Collection` or the array) rather than on the means of access to the data (the `Iterator` or the array index). The data to be processed is directly and explicitly included in the way the `for`-loop is written in the source code. This new abstraction is solely a source-code enhancement; the bytecode still contains `Iterators` (for `Collections`[3]) or local `int` variables (for array indices).

It can also be useful for parallelisation to recognise a "weaker" form of loop, namely `for`-loops that do not explicitly iterate over a `Collection` or an array, but still use an `Iterator` or an `int` index.

The loop join point model should make it possible to extract information regarding the execution context at the join point. This information should contain the iteration space (that is to say, the instance of `Iterator` or the range of integers). For the "stronger" form of loops, it should also include the data being processed (that is to say, the specific instance of `Collection` or the array). The execution context is particularly useful for selecting loops, as shown in Section 4.5.

---

[3] Java 5 also provides a new interface called `java.lang.Iterable<T>` [GJSB05, Section 14.14.2], which contains a single method signature: `Iterator<T> iterator()`. `java.util.Collection` and other methods in the Java Collection Framework have been retrofitted to implement this interface (see `http://www.jcp.org/aboutJava/communityprocess/jsr/tiger/enhanced-for.html`). The enhanced `for` construct can be used with `Iterables` that would not be `Collections`. In the context of Java 5, `Iterable` could be used wherever `Collection` appears in this chapter.

"Strong" form	"Weak" form

```// Collection c = ...```
```Iterator it = c.iterator() ;```
```while(it.hasNext()) {```
```   Ex obj = (Ex)it.next() ;```
```   /* Do something with obj */```
```}```
 	
Context: Collection and Iterator.
	```// Iterator it = ...```
	```while(it.hasNext()) {```
	``` Object obj = it.next() ;```
	``` /* Do something with obj */```
	```}```
	
	Context: Iterator.

```// Ex[] a = ...```
```for (int i=0 ; i<a.length ; i++){```
```   Ex obj = a[i] ;```
```   /* Do something with obj */```
```}```
 	
Context: array, min, max and stride.
	```for (int i=MIN ; i<MAX ;i+=STRIDE){```
	``` /* Do something */```
	```}```
	
	Context: min, max and stride.

Figure 4.1: Summary of the patterns to be recognised by the loop join point model.

A summary of the forms of loop that the join point ought to recognise is presented in Figure 4.1. Further details on recognising the loops and on exposing the context are presented in Sections 4.3 and 4.4, respectively.

4.2   From source or from bytecode

Although this may seem to be an implementation decision, choosing whether the join point is recognised at source code level or at bytecode level may completely change the model.[4] The way loops are programmed in Java is not necessarily directly reflected in the generated bytecode. For example, instinctively, most Java programmers would consider the body of a `for`-loop to be the lines of code within the curly brackets following the `for(;;)` statement. However, a loop with the same effect can also be written in different ways, for example as a `while`-loop, or with some of the `for` statements displaced, as shown in Listing 4.3.

In addition, the main conditional expression of a loop may encompass several instructions, in particular when it involves a call to a method or a complex expression, as shown in Listing 4.4. Although the condition may not seem to be part of the loop body, it could always be refactored so as to be so (for example through a temporary `boolean` variable). Moreover, the compiled code does not necessarily reflect the way a complex expression has been written in the source code.

---

[4]See Appendix B for a description of one possible scheme based on recognition of join point only at the source-code level.

Listing 4.3: Simple examples of equivalent loops.

```
for (int i = 0 ; i < MAX ; i++) {
    /* A */
}

int j = 0 ;
int STRIDE = 1 ;
for ( ; j < MAX ; j += STRIDE ) {
    /* A */
}

int k = 0 ;
while (k < MAX) {
    /* A */
    k++ ;
}
```

Listing 4.4: Loop with more complex conditions.

```
int i = 0 ;
while (condition(i) || (i<10)) {
    /* A */
    i++ ;
}

int j = 0 ;
boolean ok = condition(j) || (j<10) ;
while (ok) {
    /* A */
    j++ ;
    ok = condition(j) || (j<10) ;
}
```

Since the main concern is to recognise the behaviour of the code, rather than the way it was written, the choice has been made to base the representation of loops at the bytecode level rather than at the source code level. As a result, the representation is more robust to variations in programming style. However, this choice introduces limitations regarding (a) the potential specific handling of an abrupt exit[5] (see Section 4.3.3), and (b) the nature of the control-flow graphs. Indeed, as explained in more detail in Section 4.6, the loop join point model expects a *reducible* (or *well-structured* [ASU85, Muc97]) graph. Java source-code produces bytecode with reducible control-flow graphs, but this is not necessarily the case for bytecode produced by other means.[6]

A bytecode approach also follows the evolution of AspectJ, which, since version 1.1, does the shadow-matching and weaving only at bytecode level [HH04].

4.3 Shadow matching: recognising the loops

This section describes the way the loops are recognised: specifically, how the *shadows* of the loops are identified. The *shadow* of a join point is defined as follows: "*[A] join point is a point in the dynamic call graph of a running program [...]. Every [such] dynamic join point has a corresponding static shadow in the source code or bytecode of the program. The AspectJ compiler inserts code at these static shadows in order to modify the dynamic behavior of the program*" [HH04].

Although the aim is to recognise loops of the forms presented in Section 4.1, this section investigates various forms of loop with respect to their ability to constitute a basis for a loop join point model. The main requirements for a join point shadow are:

1. the ability to weave *before*-advice, *after*-advice and *around*-advice;

2. the ability to extract the context of execution at the join point.

---

[5] Abrupt exits are due in particular to `break` statements [GJSB05, Section 14.15], and are different from exits due to exceptions. The remainder of this chapter, apart from Section 4.6, does not take exceptions into account.

[6] Certain representations of control-flow graphs can be non-reducible if exceptions are taken into account.
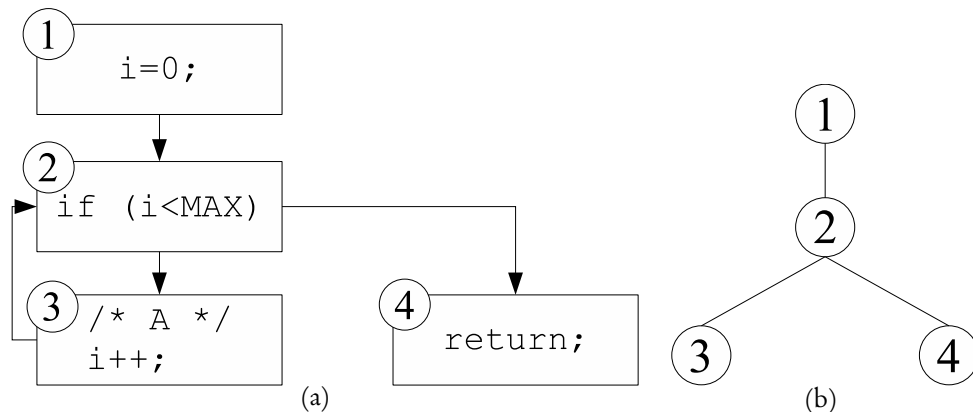
Figure 4.2: Control-flow graph (a) and dominator tree (b) for a simple `for`-loop.

4.3.1   Dominators, back edges and natural loops

The initial approach for finding loops analyses the control-flow graph (constructed from the bytecode) and follows the method described in [ASU85, Muc97]. This method is based on finding *dominators* and *back edges*, defined as follows: *"Node* d *of a flow graph* dominates *node* n *[...] if every path from the initial node of the flow graph to* n *goes through* d. *[... The] edges in the flow graph whose heads dominate their tails [are called]* back edges. *(If* a → b *is an edge,* b *is the* head, a *is the* tail. *[... Also,* a *is a* predecessor *of* b, *and* b *is a* successor *of* a *...]) Given a* back edge n → d, *we define the* natural loop *of the edge to be* d *plus the set of nodes that can reach* n *without going through* d. *Node* d *is called the* header *of the loop"* [ASU85]. By their very definition, dominators form a tree structure, called the *dominator tree*.

Figures 4.2(a) and 4.2(b) represent, respectively, the (block-level[7]) control-flow graph and the associated dominator tree for the simple `for`-loop shown in Listing 4.3. In this example, the only back edge is 3 → 2, and its natural loop comprises blocks (nodes) 2 (which is the header) and 3.

Natural loops can be confusing because there could be several loops with the same header. As shown in Figure 4.3, what appears to be a single loop actually corresponds to two natural loops sharing the same header. In such a case, defining the points immediately *before* or *after* a natural loop would be ambiguous. Therefore, instead of using natural loops for the join point model, the union of all the natural loops sharing the same header is considered as a single *combined loop*. In order to avoid ambiguous cases, implementations should consider a node containing only an unconditional `goto` as being the same node as its successor node. In the remainder

---

[7]i.e., the nodes of the control-flow graph are *basic blocks* [ASU85] of code statements.

```
int i = 0 ;
while (i<MAX) {
    if (cond(i++)) {
        /* A */
    } else {
        /* B */
    }
}
```
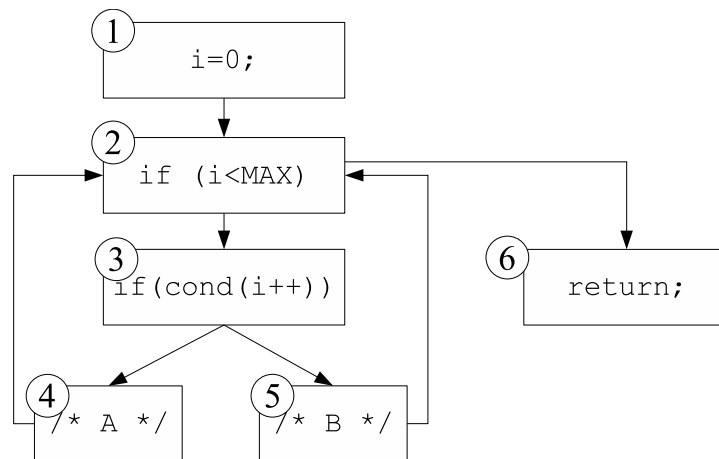

Figure 4.3: A combined loop consisting of two natural loops with the same header.

of this chapter, the term *"loop"* will be used to mean a "combined loop", unless otherwise stated.

Following this style, an *inner loop* is a loop whose blocks are all contained within another loop, but do not share the latter's header. This also happens to match the natural definition of inner loops at the source-code level.

In the following sections, three categories of loops are presented, together with characteristics pertinent to their possible use as join points. The categories introduce increasing degrees of constraint which affect their ability to weave the three forms of advice: *before*, *after* and *around*. The loops that are to be recognised by the loop join point model, as described in Section 4.1, fall into the third category.

4.3.2 Loops in the general case

A loop always has a unique entry point, namely its header. *Before-advice* can therefore always be woven in a *pre-header*, that is, a node (block) inserted before the header to which the jumps from outside the considered loop are redirected, but the jumps from inside it are not, as shown in Figure 4.4.
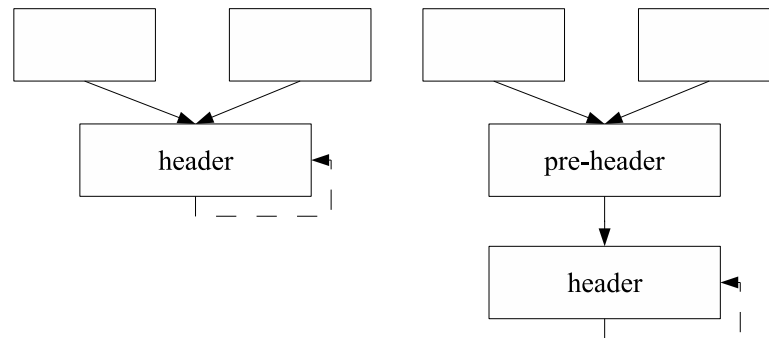
Figure 4.4: Insertion of a pre-header.

Without further constraint, it cannot be guaranteed that there is a unique point in the control flow that is executed immediately after execution of a loop. In order to introduce appropriate constraints, the following definitions are added. A node in a loop is an *exit node* if it can branch outside that loop. A node outside a loop which has predecessors inside that loop is termed a *successor node* of the loop.

Typically, a non-nested loop which contains a `break` statement has two exit nodes and one successor node, while a double loop nest with a `break` statement in the inner loop that branches outside the outer loop has two exit nodes and two successor nodes. For example, Figure 4.5 shows the source code and the corresponding (block-level) control-flow graph for a double loop nest in which:

- the inner loop consists of blocks 4, 5 and 6; its exit nodes are blocks 4 and 5; its successor nodes are blocks 7 and 8; and

- the outer loop consists of blocks 2, 3, 4, 5, 6 and 7; its exit nodes are blocks 2 and 5; its (sole) successor node is block 8.

In this case, "after" the inner loop ($\{4, 5, 6\}$) is both on edge $4 \rightarrow 7$ and on edge $5 \rightarrow 8$.

In such cases, where there are several successor nodes, there are multiple points corresponding to the transition from blocks within the loop to blocks outside the loop: weaving an *after* piece of advice would require replication of the woven code at all edges between exit nodes and their successor nodes. Although it is, in principle, possible to achieve this, some aspect-oriented tools do not allow this kind of weaving.

More fundamentally, weaving *around*-advice into this kind of loop would not be possible, even theoretically. Indeed, the execution of an *around* piece of advice

```
int i = 0 ;
outerloop:
while (i < maxI) {
    int j = 0 ;
    while (j < maxJ) {
        if (c(i,j))
            break outerloop ;
        j++ ;
    }
    i++ ;
}
/* A */
```
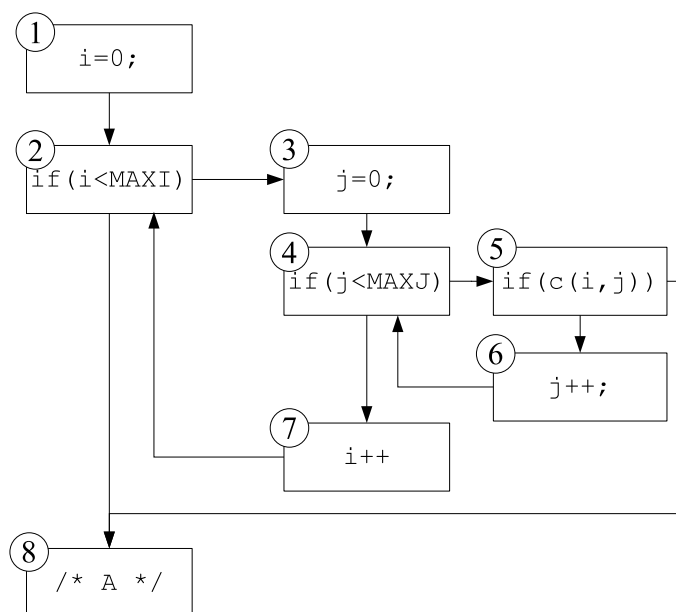

Figure 4.5: Two nested loops with a break statement jumping outside the outer-loop.

returns to the point where the original join point would have returned (i.e. just after where `proceed()` returns[8]). Although it is possible to get and change the value returned by the original join point with `proceed()`, there is no mechanism to allow several return points from `proceed()`. Using `proceed()` in an *around*-advice is not even compulsory: where would the control-flow be redirected in a case where `proceed()` is not used and there are several potential successors?

4.3.3   Loops with a unique successor node

The problem of having multiple successor nodes occurs when there are nested `break` or `continue` statements that branch outside the inner-most loop to which they belong. The default case (of a `break` statement with no label specified) corresponds to an exit node that branches outside the loop, but to the same successor node as the normal exit would go, as long as nothing is executed between the branching test and the `break` statement itself.

In this case, weaving an *after* piece of advice could be done either at the end of each exit node (possibly at multiple points, as described previously) or at the beginning of the (unique) successor node (which thus guarantees a single weaving point). Weaving an *after-advice* (at a single weaving point) therefore consists of inserting a *pre-successor*, i.e., a new node inserted prior to the successor node, to which the jumps from the exit nodes to the (unique) successor node are redirected.

A loop with a unique successor node can also be reduced to a single node in the control-flow graph. This then makes it possible to weave an *around-advice* at the join point for the loop.

Just as there are two different constructs for writing after-advice, depending on whether the execution returns normally or throws an exception[9], so might abrupt exits (due to `break` statements) be handled differently. However, there are cases where it is not possible to tell from the bytecode how such exits would differ from those due to the main condition of the loop evaluating to `false`. This is a limitation that might have been avoided if a source-code representation had been used, but it does make the model robust to changes in programming style, as illustrated by the code in Listing 4.5. The two loops in this listing might well produce the same bytecode and control-flow graph, in which case the use of `break` would not be

---

[8]See Sections 2.1.3 and A.4.

[9]"`after() returning(...):`" executes the advice after a normal execution, "`after() throwing(...):`" executes the advice if an exception has been thrown, and "`after():`" executes the advice in both cases.

Listing 4.5: Illustration of a possible special handling of `break` statements.

```
while (a && b) {
   /* Do something */
}

while (a) {
   if (!b)
      break ;
   /* Do something */
}
```

distinguishable from the use of the "`&&`" operator. It would thus be impossible to treat an exit from the loop due to the `break` statement any differently than an exit from the loop due to `b` evaluating to `false`.

4.3.4   Loops with a unique exit node

Having a unique exit node makes it possible to identify the variables required for the context exposure, as presented in Section 4.4. In particular, it is possible, from a single exit condition, to predict (as far as possible) that the loop iterates over an `Iterator` or over a specific range of integers (see Sections 4.1 and 4.4). The full potential of a loop join point can only be exploited if its model comprises information regarding the behaviour of the loop. Moreover, however clever the prediction and the context exposure may be, the programmer of an aspect dealing with loops might want to handle cases where there is no possibility of an abrupt exit (i.e., there is no `break` statement in the loop).

As shown in Listing 4.5, loops with complex conditions (in particular expressions comprising *and* and *or* operations) may create several exit points, and thus be ineligible for this category. From the point of view of the programmer, loops in this category are the loops without `break` statements and whose exit condition is equivalent to a single evaluation (this includes method calls or complex conditions that must be fully evaluated, via a boolean variable, for example). The loops in Figure 4.1 are a special case of this category, to the extent that specific types and ordering of the iteration space are expected.

	Before	After	Around	Context exposure
several successor nodes	√	multiple weaving points	×	×
several exit nodes, 1 successor node	√	√	√	×
1 exit node, 1 successor node	√	√	√	√

Table 4.1: Different loop types and their weaving capabilities.

4.3.5 Summary

Three categories of loops have been identified, with increasing degrees of constraint. Although only the last form (unique exit node) is suitable for the main objective given in Section 4.1, all three forms could be implemented by a different pointcut, each with different meaning and weaving capabilities. The more general form (several successor nodes possible) would only allow the weaving of *before*-advice, and possibly *after*-advice if the implementation of the weaver allows multiple weaving points. The intermediate form (unique successor node possible) and the restricted form (only one exit node and one successor node) would allow the weaving of *before*-, *after*- and *around*-advice. The latter also guarantees that there is a single condition for exit from the loop.

The above situation is summarised in Table 4.1, which also shows the context exposure capabilities, as described in Section 4.4.

4.4 Context exposure

Although loops do not have arguments in the same way that other join points (such as method calls) do, they often depend on contextual information to which programmers may want access. In particular, two forms of contextualised loop are frequently found, namely:

- loops iterating over an `Iterator`, and

- loops iterating regularly over a range of integers.

Knowing that a loop is of one of these forms allows one to determine, at compile-time, the execution behaviour of the loop in some detail. The `Iterator` or the range of integers can be considered as a first set of arguments of the loop (see

Section 4.4.1). In order to make these compile-time predictions meaningful, only loops with unique exit points and unique successors are considered for context exposure. This exempts from consideration loops which have any potential abrupt exits (e.g., using `break` statements); a potential use of `break` would make the finding of a range of integers or of an `Iterator` less useful, since the loop might exit before the anticipated end.

In addition, the data treated by the loops may be of interest, in particular for applications that need to know on what part of the data the loop is working (for example, parallelisation using MPI). This can form another set of contextual arguments for the loop (see Section 4.4.2).

4.4.1   Iteration space

4.4.1.1   Loop iterating over a range of integers

Loops iterating over a range of integers, following an arithmetic sequence, are one of the most common forms of loop. They consist of: initialising an integer local variable before the loop; incrementing this variable by some constant value (the stride) at the end of each iteration; and exiting the loop when the value reaches a given maximum value. This form of loop follows the pattern shown in Listing 4.3. This category of loop is similar to the "*trivial*" loops that are the target of parallelisation when using `javab` (a bytecode parallelisation tool) [BG97].

As explained in Chapter 3, exposure of the iteration space is essential to make it possible to write aspects for parallelisation. The initial value, the stride and the final value will be available in the execution context of the loop join point model, whenever possible. Since these values are parameters ruling the execution of the loop, they could be considered, in aspect-oriented models such as AspectJ, as "arguments" of the loop.

Knowing in advance what the range of integer values is going to be when the loop is executed is not always possible. In order to be exposed to the join point model, these values have to be determinable before the join point is encountered. The availability of these values will depend on the capabilities of the static analysis implemented in the shadow matcher. Determination of these values ought to be implemented in a conservative way, discarding the cases where it cannot be guaranteed that the values will not change during the execution of the loop.

4.4.1.2   Loop iterating over an Iterator

Another frequent form of loop (found in particular in Java programs) is that conducted by an `Iterator`. In a manner similar to that presented in Section 4.4.1.1, the instance of `Iterator` controlling the loop can be seen as an "argument" to be included in the join point context.

4.4.2   "Iterable" data

Both forms of loop presented in Section 4.4.1 may correspond to a "strong" form of loop, as shown in Section 4.1. The range of integers, or the `Iterator`, may correspond to an array or, respectively, to a `Collection`. According to the Java 5 terminology, they can be considered as *"iterables"*. Again, exposing this extra information may be useful, for example for certain parallelising schemes which require the programmer to transfer data between processing units explicitly (e.g. MPI), or for loop selection, as explained next.

4.5   Loop selection

This section analyses, and proposes solutions to, the problem of writing pointcuts for loops. In particular, the aim is to determine which characteristics can be used for making a loop selection. In aspect-oriented systems such as AspectJ, the means of selection for a join point is, in most cases, ultimately based on the naming of some source element characterising the join point, possibly using a regular expression. For example, to advise a method call or a group of methods, the pointcut has to contain an explicit reference to some names characterising the method signatures, whether it be a pattern matching the name of the methods, or a pattern matching the parameter types. Since loops cannot be named, it is impossible to use a name-based pattern to write a pointcut that would select a particular loop.

Neither loop labels, nor Java 5 (or C#) metadata, can be used to identify a particular loop in a method. Firstly, the loop labels will not be kept in the bytecode (and, in any case, they are rarely used, unless motivated by a `break` statement branching outside an inner loop). Secondly, Java 5 metadata cannot be applied to statements (apart from variable declarations).

If it is known for certain that all the loops within a method are to be advised, it would be possible, in AspectJ, to use pointcut constructs such as `withincode`

or `cflow`[10] to restrict the pointcut to all the loops contained in the methods traditionally picked up by those constructs. However, selecting only one of several loops within the same method turns out to be impossible without any further mechanism.

In order to solve this problem, it is proposed that selection of loops is made to rely on the data being processed, as well as the method in which it is located. In this case, the context —or what are called the "arguments" of the loop in Section 4.4— can be used to refine the selection. For example, the programmer might want to write a pointcut that would select only loops iterating over a specified range of integers, over a particular array, or over a particular `Collection`. Such an example is shown in Listing 4.11 (Section 4.9), in which the parallelising advice applies only to arrays of `bytes`.

More speculatively, there might be a potential application for metadata, which could be introduced in the declarations of the local variables that refer to the arrays, `Collections` or `Iterators` utilised as "arguments" to certain loops. Also, a data-based selection would benefit from a `dflow` pointcut, as described in [MK03].

In cases where the loop that is to be selected is within a loop nest, more selective `cflow` expressions could be used, for example using more expressive control-flow languages [DT04].

4.6   Issues related to exceptions

Without distinguishing edges due to exceptions from normal edges in the control flow graph, the model may not work properly in certain cases that involve exceptions.

Firstly, exceptions are handled by *traps* according to the position in the byte-code at which they are thrown. Each trap handles a linear portion of the bytecode, described only by a beginning and an end instruction. Thus, weaving may insert code within the range of a trap when this may not have been intended. Secondly, combined loops correspond approximately to loops written in the source code, as long as the graph is *reducible* (or *well-structured*). Because there is no "goto" statement [ASU85, Ch 10.4] in the Java language, this is the case for bytecode

---

[10]See Appendix A.

produced from Java source code.[11] However, taking exceptions into account (as possible causes for loops) adds extra edges to the graph. In the Soot representation [RH98][12], on which the prototype implementation presented in Section 4.7 relies, this may make the graph non-reducible [Jor03]. The main characteristics of non-reducible graphs are that: (a) loops may have several headers; and (b) there are still cycles in the graph after all the back edges have been removed.

As an illustration of the problems of exceptions, Listing 4.6 shows an example of code that involves loops and exceptions (taken from [MH02, Mie03]). Figure 4.6 shows the corresponding block-level control-flow graph for this example (including exceptions, shown as dashed lines) using the Jimple intermediate representation, as produced by the control-flow graph viewer included in the Soot framework.[13]

This example demonstrates what would happen if edges due to exceptions were treated as regular edges, and could therefore belong to loops. Without entering into the details of the syntax of Jimple, in this example, `i0` and `i1` represent i and j, respectively, in the Java source-code.

Listing 4.6: Example of nested loops involving exceptions.

```
public int foo (int i, int j) {
   while (true) {
      try {
         while (i < j)
            i = j++/i ;
      } catch (RuntimeException re) {
         i = 10 ;
         continue ;
      }
      break ;
   }
   return j ;
}
```

The back edges found using the method described in Section 4.3.1 are $4 \rightarrow 1$ and $5 \rightarrow 5$. The graph is not reducible because, after these back edges have been removed, a cycle made of nodes 1 and 5 exists. This gives a loop comprising nodes

---

[11]The use of Java bytecode obfuscators, which aim to prevent "easy" decompilation, might introduce "goto" statements into the bytecode, which may make the corresponding graph non-reducible.

[12]Soot is a Java bytecode analysis and transformation framework. It can be found at `http://www.sable.mcgill.ca/soot/`.

[13]This corresponds to the representation of `soot.toolkits.graph.ExceptionalBlockGraph`, in Soot 2.1.0.
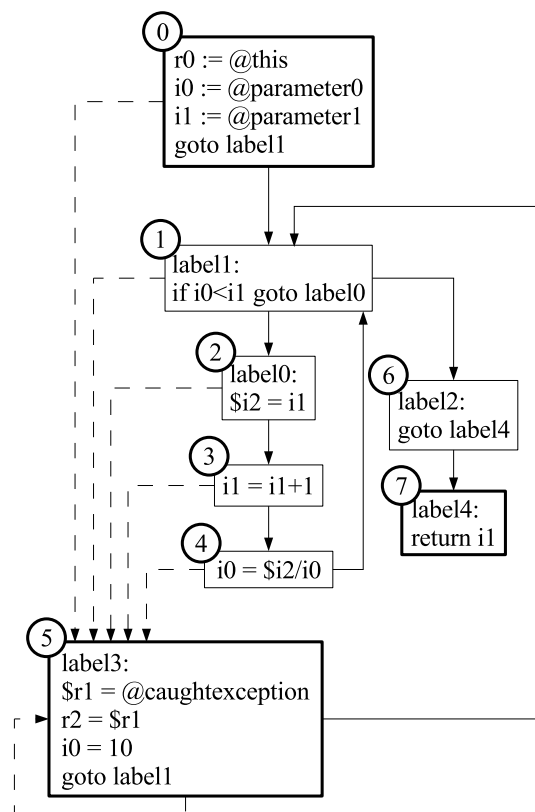
Figure 4.6: Complete block-level control flow graph.

1, 2, 3 and 4 —which corresponds to "while (i<j) i=j++/i;" in the source-code— and another loop comprising node 5 (which handles the exception in the source-code) only. Although the first loop is meaningful, and corresponds to what would be naturally expected by looking at the source-code, the second would cause *before-advice* to be inserted just before the exception is caught, and *after-advice* just before "continue" (without even dealing with the correctness of trap handling). This effect would not necessarily be meaningful or useful for advising this loop.

Moreover, such code is not robust to changes of compilation strategy. For example, a different compiler might insert an extra, "useless" goto statement between nodes 0 and 1 in this graph, yielding the control-flow graph shown in Figure 4.7. In this case there is a third back edge $(5 \to 8)$, which gives a natural loop that could be assimilated into the outer "while(true) { ... }" loop in the source-code. The method based on these control-flow graphs is not suitable for cases involving exceptions, since the loop join point model should depend as little as possible on the compilation strategy utilised and on the way branches due to exceptions are represented in the graph.
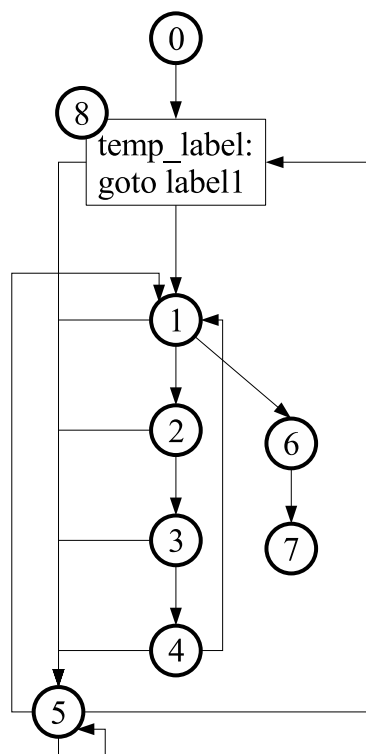
Figure 4.7: Another possible control-flow graph.

The problem with exceptions lies in the edges they add to the graph in the Soot representation [Jor03]. The documentation for class `soot.toolkits.graph.ExceptionalUnitGraph` [14] states: "*For every* `Unit` *which may implicitly throw an exception that could be caught by a* `Trap` *in the* `Body`, *there will be an edge from each of the excepting* `Unit`'s *predecessors to the* `Trap` *handler's first* `Unit` *(since any of those predecessors may have been the last* `Unit` *to complete execution before the handler starts execution).*" The edges coming from these predecessors that do not throw exceptions distort the dominator tree when trying to find the back edges. A possible solution would be to change this representation and to introduce separate nodes for throwing exceptions. For each node A that could potentially throw an exception represented as an edge from A to B, a new node E_A would be inserted before A, so that all the edges pointing to A would be redirected to E_A, and an extra edge $E_A \rightarrow B$ would be added. The resulting control-flow graph for the example in Listing 4.6 is sketched in Figure 4.8. This is similar to the graph in Figure 4.6, but contains extra nodes E_1, E_2, E_3 and E_4, which precede nodes 1, 2, 3 and 4, respectively, and represent the cases where an exception would be thrown in one of these nodes, thus preventing the operations in that node from being performed. This representation now gives two back edges ($4 \rightarrow E_1$ and $5 \rightarrow E_1$) corresponding to a single combined loop. Although the source code may seem to comprise two loops, finding only one loop is not surprising because the statement executed immediately after a successful evaluation of (`i<j`) to `false` is the final `return` statement. To avoid ambiguity, chains of unconditional `goto`s should be considered as a single node if they can all throw exceptions to the same catching blocks.

This particular example relies on exceptions being thrown to make the loop structure. This is quite unusual and can be confusing. The solution to this is to forbid the recognition of such loops and allow only loops made of normal branches. The handling of exceptions would then be handled as usual according to the AspectJ model, that is, using "`after returning`" and "`after throwing`".

4.7 Implementation in `abc`: LOOPSAJ

Although the loop join-point model could potentially be implemented in various aspect-oriented tools, based, for example, on Java or C#, the focus in this work

---

[14] The API documentation for Soot is available at: `http://www.sable.mcgill.ca/soot/doc/`.
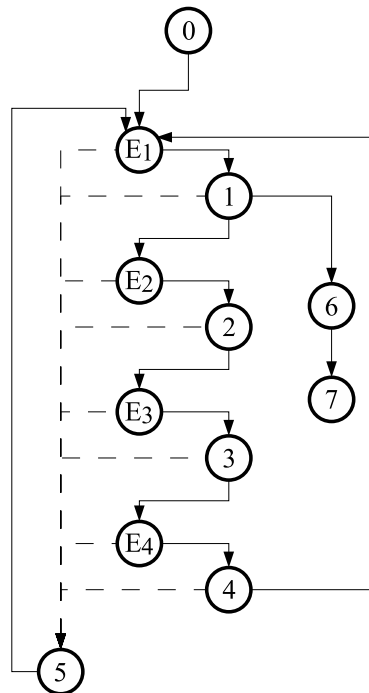
Figure 4.8: Control-flow graph with special nodes for exceptions.

has been put on a join point model integrable into AspectJ. The implementation uses abc,[15] an alternative AspectJ compiler, for two main reasons:

- extensibility was at the core of the abc design [ACH$^+$05b, ACH$^+$04a]; and

- abc relies heavily on the Soot framework [RH98], which provides most of the infrastructure for performing the analyses, in particular those described in Section 4.3.1.

This section describes an extension for abc, known as LOOPSAJ, which implements a loop join point for AspectJ and subsequently provides the loop() pointcut. The latter picks out loops with unique exit points (as described in Section 4.3.4) and provides contextual information where possible. Other pointcuts for the other forms of loops could also be provided (by lowering the degree of constraint imposed in the shadow matcher).

4.7.1   Shadow matching

The Soot framework, and subsequently abc, uses Jimple, which is a three-address representation of bytecode. This makes it possible to look for loops at bytecode

---

[15]http://www.aspectbench.org/

level (as described in Section 4.2). The shadow matcher and all pre- or post-transformations operate on this representation.

LOOPSAJ extends the `abc` method that finds the shadows in each method, so that it looks for loops as well. For each method processed, the control-flow graph and its corresponding dominator tree are built using the Soot framework toolkit. Then combined loops are identified, as described in Section 4.3.1.

`abc` provides two kinds of classes representing a shadow-match: `BodyShadow-Match` and `StmtShadowMatch` (both extend `ShadowMatch`). The former is utilised when the shadow is the whole method body; for example, when a method-execution pointcut is used. The latter is used for pin-pointing a specific statement (or group of statements) in the method; for example, when a method-call pointcut is used.

One of the requirements of `abc` is to insert `nop` operators in the shadow, at the points where *before* and *after* [16] pieces of advice might be woven. Given this, most of the `abc` infrastructure can already handle loop shadows for *before* and *after* pieces of advice.

However, handling *around* pieces of advice requires a few modifications in the `abc` around-weaver [Kuz04]. One of the cases where a group of statements is used is the constructor-call shadow match. In this case, two consecutive statements are included in the shadow-match. However, loop shadows are not necessarily formed by consecutive statements. Indeed, at bytecode or Jimple level, it is possible that the blocks forming a given loop are interleaved with blocks that do not belong to that loop. For this reason, `StmtShadowMatch` has been extended by `NonContiguousStmtGroupShadowMatch`, for which the around weaver has been modified.

Although the original `abc` can be compiled purely by a Java compiler, AspectJ is required for compiling LOOPSAJ. The handling of the new type of shadows and, more importantly, keeping the control-flow graph and the loop structures synchronised with the method content are implemented using aspects.

Indeed, the control-flow graph and the loop structures are created during the shadow matching. However, operations such as weaving modify the set of instructions in the methods. For example, weaving an *around* piece of advice is implemented by placing the statements that form the shadow into a separate method[17] and replacing them by an invocation of that method. Because the structure used

---

[16]It is not always possible to insert *after-advice*, as described in Section 4.3.

[17]This may be done with or without a *closure* [ACH⁺04b, ACH⁺05a].

by LOOPSAJ to represent the loops corresponds to the instructions that form the loop, this structure has to be updated to take into account this operation.

Although `abc` is well designed for producing separate extensions, there was originally nothing to update the graphs and set of instructions that constitute the loops when a modification is made. Such modifications are performed in various places in the code, and at various stages (for example, marking the shadows or weaving). The Soot representation —on which `abc` is based— uses class `Chain` to describe the sequence of instructions in a method. Its methods `insertBefore()`, `insertAfter()` and `remove()` are utilised for all transformations. Advising these using aspects makes it possible to implement a systematic and consistent behaviour each time a chain modification affects a loop, wherever and whenever this may be.

4.7.2 Context exposure and transformations

Exposing the context, as described in Section 4.4, depends on the cleverness of analysis and on the feasibility of certain transformations. For the context exposed to make sense, it has to be constant during execution of the join point.

In order to ensure this, as long as it is possible to predict that the transformation will not change the meaning of the loop, loop-invariant assignments are moved to the pre-header (before the shadow matching takes place), using a scheme inspired by that presented in [ASU85, Section 10.7].

4.7.2.1 Exposing the iteration space context

Further, in the case of a loop iterating over a range of integers, if the context values are numerical constants, temporary variables are introduced (and initialised in the pre-header), in order to make it possible to modify these values via calls to `proceed(...)` within an *around-advice*. An example transformation is shown in terms of source-code in Listing 4.7.

The part of the implementation that determines the feasibility of these transformations uses the dataflow analysis facilities provided by Soot; these have also been used to implement a code-motion method and a reaching-definition analysis [Muc97, ASU85].

Listing 4.7: Code-motion example.

```
/* Moving the invariants outside */

int i = 0 ;
while (i < 10) {
   /* ... */
   int stride = 3 ;
   i = i + stride ;
}



// -------------------------------------------------

/* First step: moving the invariants outside */

int i = 0 ;
int stride = 3 ;
while (i < 10) {
   /* ... */
   i = i + stride ;
}

// -------------------------------------------------


/* Second step: storing the values in temporary variables */

int stride = 3 ;
int minimum = 0 ;
int maximum = 10 ;
int i = minimum ;
while (i < maximum) {
    /* ... */
    i = i + stride ;
}
```

4.7.2.2 Exposing the originating "iterable" data context

It is not always possible to find an array to which the range of integers corresponds (i.e. when `minimum=0`, `stride=1`, and `maximum` is the length of the array). For example, if the bounds and the array are passed as arguments to the containing method, finding the array that was the origin of these values might require much more complex, cross-methods and points-to, analysis. The present implementation requires at least the statement initialising `maximum` to the length of the array to be within the same method as the loop.

Similarly, a `Collection` will only be exposed if the `Iterator` used for the loop comes from a call to `Collection.iterator()` and `Iterator.next()` is not called before the beginning of the loop.

4.7.2.3 Writing pointcuts

For loops iterating over a range of integers, the boundary values are passed via the `args` construct of AspectJ, to which `int` values are bound (for `minimum`, `maximum` and `stride`). Additionally, an extra argument will be bound to the originating array, if one has been found.

For loops iterating over an `Iterator`, the first argument of `args` will be bound to the corresponding instance of `Iterator`. Also, an extra argument will be bound to the originating `Collection`, if one has been found.

In cases where the originating array or `Collection` do not matter, it is recommended to use the double-dot wildcard notation ("..") [KHH+01b], to make the argument optional. For example:

- `loop() && args(min, max, stride)` will match only loops iterating over an arithmetic sequence of integers for which the compiler was unable to find an array (although one may exist);

- `loop() && args(min, max, stride, ..)` will match all the loops iterating over a particular arithmetic sequence of integers; and

- `loop() && args(min, max, stride, array)` will match all the loops iterating over an array, a reference to which will be bound to pointcut parameter "`array`".

Moreover, the way pointcuts are written to match certain loops can have an impact on the performance, as shown in Chapter 5.

4.7.3   Limitations

One of the main limitations of LOOPSAJ is the predictability on which the invariant code-motion is based. Although code-motion is currently performed successfully in most useful cases, it will not be performed in cases where an invariant is not discovered by the data analysis. The implementation of such transformations ought to be conservative, that is to say, it should not be done unless it is certain that the resulting code will be equivalent.

Another limitation is the lack of points-to analysis in respect of `Iterators`. Indeed, even though an `Iterator` instance may look as though it is being iterated over regularly in the loop (*i.e.* there is one and only one call to `next()` per iteration), nothing guarantees that no other thread is holding a reference to the same `Iterator` and is calling `next()` concurrently. In this respect, the exposure of `Iterators` is probably not sufficiently conservative. There might be a solution to this problem if a form of whole-program analysis were to be used. (This concurrency problem does not occur for loops iterating over a range of integers since `int` is a primitive type and the `int` values are local variables that cannot be modified by another thread.)

More generally, there could be further dependency analysis to provide safeguards in case of concurrent execution of a join point. Again, whole-program analysis may be required to be sure that a loop can be executed in parallel. This topic is addressed further in Section 4.8.

In addition, the recognition of loops based on bytecode makes it sensitive to loop transformations performed by the compiler. For example, an unrolled loop would not be matched by the loop pointcut. This limitation is analogous to that of AspectJ not being able to recognise an execution join point to a method that has been inlined.

In whatever way a weaver capable of handling loop join points is implemented, it should be stated clearly by the implementers how conservative their implementation is, and, in particular, how certain it is that a specified `Collection` is the origin of an `Iterator`.

4.8   Join point reflection and loop analyses

AspectJ provides reflection capabilities to obtain information about the join point within pieces of advice. This is achieved via the special variable `thisJoinPoint`,

of type `org.aspectj.lang.JoinPoint`. In particular, `thisJoinPoint` can give access to `args`, `this` and `target`, whether or not they have been bound to pointcut parameters. For example, in Listing 4.8, the first piece of advice uses reflection to get the join point arguments via `thisJoinPoint.getArgs()`, whereas the second and third pieces of advice gain access to the arguments by using the `args` pointcut and binding their value to the advice parameter (`t`). The fundamental difference between these three approaches is that:

- the first piece of advice is executed before all calls to `run()`, whatever the arguments may be;

- the second piece of advice is only executed before calls to `run()`, the argument of which is an instance of `Test1` or one of its subclasses (this test being performed at runtime); and

- the third piece of advice is only executed before calls to `run()`, the argument of which is an instance of `Test2` or one of its subclasses.

In particular, join point reflection can be used to handle specific cases within a piece of advice when its pointcut matches several join points of different types or with different types of arguments.

Another example is the array argument of the loop join point, as implemented in Section 4.7. The pointcut `loop() && args (min, max, stride, ..)` makes the matching of an array as the fourth parameter optional. It is, however, possible to obtain the value for the array representing the data over which the loop is iterating using `thisJoinPoint.getArgs()`, as shown in Listing 4.9.

Reflection can also provide more information about a join point via the *signature* of the join point.   The signature contains details in the woven code corresponding to the join point.   `Signature`[18] is an interface that "*parallels* `java.lang.reflect.Member`" [asp].   For example, `thisJoin-Point.getSignature()` returns subinterfaces implementing `MethodSignature` or `FieldSignature` when the join points correspond to something happening, respectively, on a method (`call` or `execution`) or on a field (`set` or `get`).   In turn, these subclasses can provide even more information about the join point (for example, the return type, in a method signature).

The signature of the loop join point can be used as a means to provide additional runtime information about the loop it represents. This is an opportunity

---

[18]`Signature` is part of the `org.aspectj.lang` API.

Listing 4.8: Example use of `thisJoinPoint`.

```java
public class Test {
   public static void run(Test1 t1) {
      System.out.println("Test.run(Test1): "+t1) ;
   }
   public static void run(Test2 t2) {
      System.out.println("Test.run(Test2): "+t2) ;
   }
   public static void main(String[] arg) {
      Test1 t1 = new Test1() ; Test2 t2 = new Test2() ;
      Test1 t2int1 = new Test2() ;
      System.out.println(" * A *") ; run(t1) ;
      System.out.println(" * B *") ; run(t2) ;
      System.out.println(" * C *") ; run(t2int1) ;
   }
   static class Test1 {
      public String toString() { return "Test1" ; }
   }
   static class Test2 extends Test1 {
      public String toString() { return "Test2" ; }
   }
}

aspect TraceTest {
   pointcut calltorun(): call(* Test.run(..)) ;
   before(): calltorun() {
      // Matches all calls to Test.run(..)
      System.out.println("Call to run with args: "
          +java.util.Arrays.asList(thisJoinPoint.getArgs())) ;
   }
   before(Test.Test1 t): calltorun() && args(t) {
      // Matches calls to Test.run(Test1) and Test.run(Test2)
      System.out.println("Call to run, before(Test1): "+t) ;
   }
   before(Test.Test2 t): calltorun() && args(t) {
      // Matches calls to Test.run(Test2)
      System.out.println("Call to run, before(Test2): "+t) ;
   }
}

/*        Result of execution:
 * A *
Call to run with args: [Test1]
Call to run, before(Test1): Test1
Test.run(Test1): Test1
 * B *
Call to run with args: [Test2]
Call to run, before(Test1): Test2
Call to run, before(Test2): Test2
Test.run(Test2): Test2
 * C *
Call to run with args: [Test2]
Call to run, before(Test1): Test2
Call to run, before(Test2): Test2
Test.run(Test1): Test2
 */
```

Listing 4.9: Example use of `thisJoinPoint` with LOOPSAJ.

```
import java.util.* ;

public aspect LoopsAJTestAspect {
   before(): loop()   {
      System.err.println("loop(). Arguments: "
            +Arrays.asList(thisJoinPoint.getArgs())) ;
   }

   before(int min, int max, int stride): loop()
                                    && args (min, max, stride) {
      System.err.println("loop() "
            +"with min, max, stride but no array found.") ;
   }

   before(int min, int max, int stride, double[] array): loop()
                                  && args (min, max, stride, array) {
      System.err.println("loop() "
            +"with min, max, stride and array of double found.") ;
   }

   before(int min, int max, int stride): loop()
                                    && args (min, max, stride, ..) {
      System.err.println("loop() "
            +"with min, max, stride found. Maybe there is an array.") ;
      Object[] arguments = thisJoinPoint.getArgs() ;
      if (arguments.length>=4) {
         System.err.println("  An array has been found: "
                                  +arguments[3]) ;
      } else {
         System.err.println("  No array has been found.") ;
      }
   }
}
```

to perform further analyses of the loop, which may be used within an advice to choose a parallelisation strategy. Analyses such as those presented in [BG97] can be performed during the shadow matching. Their outcome can be passed to the runtime environment via the join point signature, as follows:

Analysis of exceptions. It can be useful to determine in advance whether certain exceptions can be raised within the loop.[19] For example, if it is guaranteed that no `ArrayIndexOutOfBoundsException` can occur within a loop, dealing with this kind of exception is unnecessary.

Analysis of scalar variables. The prototype implementation does not allow *around* advice to be used where a local variable which is used after the loop may be redefined within the loop. Further scalar analyses could make it possible to relax this condition and let the aspect programmer decide what to do.

Analysis of arrays. The signature could also comprise information about loop carried data dependencies and load-imbalance.

A *hybrid approach* to aspects for parallelisation could combine the strength of the compiler analyses and the flexibility of aspects.

4.9   Aspects for parallelisation

This section presents an application of the `loop()` pointcut, namely parallelisation of loops.

The example advice shown in Listing 4.10 executes in parallel (using Java threads with cyclic loop scheduling) all the loops contained in class `LoopsAJTest` which are recognised as iterating over a range of integers. As shown, the `loop()` pointcut combines ideally with the "*worker object creation pattern*" [Lad03], which creates new `Runnables` to execute join points on separate threads.

The aspect shown in Listing 4.11 is slightly more complex. It executes in parallel, using MPI for Java,[20] the loops working on an array of `bytes` that are in method `LoopsAJTest.test`. The original array, `a`, is exposed to the pointcut. It is then sliced into an array `p` within each MPI task. Thereafter, `proceed()` uses array `p` instead of `a`, so that the loop in each MPI task only iterates over its local portion of `a`.

---

[19]More generally, this could be applied to other join points, as well.

[20]`http://www.hpjava.org/mpiJava.html`

Listing 4.10: Loop parallelisation using Java Threads.

```
void around(int min, int max, int step):
  within(LoopsAJTest)
  && loop() && args (min, max, step, ..) {
    int numThreads = 4 ;
    Thread[] threads = new Thread[numThreads] ;
    for (int i = 0 ; i<numThreads ; i++) {
      final int t_min = min+i ;
      final int t_max = max ;
      final int t_step = numThreads*step ;
      Runnable r = new Runnable () {
        public void run() {
          proceed(t_min, t_max, t_step) ;
        }
      } ;
      threads[i] = new Thread(r) ;
    }
    for (int i = 1 ; i<numThreads ; i++) {
      threads[i].start() ;
    }
    threads[0].run() ;
    try {
      for (int i = 1 ; i<numThreads ; i++) {
        threads[i].join() ;
      }
    } catch (InterruptedException e) {  }
  }
```

Listing 4.11: Loop parallelisation using mpiJava.

```
import mpi.* ;

aspect MPIParallel {
  int rank ;
  int NP_COUNT ;

  void around(String[] arg):
      execution(void LoopsAJTest.main(..)) && args(arg) {
    try {
      MPI.Init(arg);
      rank = MPI.COMM_WORLD.Rank();
      NP_COUNT = MPI.COMM_WORLD.Size();

      proceed(arg) ;

      MPI.Finalize();
    } catch (MPIException e) { /* ... */ }
  }

  void around(int min, int max, int stride, byte[] a):
        loop() && args(min, max, stride, a, ..) &&
        withincode(* LoopsAJTest.test(..)) {
    try {
      MPI.COMM_WORLD.Barrier();
      int slice_length = a.length / NP_COUNT ;
      byte[] p = new byte[slice_length] ;
      if (rank == 0) {
        for (int i = 0 ; i < slice_length ; i++)
           p[i] = a[i] ;
        for (int k = 1; k < NP_COUNT; k++)
           MPI.COMM_WORLD.Ssend(a, k*slice_length,
                      slice_length, MPI.BYTE, k, k) ;
      } else {
        MPI.COMM_WORLD.Recv(p, 0, slice_length,
                   MPI.BYTE, 0, rank) ;
      }

      proceed(0, slice_length, 1, p) ;

      MPI.COMM_WORLD.Barrier();
      if (rank == 0) {
        for (int i = 0; i < slice_length; i++)
           a[i] = p[i];
        for (int k = 1; k < NP_COUNT; k++)
           MPI.COMM_WORLD.Recv(a, k*slice_length,
                slice_length, MPI.BYTE, k, k);
      } else {
         MPI.COMM_WORLD.Ssend(p, 0,
                slice_length, MPI.BYTE, 0, rank);
      }
      MPI.COMM_WORLD.Barrier();
    } catch (MPIException e) { /* ... */ }
  }
}
```

When using these kinds of aspects, the programmer needs to make sure that the loops that are going to be executed in parallel can actually be parallelised. This can be helped by using reflection, where this provides results of more complex analyses (see Section 4.8).

Other examples of aspects for parallelisation are presented in Chapter 5.

4.10   Related topics

This section explores two related potential fine-grained join points (i.e. join points that recognise complex behaviour within a method and not only at the interface of the object), namely a "loop-body" join point (Section 4.10.1), and an "if-then-else" join point (Section 4.10.2).

4.10.1   "Loop-body" join point

The model of loop join point presented thus far takes an outside view of the loop; the points *before* and *after* the loop are not within the loop itself. As a consequence, however many iterations there may be for a given loop, *before* and *after*-advice will be executed only once. For some applications, for example for inserting a piece of advice before each iteration, it might be desirable to advise the loop body. However, the semantics would be difficult to define.

Even in the source-code, there is ambiguity about where to weave *before* and *after* advice in such a case. For example, is the termination condition in the loop-body or not? (see Listing 4.12). This question is even more pertinent for complex conditions that may include calls to methods.

Again, a basic-block control-flow approach may solve the problem. It may be possible to define that "before" the loop-body is the point at the beginning of the header, included in the loop, and that "after" the loop-body is a point inserted on the back edge of the natural loop. If there were several back edges in the corresponding combined loop, an equivalent of the "pre-header" could be inserted between the back edges and the header, in order to keep a single weaving point. In the case of a `while`-loop or a `for`-loop, "before" the loop-body would also be before the evaluation of the condition.

Without any enhancement, such a model would not comprise any contextual information (or "arguments" to the loop-body).

Listing 4.12: Loop-body join point: where are "before" and "after"?

```
int i = 0;
while (i<2) {
   /* Is ''before'' the loop-body right here, or
      should it be before (i<2) is evaluated?     */
   System.out.println("i: "+i) ;
   i++ ;
   /* Is ''after'' the loop-body here? Would ''i++''
      be included in the loop-body of the equivalent
      for-loop?       */
}

i = 0 ;
do {
   /* Before the loop-body */
   System.out.println("i: "+i) ;
   i++ ;
   /* Is ''after'' here, or should it be after (i<2)
      has been evaluated?    */
} while (i<2) ;
```

An application of this model could be to assert loop invariants by using aspects, provided that the elements upon which the conditions depend can be made visible to the aspects.

4.10.2 "If-then-else" join point

Why stop at loops? Similar techniques could be applied so as to provide aspect-oriented languages such as AspectJ with a model for an "if-then-else" join point.

At source-code level, there is again the question of whether the evaluation of the condition should or should not be included in the "if-then-else" join point.

A basic-block control-flow approach may help to define a model. A possible way to find the shadows of "if-then-else" constructs might be in the combined use of dominators and *postdominators*. "*[We] say that node* p postdominates *node* i *[...] if every possible execution path from* i *to [the exit] includes* p" [Muc97]. Given a node a that branches conditionally to other nodes (unconditional branching presents no interest), the smallest subgraph G of the control-flow graph that contains another node b such that a dominates all the nodes in G and b postdominates a, would represent an area of conditional execution, starting from a and joining back at b. Since a would dominate all the nodes in G, it would be the unique entry node to G. Since b would postdominate a, b would be the unique exit node from G. Just

before the conditional jump in a would be the *before* weaving point, and just before b (for edges coming from inside G) would be the *after* weaving point.

Again, it is unclear what kind of contextual information could be included in such a model. Without it, such an "if-then-else" join point would represent areas of code that will only be partially executed (for a given (dynamic) join point).

However, going a step further, by making it possible to advise `goto` statements directly in the bytecode, may break modularity and consistency, even within a method, which would counteract the benefits of using aspects.

Apart from the usual debugging and tracing applications of such join points, another successful approach for defining fine-grained join points (including conditional `if` blocks) has been applied to code-coverage analysis [RS05].

4.11   Summary

This chapter demonstrates that it is possible to provide AspectJ (and, theoretically, other aspect-oriented systems) with a join point for loops, which can be applied, in particular, for loop parallelisation.

The main remaining difficulties are: (a) the cleverness of the context analysis, and (b) the mechanisms for selecting loops. The context analysis is mostly implementation-dependant. Reflective mechanisms added to the join points can be a powerful means of conveying the results of deeper analyses to the aspect, without extending the language further than the new pointcuts. But the loop selection problem is more fundamental, especially because loops cannot be named or tagged. Practical uses of this join point, and how to write pointcuts for selecting only certain loops are presented in Chapter 5.

More generally, this chapter shows that join points need not be limited to "simple" operations, but can also address more complex behaviours.

The most fundamental remaining problem lies in the mechanisms for selecting loops, especially because loops cannot be named or tagged. Such fine-grained problems would benefit from other pointcut mechanisms, in particular more expressive data-flow and control-flow pointcut description mechanisms. In addition, loops based on recursion constitute a different problem that would also benefit from more advanced pointcut mechanisms, but would not necessarily require join points other than `call` and `execution`.

The other limitations of this model are mainly due to relying on the bytecode for recognising the loops. This design decision has been made for the same reasons that it has been made in AspectJ, namely, making something applicable to a wider code base (especially that for which the source is not available) and less sensitive to changes in programming styles. These limitations of LoopsAJ can be compared to certain limitations of AspectJ, in particular, the inability to weave after an exception handler[21] and the inability to match a call to a private method of an inner class from another inner class in the same class[22]. In the process of developing an application, the concern of keeping track of a loop structure crosscuts the compilation stage and the weaving stage. A potential solution to such problems might consist of passing more information from source-code to bytecode, perhaps by using specific attributes in the class-file format, or mechanisms such as Soot tags in `abc`, at the expense of the possibility to use aspects in arbitrary bytecodes.

---

[21]See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=33636`.
[22]See `https://bugs.eclipse.org/bugs/show_bug.cgi?id=124845`.

Chapter 5

Applications and performance evaluation

This chapter investigates applications of the techniques presented in Chapters 3 and 4. It provides examples of the flexibility of aspects for parallelisation and evaluates the performance costs incurred by the techniques employed for enabling the applications to be parallelised using aspects.

The example aspects for parallelisation presented so far utilise Java-threads or MPI. However, Section 5.1 demonstrates the flexibility of aspects by showing example aspects for other parallelisation schemes, which can also be re-used in many applications and woven on demand, without tangling the original code. The remainder of the chapter studies the performance obtained using the techniques presented in Chapters 3 and 4.

Although some of the theory presented in Chapters 3 and 4 is not limited to a Java environment, all the examples used are specifically based on AspectJ and Java class-files.[1]

Two main factors have been found to have a substantial influence on the timing results obtained:

1. Because the examples used are Java-based, the performance costs depend on which Java Virtual Machine (JVM) and which processor architecture are used. Although the core principle of Java is *"write once, run everywhere"*, the oblivious behaviour of the virtual machine (with respect to the application)

---

[1]Section 1.4 gives background information about Java with respect to high-performance computing.

incurs its own side-effects, such as the unpredictable effects of certain opti-misations. As shown throughout the test-cases presented in this chapter (in Sections 5.5, 5.6 and 5.7), the choice of JVM (and platform) can completely change the results: in some Java environments[2], refactoring an application to enable the use of parallelising aspects introduces almost no overhead; in others, the simple act of doing these refactorings can quadruple the execution time. The choice of Java environment can be considered as an external factor, on which the application programmer has limited influence.

2. The ways in which the numerical code and the aspects are written can have an impact on the execution speed of the applications. This factor is evidently influenced by the application programmer. Two strategies can be adopted for enabling an application to be parallelised using aspects:

   (a) it can be refactored so as to expose the iteration space of the loops to execute in parallel at the interface of the classes —this is the technique presented in Chapter 3, which makes it possible to use AspectJ as it is— or

   (b) the loops can be advised directly using an aspect compiler that imple-ments the join point model for loops presented in Chapter 4—in general, there are several different ways of writing pointcuts using this approach (this is reflected in particular by the results presented in Section 5.5).

After describing the general requirements for writing aspects for parallelisation using refactored code for AspectJ (in Section 5.2) and using LOOPSAJ —the pro-totype implementation of the join point for loops— (in Section 5.3), this chapter presents test-cases that aim to compare these two techniques, with each other, and with the standard Java implementation.

Three test-cases are presented in Sections 5.5, 5.6 and 5.7; these sections each describe one application and present performance results for it. The performance results show both the cost of using aspects in the sequential code, and the execution time in multi-threaded parallel configurations. (Performance results for aspects using MPI have been omitted because mpiJava was not fully functional with the combination of JVM and machines used for the experiments.) In Sections 5.6 and 5.7, performance results obtained using the refactoring method (as described

---

[2] *"Java environment"* is used to encompass all the layers from processor-type to the virtual machine of a particular vendor, through the operating system.

in Chapter 3 and Section 5.2) are compared with results obtained using LOOPSAJ aspects (as described in Chapter 4 and Section 5.3).

5.1 Aspects for flexibility in implementing parallelisation strategies

Chapters 3 and 4 have shown that it is possible to use aspects for choosing a parallelisation scheme. The examples showed either a multi-threaded aspect or an MPI aspect, which would parallelise the application using multiple Java threads or MPI, respectively. This section aims to demonstrate the flexibility introduced by the use of aspects by showing other parallelisation strategies.

The examples in this section provide several aspects that can parallelise two similar applications differently. These aspects are made abstract, in a manner similar to abstract classes in Java. In abstract aspects, the advice is fixed but the pointcut descriptor is abstract and has to be concretised in non-abstract sub-aspects. In the following examples, the abstract pointcut (`loopsToParallelise`) is expected to select the loops to parallelise and takes three arguments (the minimum, the maximum and the stride of the loop). These abstract aspects are therefore independant of the choice between the refactoring techniques (as seen in Chapter 3) or LOOPSAJ (as seen in Chapter 4). Four aspects have been written, each one implementing a different parallelisation strategy, as follows:

- aspect `ThreadBlockScheduling`, shown in Listing 5.1, parallelises the selected loop using block scheduling, each block being executed in a `Thread` that is managed in the advice;

- aspect `ThreadPoolBlockScheduling`, shown in Listing 5.2, parallelises the selected loop using block scheduling, each block being executed via a `Runnable` object run in a thread-pool;

- aspect `ThreadPoolCyclicScheduling`, shown in Listing 5.3, parallelises the selected loop using cyclic scheduling, each part being executed via a `Runnable` object run in a thread-pool;

- aspect `ForkJoinBlockScheduling`, shown in Listing 5.4, parallelises the selected loop using block scheduling in a Fork-Join parallel task, possibly

with more blocks than threads in order to take advantage of the queue of tasks in the Fork-Join framework [Lea00].

Other parallelisation aspects could be implemented, for example using a distributed Fork/Join framework [LMGF05]. Each of these four aspects can be plugged in or removed at will, since their inclusion in the application is optional. The flexibility introduced makes it possible to choose and adapt a parallelisation strategy depending on the application, without introducing any code-tangling in the computational units.

The examples chosen to illustrate the use of these parallelisation aspects consist of a dense matrix multiplication (shown in Listing 5.5) and a triangular matrix multiplication (shown in Listing 5.6), although these aspects could be applied to a large number of applications that contain parallelisable loops. All of the four abstract aspects presented above could be used, but only two concrete aspects are presented, using the thread block scheduling (shown in Listing 5.7) and the fork-join block scheduling (shown in Listing 5.8), respectively. These concrete aspects use LOOPSAJ.

In the triangular matrix multiplication, a block scheduling using as many blocks as threads is not the best solution, since the blocks do not require the same amount of computation. Plugging in the aspect for cyclic scheduling, or the aspect for using the Fork-Join framework, would be a better choice in this case.

Listing 5.1: Aspect for parallelisation using block scheduling.

```
public abstract aspect ThreadBlockScheduling {

   abstract pointcut loopsToParallelise(int min, int max, int stride) ;

   public final int THREADS_COUNT ;
   public ThreadBlockScheduling() {
      THREADS_COUNT=Integer.parseInt(System.getProperty("threads","1"));
   }

   void around(int min, int max, final int stride):
             loopsToParallelise(min, max, stride) {
      Thread[] threads = new Thread[THREADS_COUNT] ;

      int chunk_length = ((max-min)/(THREADS_COUNT*stride))*stride ;
      if (((max-min)%THREADS_COUNT)!=0)
         chunk_length += stride ;

      for (int k = 0 ; k < THREADS_COUNT ; k++) {
          final int slice_min = min + k*chunk_length;
          int temp_max = min + (k+1)*chunk_length;
          if (temp_max>max) temp_max = max ;
          final int slice_max = temp_max;
          thread[k] = new Thread(new Runnable() {
              public void run() {
                   proceed(slice_min, slice_max, stride) ;
              }
          }) ;
      }

      try {
         for (int k = 1 ; k < THREADS_COUNT ; k++) {
            thread[k].start() ;
         }
         thread[0].run() ;
         for (int k = 1 ; k < THREADS_COUNT ; k++) {
            thread[k].join() ;
         }
      } catch (InterruptedException ie) {/* ... */}
   }
}
```

Listing 5.2: Aspect for parallelisation in a thread-pool using block scheduling.

```
public abstract aspect ThreadPoolBlockScheduling {

  abstract pointcut loopsToParallelise(int min, int max, int stride) ;

  public final int THREADS_COUNT ;
  private final ThreadPool threadPool ;
  public ThreadPoolBlockScheduling() {
     THREADS_COUNT=Integer.parseInt(System.getProperty("threads","1"));
     threadPool = new ThreadPool(THREADS_COUNT) ;
  }

  void around(int min, int max, final int stride):
          loopsToParallelise(min, max, stride) {
     Runnable[] runnables = new Runnable[THREADS_COUNT] ;

     int chunk_length = ((max-min)/(THREADS_COUNT*stride))*stride ;
     if (((max-min)%THREADS_COUNT)!=0)
        chunk_length += stride ;

     for (int k = 0 ; k < THREADS_COUNT ; k++) {
         final int slice_min = min + k*chunk_length;
         int temp_max = min + (k+1)*chunk_length;
         if (temp_max>max) temp_max = max ;
         final int slice_max = temp_max;
         runnables[k] = new Runnable() {
             public void run() {
                 proceed(slice_min, slice_max, stride) ;
             }
         } ;
     }

     threadPool.run(runnables) ;
  }
}
```

Listing 5.3: Aspect for parallelisation in a thread-pool using cyclic scheduling.

```
public abstract aspect ThreadPoolCyclicScheduling {

  abstract pointcut loopsToParallelise(int min, int max, int stride) ;

  public final int THREADS_COUNT ;
  private final ThreadPool threadPool ;
  public ThreadPoolCyclicScheduling() {
     THREADS_COUNT=Integer.parseInt(System.getProperty("threads","1"));
     threadPool = new ThreadPool(THREADS_COUNT) ;
  }

  void around(int min, final int max, final int stride):
          loopsToParallelise(min, max, stride) {
     Runnable[] runnables = new Runnable[THREADS_COUNT] ;

     for (int k = 0 ; k < THREADS_COUNT ; k++) {
         final int local_min = min + k*stride;
         runnables[k] = new Runnable() {
             public void run() {
                 proceed(local_min, max, stride*THREADS_COUNT) ;
             }
         } ;
     }

     threadPool.run(runnables) ;
  }
}
```

Listing 5.4: Aspect for parallelisation using the Fork-Join framework.

```
import EDU.oswego.cs.dl.util.concurrent.*;

public abstract aspect ForkJoinBlockScheduling {

   abstract pointcut loopsToParallelise(int min, int max, int stride) ;

   public final int THREADS_COUNT ;
   public final int NSLICES ;
   private final FJTaskRunnerGroup taskrunner ;
   public ForkJoinBlockScheduling() {
      THREADS_COUNT=Integer.parseInt(System.getProperty("threads","1"));
      NSLICES = Integer.parseInt(System.getProperty("slices","1")) ;
      taskrunner = new FJTaskRunnerGroup(THREADS_COUNT) ;
   }

   void around(int min, int max, final int stride):
           loopsToParallelise(min, max, stride) {
      FJTask[] tasks = new FJTask[NSLICES] ;

      int chunk_length = ((max-min)/(NSLICES*stride))*stride ;
      if (((max-min)%NSLICES)!=0)
         chunk_length += stride ;

      for (int k = 0 ; k < NSLICES ; k++) {
         final int slice_min = min + k*chunk_length;
         int temp_max = min + (k+1)*chunk_length;
         if (temp_max>max) temp_max = max ;
         final int slice_max = temp_max;
         tasks[k] = new FJTask() {
            public void run() {
               proceed(slice_min, slice_max, stride) ;
            }
         } ;
      }

      FJTask parTasks = FJTask.par(tasks) ;

      try {
         taskrunner.invoke(parTasks) ;
      } catch (InterruptedException ie) {/* ... */}
   }
}
```

Listing 5.5: Multiplication of two dense matrices.

```java
public class DenseMatrixMultiplication {
    public void run() {
        int N = /* ... */
        double[][] a = /* an N*N matrix */
        double[][] b = /* an N*N matrix */
        double[][] c = /* an empty N*N matrix */
        for (int i = 0; i<c.length; i++) {
            double row[] = c[i] ;
            for (int j = 0 ; j<=row.length ; j++) {
                double sum = 0 ;
                for (int k = 0 ; k<N ; k++) {
                    sum += a[i][k] * b[k][j] ;
                }
                c[i][j] = sum ;
            }
        }

        /* ... */
    }
}
```

Listing 5.6: Multiplication of two triangular matrices.

```java
public class TriangularMatrixMultiplication {
    public void run() {
        int N = /* ... */
        double[][] a = /* an N*N lower triangular matrix */
        double[][] b = /* an N*N lower triangular matrix */
        double[][] c = /* an empty N*N lower triangular matrix */
        for (int i = 0; i<c.length; i++) {
            for (int j = 0 ; j<=i ; j++) {
                double sum = 0 ;
                for (int k = 0 ; k<N ; k++) {
                    sum += a[i][k] * b[k][j] ;
                }
                c[i][j] = sum ;
            }
        }

        /* ... */
    }
}
```

Listing 5.7: Aspect for multiplying matrices in parallel using block scheduling.

```
public aspect MatrixThreadBlockScheduling extends ThreadBlockScheduling{
    pointcut withinrun():
        withincode(void TriangularMatrixMultiplication.run(..)) ;
    pointcut doublearrayloop(): loop() && args(*,*,*,double[][]) ;

    pointcut loopsToParallelise(int min, int max, int stride):
        withinrun() &&
        doublearrayloop() &&
        args(min, max, stride, *) ;
}
```

Listing 5.8: Aspect for multiplying matrices in parallel using the Fork-Join framework.

```
public aspect MatrixFJBlockScheduling extends ForkJoinBlockScheduling {
    pointcut withinrun():
        withincode(void TriangularMatrixMultiplication.run(..)) ;
    pointcut doublearrayloop(): loop() && args(*,*,*,double[][]) ;

    pointcut loopsToParallelise(int min, int max, int stride):
        withinrun() &&
        doublearrayloop() &&
        args(min, max, stride, *) ;
}
```

5.2 Aspects for refactored code, using AspectJ

The *Red-Black* test-case has been used to measure and compare the performance of the object-oriented loop models, as presented in Section 3.2, with the performance of the equivalent loops written in the traditional way. The models are rectangle-based, which matches the iteration space of the Red-Black algorithm.

The *Crypt* application has been used to measure the impact of refactorings, as shown in Section 3.1.

Both of these kinds of refactoring make it possible to use AspectJ, as it stands, for writing aspects for parallelisation. As shown in Chapter 3, the key technique for writing aspects for parallelism using AspectJ consists of exposing the iteration space as a parameter to a method which solely contains the loop that is to be parallelised.

5.3 Aspects for the join point for loops, using LoopsAJ

As described in Section 4.5, writing pointcuts to select specific loops can be difficult. The way pointcuts are written can also influence the performance obtained, in particular when `cflow`-related constructs are used.

Listing 5.9: Writing pointcuts for parallelisation using the join point for loops.

```
public void process(double[] array, int N) {
    outer:
    for (int i = 0 ; i < array.length ; i++) {
        inner:
        for (int j = 1 ; j < N ; j ++) {
            /* Do something with the array */
        }
    }
}
```

For example, consider the loops shown in Listing 5.9. The pointcuts to select the outer loop in method `process` can be of two forms:

1. *data-based*:

    pointcut loopOnArrayOfDouble(double[] a):

    loop() && **args**(*,*,*,a) ; or

2. *cflow-based*:
```
pointcut outerLoop():
loop() && !cflowbelow(loop()) ;
```

The first pointcut, `loopOnArrayOfDouble(double[] a)`, makes the selection according to the data type handled by the loop. Although the data type check is performed at run-time (via `instanceof`), this could be optimised and determined at compile type. Indeed, `double[]` has no subtype, and its only supertype is `Object`, thus, declaring the variable to be of type `double[]` at compile-time guarantees the same type at runtime. No additional runtime checks are required to check whether the pointcut matches the inner loop.

The second pointcut, `outerloop()` makes the selection according to the state of the control flow. "`loop() && !cflowbelow(loop())`" selects all the loops that are not below the control flow of any loop, which is exactly what is required to match outer loops. However, the implementation of `cflow` and `cflowbelow` relies on a counter, or a stack, which is incremented and decremented on entry and exit, respectively, of the join point described within `cflow()`, that is, in this case, all the loops (with a unique exit). A test to check the value of this counter is performed before entering each loop. In the example shown in Listing 5.9, an extra test of the `cflow` counter is made on every iteration of the outer loop in order to check if the pointcut also matches the inner loop.

As the results presented in Sections 5.5 and 5.7.2 demonstrate, choosing either one of these two ways of writing the pointcut descriptors can have an impact on the performance obtained, depending on the optimisation strategies of the JVM and on the test-case. Extensive work on optimising `cflow`-related implementations has been done for `abc` [ACH+04b, ACH+05a] and has been integrated subsequently into `ajc`[3].

Independently of the choice between these two ways of writing pointcuts, Sections 5.6.2 and 5.7.2 present applications and performance results for three variants of the test-cases:

1. without any aspect —this is the *reference*;

2. with an aspect that advises the loop to be parallelised but uses only `proceed()` (see Listing 5.10) —this shows the *overhead of only weaving*;

---

[3]`ajc` is the original AspectJ compiler.

3. with an aspect that advises the loop to be parallelised so that it is executed in threads (see Listing 5.11) —this shows the *full overhead* of weaving, creating new instances of `Runnable` and executing them in multiple threads.[4]

Listing 5.10: Aspect that simply proceeds with the original join point execution.

```
public aspect ProceedOnly {
   /*
    ...
    Definition of pointcut loopstoparallelise()
    ...
    */

   void around(final int min, final int max, final int stride):
         loopstoparallelise() && args(min, max, stride, ..) {
     proceed(min, max, stride) ;
   }
}
```

---

[4]Because the refactored version of the Red-Black application has been tested with a thread pool, the LOOPSAJ aspect for parallelising this application also uses a thread pool.

Listing 5.11: Aspect that splits the loop recognised by the pointcut into blocks and executes it in several threads.

```
public aspect ExecuteLoopInThreadPool {
   /*
     ...
     Definition of pointcut loopstoparallelise()
     Initialisation of THREADS_COUNT
     ...
    */

   void around(final int min, final int max, final int stride):
          loopstoparallelise() && args(min, max, stride, ..) {

      Thread threads[] = new Thread[THREADS_COUNT] ;

      int chunk_length = ((max-min)/(THREADS_COUNT*stride))*stride ;
      if (((max-min)%THREADS_COUNT)!=0)
         chunk_length += stride ;

      for (int k = 0 ; k < THREADS_COUNT ; k++) {
         final int slice_min = min + k*chunk_length;
         int temp_max = min + (k+1)*chunk_length;
         if (temp_max>max) temp_max = max ;
         final int slice_max = temp_max;
         threads[k] = new Thread(new Runnable() {
            public void run() {
               proceed(slice_min, slice_max, stride, array) ;
            }
         }) ;
      }

      try {
         for (int k = 1 ; k < THREADS_COUNT ; k++) {
            threads[k].start() ;
         }
         threads[0].run() ;
         for (int k = 1 ; k < THREADS_COUNT ; k++) {
            threads[k].join() ;
         }
         } catch (InterruptedException e) {
         }
   }
}
```

5.4 Experimental environment

5.4.1 Machines

Four machines have been used for running the experiments presented in the remainder of this chapter. Their configurations are as follows:

- a PC with a single Athlon XP2500+ processor (32-bit x86 compatible) at 1.83 GHz, with 512 KB of cache, with 1 GB of RAM running Linux (kernel 2.6.9);

- a PC with two Pentium-III (Coppermine) processors at 870 MHz, with 256 KB of cache each, sharing 512 MB of RAM and running Linux (kernel 2.4.18);

- a Sun server with 4 UltraSparc processors at 450 Mhz, with 8 MB of cache each, sharing 1 GB of RAM and running SunOS (Solaris) 5.8; and

- an SGI Origin 3400 with 16 MIPS processors at 400 Mhz, with 8 MB of L2 cache each, sharing 4 GB of RAM and running Irix 6.5.

5.4.2 Java virtual machines

Three brands of JVM have been used for running the experiments presented in the remainder of this chapter, namely Sun, IBM and SGI.

Sun JVMs are available for both the x86 (Pentium-III and Athlon) and Sparc architectures. Versions 1.4.2 and 1.5.0 have been used. All Sun JVMs come with client and server modes.[5] The Sun JVMs for the Sparc machine also have a 64-bit mode (which is a variant of the server mode).

The IBM JVM is only available for the x86 architectures of the above machines.[6] Version 1.4.2 has been used (this is the latest version available at the time of writing).

The SGI JVM is only available for the SGI machine. It is based on the Sun HotSpot VM, and is available only in client mode [SSTP02].

---

[5]The Java Hotspot Virtual Machine [Hot] provides two just-in-time compilers: the *client* compiler, which is faster, and the *server* compiler, which performs stronger optimisations. The server version is tuned for server applications, where it can be worth spending more time on the compilation if the resulting code is better optimised.

[6]It is in fact available for other types of machines that were not at our disposal.

5.4.3 Compilers

The compilers that have been used for the experiments presented in the remainder of this chapter are:

- `ajc`: the original AspectJ compiler;[7]

- `abc`: the AspectBench Compiler,[8] which is an alternative compiler for the AspectJ language; and

- LOOPSAJ: an extension to `abc` which contains a prototype implementation of the loop join point presented in Chapter 4.

5.5 Test-case: data-based vs. `cflow`-based selection in LOOPSAJ

In order to show the impact on performance of choosing either the data-based or the `cflow`-based means of writing pointcuts using LOOPSAJ, a very simple example has been utilised. The source code for this example is shown in Listing 5.12. Method `run()`, the execution time of which is measured, contains a nest of three loops, which are referred to in the following as the `i`-loop, the `j`-loop and the `k`-loop, according to the name of the index used.

Three sets of tests have been performed:

1. without any aspect (compiled with `abc` and the LOOPSAJ extension);

2. with an aspect advising only the `i`-loop with just `proceed()`, using a pointcut based on the data type: this is the aspect `TestArrayProceed`, shown in Listing 5.13; and

3. with an aspect advising only the `i`-loop with just `proceed()`, using a pointcut based on the control-flow: this is the aspect `TestCFlowProceed`, shown in Listing 5.14.

Although these three variants produce the same result and could be considered as equivalent by the programmer, the potential runtime overheads differ.

---

[7]`http://www.eclipse.org/aspectj/`
[8]`http://www.aspectbench.org/`

Listing 5.12: Simple example with three nested loops.

```java
public class Test {
    public int repeat ;
    public Test(int sizeDoubleArray, int sizeIntArray, int repeat) {
        /* ... */
    }

    private final double[] doubleArray ;
    private final int[] intArray ;
    public void initialiseArrays() {
        /* ... */
    }

    public void run() {
        int R = repeat ;
        double[] array1 = doubleArray ;
        int[] array2 = intArray ;

        for(int i = 0; i<array1.length; i++)
            for (int j = 0 ; j<R ; j++)
                for (int k = 0 ; k<array2.length ; k++)
                    array1[i] *= array2[k] - 10 ;
    }

    public static void main(String[] args) {
        /* ... */
        Test t = new Test(1000,100,repeat) ;
        t.initialiseArrays() ;

        long starttime = System.currentTimeMillis() ;

        t.run() ;

        long stoptime = System.currentTimeMillis() ;
        long duration = stoptime - starttime ;
        /* ... */
    }
}
```

Listing 5.13: Array-based aspect for the simple example.

```java
public aspect TestArrayProceed {
    pointcut withinrun(): withincode(void Test.run(..)) ;
    pointcut doublearrayloop(): loop() && args(*,*,*,double[]) ;

    void around(): doublearrayloop() && withinrun() {
        proceed() ;
    }
}
```

Listing 5.14: `cflow`-based aspect for the simple example.

```
public aspect TestCFlowProceed {
    pointcut withinrun(): withincode(void Test.run(..)) ;
    pointcut outerloop(): loop() && !cflowbelow(loop()) ;

    void around(): outerloop() && withinrun() {
       proceed() ;
    }
}
```

The advice in `TestArrayProceed` matches the two array-based loops, that is to say, the `i`- and `k`-loops, at compile time. The runtime test to check whether-or-not the pointcut matches each of these loops is based on an `instanceof` test on the array given in the loop, to check the runtime type of `array1` and `array2`, just before the `i`-loop and the `k`-loop, respectively. This implies $(R + 1) \times$ `array1.length` `instanceof` checks per execution of `run()`. Theoretically, the JVM could optimise these tests, since the type is known, even at compile-time.

The advice in `TestCFlowProceed` matches the three loops at compile time, and the `cflowbelow` construct entails the introduction in the woven code of a counter which is incremented on every loop entry, and decremented on every loop exit. When the value of this counter is `0`, the condition "`!cflowbelow(loop())`" is fullfilled and the advice is executed. This implies that this counter is incremented $(R+1) \times$ `array1.length` $+ 1$ times, and decremented as many times, per execution of `run()`.

In both cases, the loops that are matched at compile-time are extracted from the original method and placed into another method in the same class (see [Kuz04, ACH$^+$04b, ACH$^+$05a]).

Intuitively, it is expected that the version without any aspect will perform best, and that the data-based version will perform better than the `cflow`-based version. Tests on the Sparc machine, using the Sun JVMs, and on the Athlon machine, using the IBM JVM 1.4.2, produce the expected results, as shown in Figures 5.1 and 5.2.

However, the results obtained on the Athlon machine using the Sun JVMs are more surprising, as shown in Figures 5.3 and 5.4. For this particular application, the use of a counter (in the `cflow`-based version) appears to be better optimised than the use of `instanceof` (in the data-based version) using the client mode of the JVM. This might be surprising, but this is, after all, a matter of optimisation strategy (optimising the use of a counter or optimising the type-checking). What
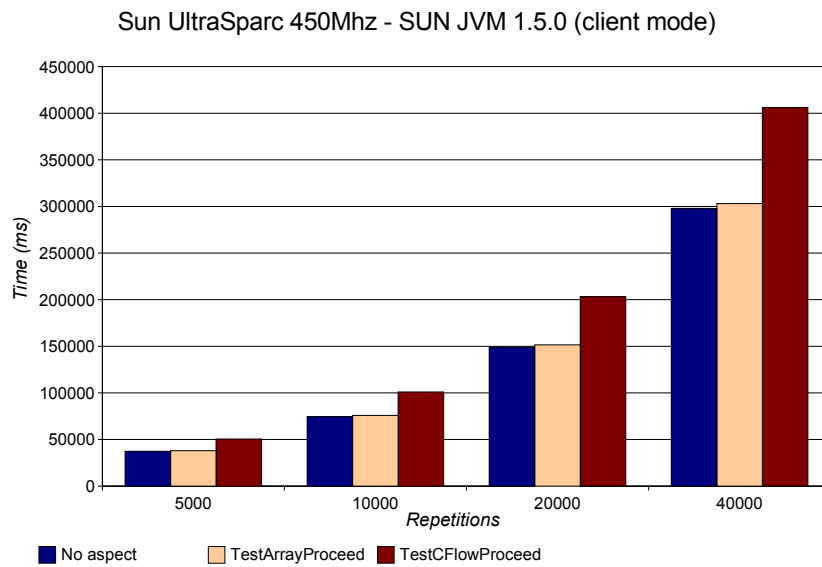
Figure 5.1: Performance comparison between data-based and `cflow`-based point-cuts on Sun JVM 1.5.0 (client)/Sparc.
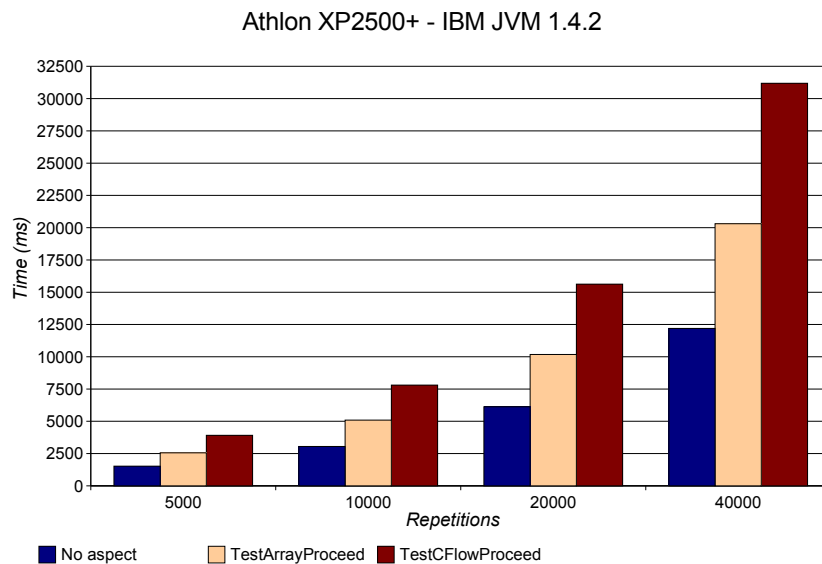
Figure 5.2: Performance comparison between data-based and `cflow`-based point-cuts on IBM JVM 1.4.2/Athlon.

is much more surprising, in both client and server modes, is that the version without an aspect —and therefore without extra code and indirections— gives worse performance results than one of the versions with an aspect —with indirections and additional runtime checks.[9]

In addition, the IBM JVM is always faster than the Sun JVM (up to about 3 times faster) in these tests.

Athlon XP2500+ - SUN JVM 1.5.0 (server mode)
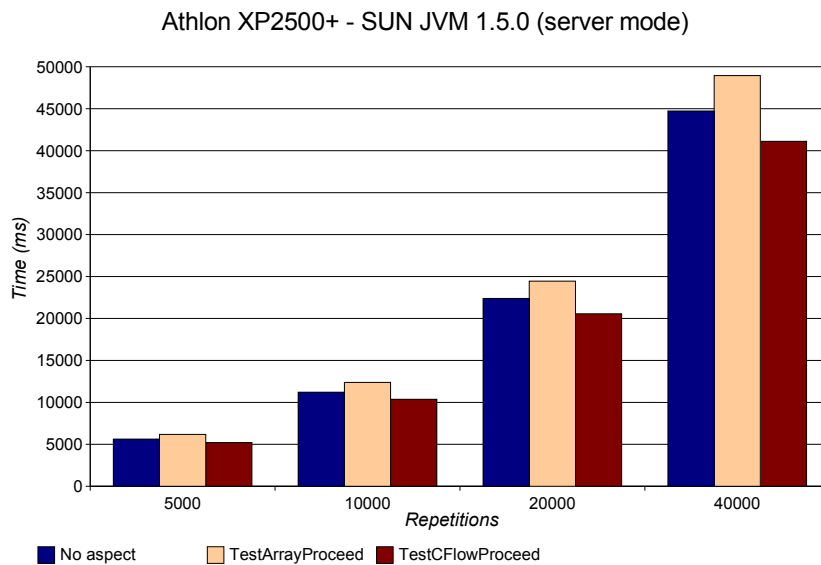
Figure 5.3: Performance comparison between data-based and `cflow`-based pointcuts on Sun JVM 1.5.0 (server)/Athlon.

---

[9]Further informal experiments have shown that the same Sun JVM does not present this surprising result on a Pentium IV. This is probably due to a bug in the Sun just-in-time compiler for Athlon. Although the AMD Athlon and the Intel Pentium are very similar, the just-in-time compiler of the Sun JVM is capable of detecting on which of these two categories of processors it is running, and subsequently triggers different optimisations.

Athlon XP2500+ - SUN JVM 1.5.0 (client mode)
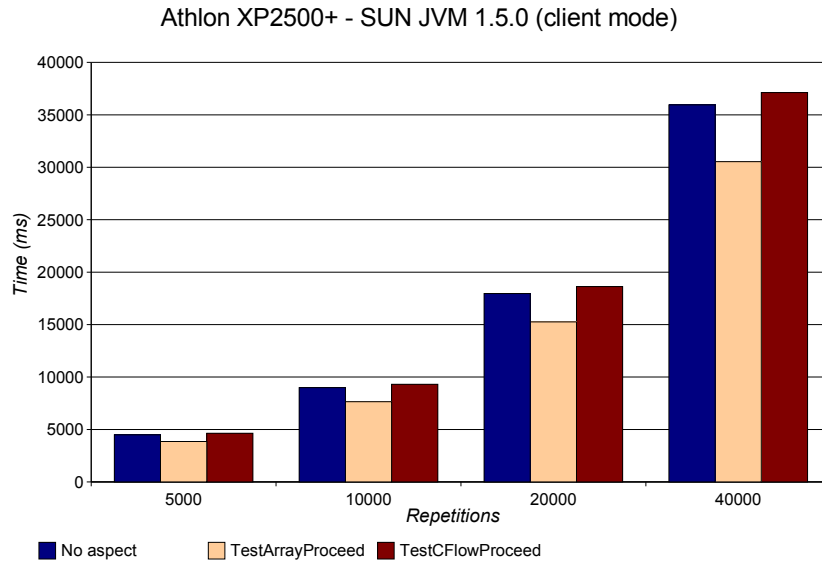
Figure 5.4: Performance comparison between data-based and `cflow`-based point-cuts on Sun JVM 1.5.0 (client)/Athlon.

5.6 Test-case: successive over-relaxation

This test-case is an implementation of the "Red-Black algorithm", which is an algorithm for Successive Over-Relaxation[10] (for solving partial differential equations). The numerical purpose of this test-case is to solve Laplace's equation $\nabla^2 u = 0$ on a square surface, with the following boundary conditions:

$$\begin{cases} \forall y \in [0, \pi], & u(0, y) = 0 \\ \forall x \in [0, \pi], & u(x, 0) = 0 \\ \forall x \in [0, \pi], & u(x, \pi) = \sin(x) \\ \forall y \in [0, \pi], & u(\pi, y) = 0 \end{cases}$$

The problem is discretised so that the surface is mapped onto a $(N + 2) \times (N + 2)$ square array. The boundaries of the square ($u_{0,j}$, $u_{N+1,j}$, $u_{i,0}$ and $u_{i,N+1}$) are initialised at the beginning and represent the constraints of the physical model. The algorithm consists of updating each value $u_{i,j \in [1..N]^2}$ of this array according to

---

[10]Details about the mathematical theory of this algorithm can be found in [PTVF93, pp. 866–869].

the following formula:

$$u_{i,j}^{(n+1)} = (1 - \omega)u_{i,j}^{(n)} + \frac{1}{4}\left(u_{i-1,j}^{(n)} + u_{i+1,j}^{(n)} + u_{i,j-1}^{(n)} + u_{i,j+1}^{(n)}\right)$$

where $u_{i,j}^{(n)}$ is the value at the ij-the position after the n-th iteration over the whole array, and ω is a constant. With this formula, the computation of any $u_{i,j}$ depends only on its North, South, East and West neighbours, as shown in Figure 5.5. Thus, the array can be decomposed into two parts, red and black, so that updating
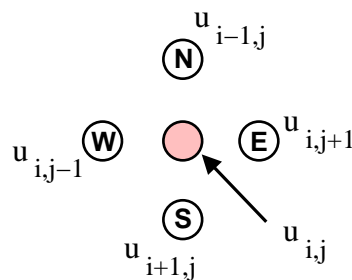
Figure 5.5: North, South, East and West in the Red-Black algorithm.

elements of one part depends only on the elements of the other part, as shown in Figure 5.6. Each red or black part of the array is embarrassingly parallelisable
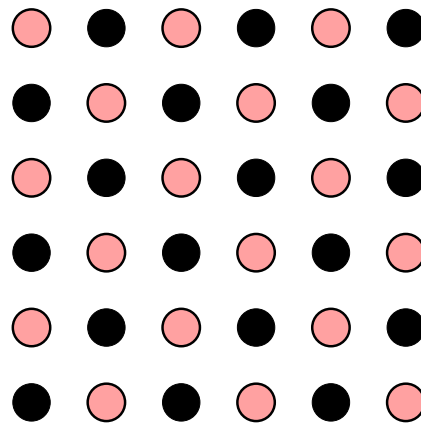
Figure 5.6: Red/Black decomposition.

(that is to say, each red (black, respectively) value can be computed independently from all the other red (black, respectively) values). The Red-Black algorithm is summarised by the following pseudo-code:

> **while** (degree of precision not achieved) {
>   compute all the red values ;
>   compute all the black values ;
> }.

Intuitively it is expected that the most efficient implementation of this algorithm in Java would be that shown in Listing 5.15. This implementation (`methodBasicA`) iterates consecutively through all the lines indexed by `i` and uses a stride of 2 in the `j`-loop to jump from one red (or black) spot to another. To ensure that the tests run long enough for the time measurements to be significant, the number of iterations is fixed and not based on any degree of precision.

Listing 5.15: Red/Black test-case: methodBasicA.

```java
public void methodBasicA (double u[][]) {
    int iterations = 0;
        while (iterations < maxIterations) {
            // Iterates through the red points.
            for (int i = 1; i <= N; i++)
                for (int j = (2 - (i % 2)); j <= N; j += 2)
                    u[i][j] = /* ... */

            // Iterates through the black points.
            for (int i = 1; i <= N; i++)
                for (int j = (1 + (i % 2)); j <= N; j += 2)
                    u[i][j] = /* ... */

            iterations++;
        }
}
```

5.6.1 AspectJ approach: object-oriented loops

The object-oriented loop model presented in Section 3.2 cannot handle stride values different from 1. Thus, `methodBasicA` has been modified to form `methodBasicB`, in which the `j`-loop uses a stride value of 1, as shown in Listing 5.16.

The test-case program also contains methods `methodRectangleLoopA`, `methodRectangleLoopB` and `methodRectangleLoopC`, which implement the two (red and black) sets of double-nested loops using models `RectangleLoopA`, `RectangleLoopB` and `RectangleLoopC`, respectively, from Section 3.2. These implementations are semantically equivalent to `methodBasicB`. The results obtained with

Listing 5.16: Red/Black test-case: methodBasicB.

```java
public void methodBasicB (double u[][]) {
    int iterations = 0;
    while (iterations < maxIterations) {
       // Iterates through the red points.
       for (int i = 1; i <= N; i++)
         for (int j = 1; j <= (N / 2); j++) {
           int jtemp = 2 * j - (i % 2);
           u[i][jtemp] = /* ... */
         }

       // Iterates through the black points.
       for (int i = 1; i <= N; i++)
         for (int j = 1; j <= (N / 2); j++) {
           int jtemp = 2 * j - 1 + (i % 2);
           u[i][jtemp] = /* ... */
         }

       iterations++;
    }
}
```

methodBasicA can be used to estimate the overhead due to re-factoring method-BasicA into methodBasicB and its subsequent equivalents.

Timers have been placed around each of these methods. The results presented in the remainder of this section have been normalised as follows:

$$\text{normalised time} = \frac{(\text{overall execution time}) \times (\text{computing threads})}{(\text{maxIterations}) \times (\text{array size})^2}.$$

Thus, each timing result represents the time needed for calculating one instance of $u_{i,j}$ plus the overhead due to the loops (whichever implementation is chosen).

5.6.1.1 Cost of refactoring

A first round of tests has been run without parallelism, in order to estimate the overhead introduced by turning regular for-loops into their object-oriented counterparts. These tests have been run for array sizes from 100 to 1000 (in steps of 100).

Figure 5.7 shows the results obtained with the IBM JVM 1.4.2 on the Athlon machine.[11] Figures 5.8 and 5.9 show the results obtained on the same machine with the Sun JVM 1.4.2 in client and server modes, respectively. Figures 5.10 and 5.11

---

[11]See Section 5.4.

show the results obtained on the same machine with the Sun JVM 1.5.0 in client
and server modes, respectively. Figure 5.12 shows the results obtained with the Sun
JVM 1.4.1 on the Sun machine. Figure 5.13 shows the results obtained with the
SGI JVM 1.4.1 on the SGI Origin. (The array of size 100×100 fits entirely in the
Athlon cache, and so does a large part of the array of size 200×200, which explains
better performance results for these sizes.)

Figure 5.7: Performance results for the Red-Black application, using object-
oriented loops, without parallelism, on IBM JVM 1.4.2/Athlon.

These graphs show that the overhead of using the `RectangleLoopX` models
depend on the JVM and on the architecture. The overhead of method calls due to
the loop object-model and to the introduction of an intermediate variable (from
`methodBasicA` to `methodBasicB`, see Listings 5.15 and 5.16) appears to be much
more important with the SGI JVM than with the others. Surprisingly, introducing
an extra variable and changing the stride to 1 improves the performance on the Sun
JVM 1.4.2 (in client mode); and, with the SGI JVM, `methodRectangleLoopC` gives
better results than `methodBasicB`.

On the Athlon machine, the relative overhead of refactoring the `for`-loops into
the object model is less visible with the Sun client JVMs and the IBM JVM than
with the Sun server JVMs. The fastest results are obtained with the IBM JVM,

Athlon XP2500+ - ajc 1.2.1 - SUN JVM 1.4.2 (client mode)

Figure 5.8: Performance results for the Red-Black application, using object-oriented loops, without parallelism, on Sun JVM 1.4.2 (client)/Athlon.

Athlon XP2500+ - ajc 1.2.1 - SUN JVM 1.4.2 (server mode)
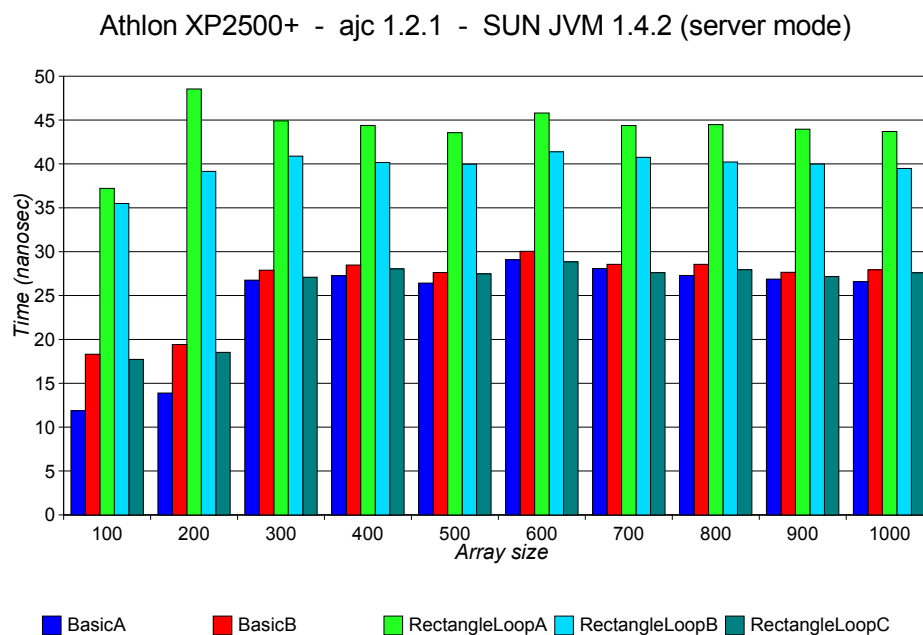
Figure 5.9: Performance results for the Red-Black application, using object-oriented loops, without parallelism, on Sun JVM 1.4.2 (server)/Athlon.

Figure 5.10: Performance results for the Red-Black application, using object-oriented loops, without parallelism, on Sun JVM 1.5.0 (client)/Athlon.

Figure 5.11: Performance results for the Red-Black application, using object-oriented loops, without parallelism, on Sun JVM 1.5.0 (server)/Athlon.

Sun Sparc (4 processors) - ajc 1.2.1 - SUN JVM 1.5.0 (64-bit server mode)

Figure 5.12: Performance results for the Red-Black application, using object-oriented loops, without parallelism, on Sun JVM 1.5.0 (server, 64-bit mode)/Sparc.
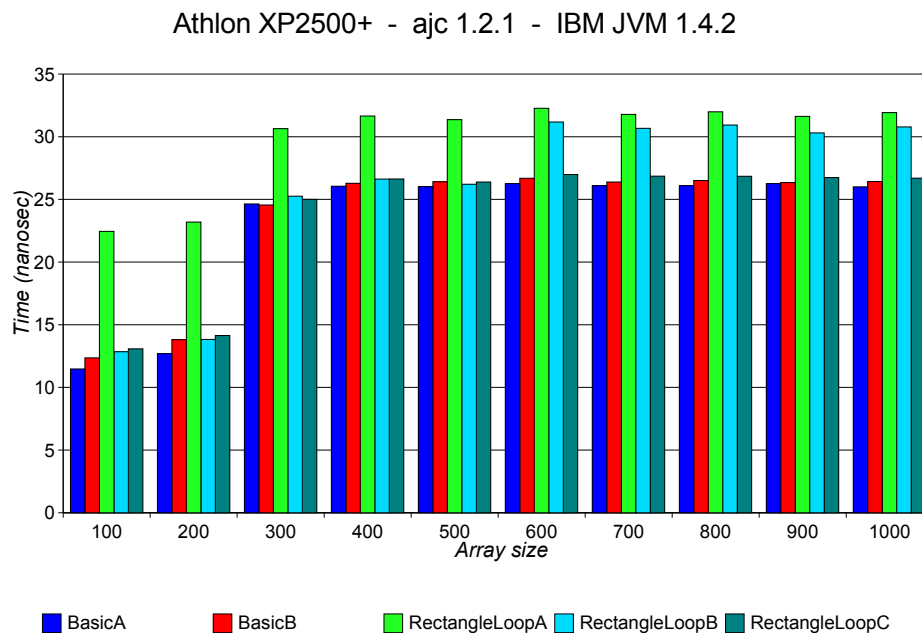
SGI Origin (16 processors) - ajc 1.2.1 - SGI 1.4.1

Figure 5.13: Performance results for the Red-Black application, using object-oriented loops, without parallelism, on SGI JVM 1.4.1/MIPS.

where the overhead is hardly noticeable on the graphs. The Sun server JVMs reach similar performance for the regular for-loops, but the cost of refactoring them into objects is much more important when using the Sun JVMs, even in server mode, than when using the IBM JVM.

In all cases, model `RectangleLoopA` (which is based on delegation) performs worse than `RectangleLoopB` (which relies on inheritance), which performs worse than `RectangleLoopC` (which is also based on inheritance, but maintains the inner loop inlined). However, the main difference between the Sun JVMs and the IBM JVM is that the IBM JVM better optimises the method calls in the inheritance-based model (`RectangleLoopB`). This can be explained by a more aggressive inlining strategy implemented in the IBM JVM [SOK+04]. The worst results, obtained with the SGI JVM, can probably be explained by the lack of interest and investment from SGI in Java-related technology.

5.6.1.2 Cost of parallelising

In a second round of tests, the version into which an aspect for parallelising the loops has been woven is run.

The aspect for parallelising the loops follows the structure shown in Listing C.2. In particular, the pointcuts utilised for making the selections of which loops are to be parallelised match the creation of the instances of `RectangleLoopX` (which can then be refined with `withincode` or `within`), as shown in Listing 5.17. When using models `RectangleLoopB` and `RectangleLoopC`, which use inheritance, the loop selection could also be based on the names of the classes extending `RectangleLoopB` or `RectangleLoopC`.

Figure 5.14 shows the results obtained using the Sun machine with four Sparc processors (using 1, 2, 3 and 4 threads). Figures 5.15 and 5.16 show the results obtained on the dual Pentium-III machine, using the Sun JVM 1.5.0 (server mode) and the IBM JVM 1.4.2, respectively (using 1 and 2 threads). The results show that `methodBasicX` and `methodRectangleLoopX` are not particularly affected by the parallelisation aspect. Remember that, for normalisation, the execution times on several threads (for method `methodMTRectangleLoopX`) have been multiplied by the number of threads. This has the effect of adding in the per-iteration overhead times due to parallel execution.

The time overheads for weaving the parallelisation aspect, even using a single thread, are not surprising since there is a cost for dealing with any pool of threads

Listing 5.17: Writing pointcuts for parallelising the object-oriented loop models.

```
pointcut rectAinstantiation():
    call (RectangleLoopA.new(..)) && !within(ParallelisationAspect);

RectangleLoopA around (Runnable2DLoopBody loopBody,
                       int minI, int maxI,
                       int minJ, int maxJ):
    rectAinstantiation() && args (loopBody, minI, maxI, minJ, maxJ) {
        return new MTRectangleLoopA (loopBody,minI,maxI,minJ,maxJ) ;
}

class MTRectangleLoopA extends RectangleLoopA {
    private final Runnable[] subLoops ;
    public MTRectangleLoopA (Runnable2DLoopBody loopBody,
        int minI, int maxI, int minJ, int maxJ) {
            super(loopBody, minI, maxI, minJ, maxJ) ;
            subLoops = new Runnable [NUM_PROC] ;
            int width = (int) Math.ceil(
                ((double) (maxI - minI)) / (double) NUM_PROC);
            for (int k=0; k<NUM_PROC; k++) {
                int min = minI + k*width ;
                int max = minI + (k + 1) * width - 1;
                if (max > maxI) max = maxI;
                subLoops[k] =
                    new RectangleLoopA(loopBody, min, max, minJ, maxJ);
            }
        }
    public void run () {
        threadPool.run(subLoops) ;
    }
}


pointcut rectBinstantiation():
    call(RectangleLoopB+.new(..)) && !within(ParallelisationAspect);
after() returning (RectangleLoopB temp):
    rectBinstantiation() {
        temp.subLoops = new Runnable [NUM_PROC] ;
        int width = (int) Math.ceil(
            ((double)(temp.maxI-temp.minI))/(double)NUM_PROC);
        for (int k=0; k<NUM_PROC; k++) {
            int min = temp.minI + k*width ;
            int max = temp.minI + (k + 1) * width - 1;
            if (max > temp.maxI) max = temp.maxI;
            temp.subLoops[k] = new MiniLoopB(temp, min, max) ;
        }
}

void around (RectangleLoopB loop):
    call(void *.run(..)) && target(loop)
        && !within(ParallelisationAspect) {
        threadPool.run (loop.subLoops) ;
}
```
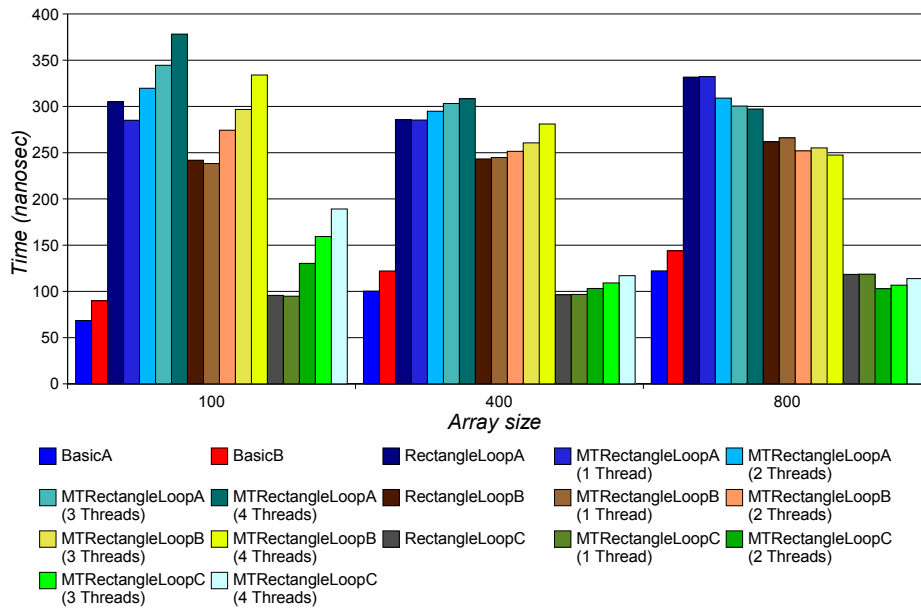
Figure 5.14: Performance results for the Red-Black application, using object-oriented loops, with an aspect for parallelisation, on Sun JVM 1.5.0/Sparc.

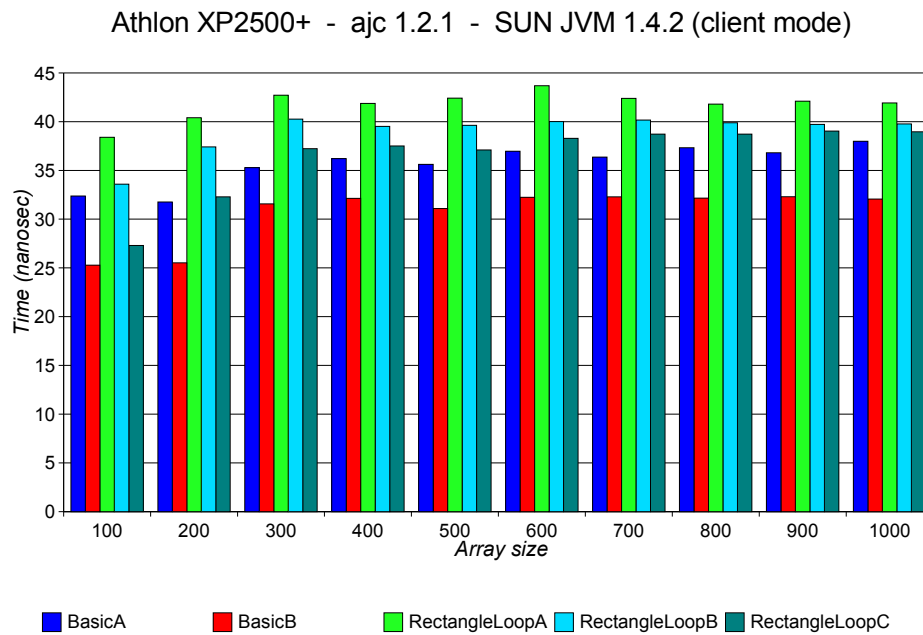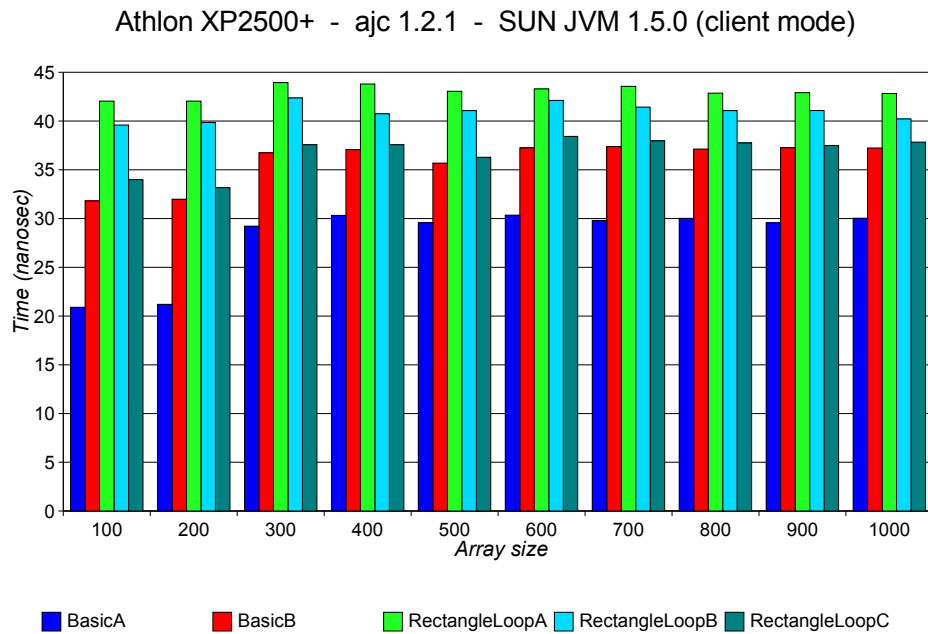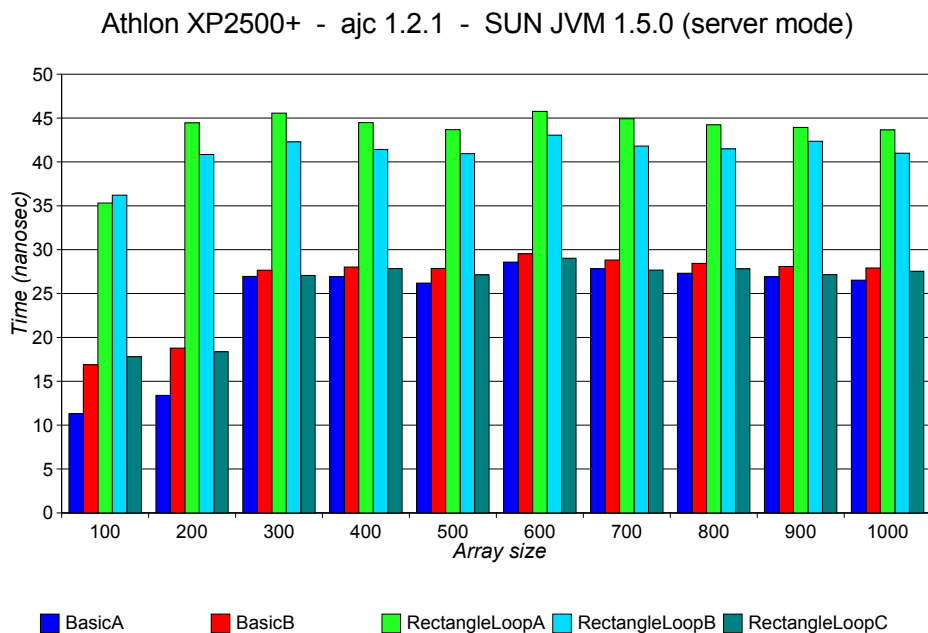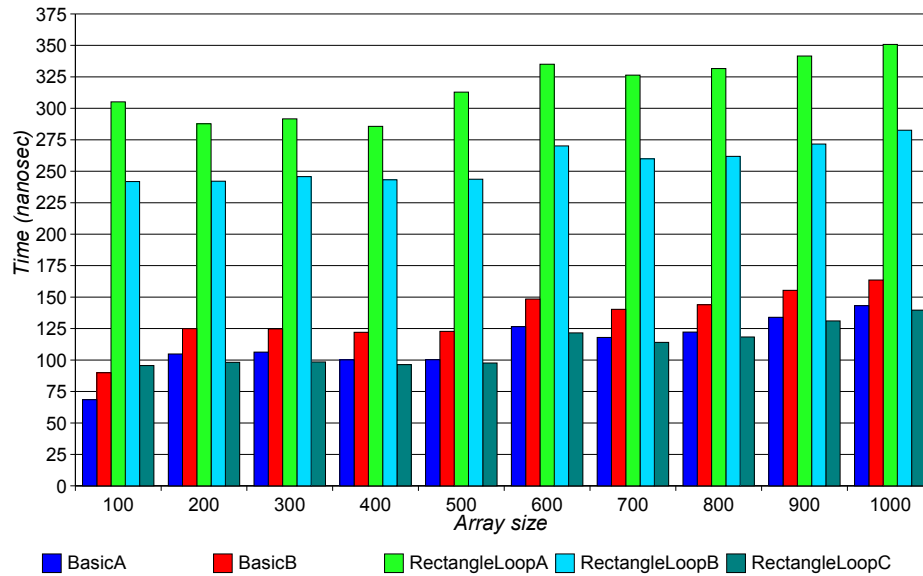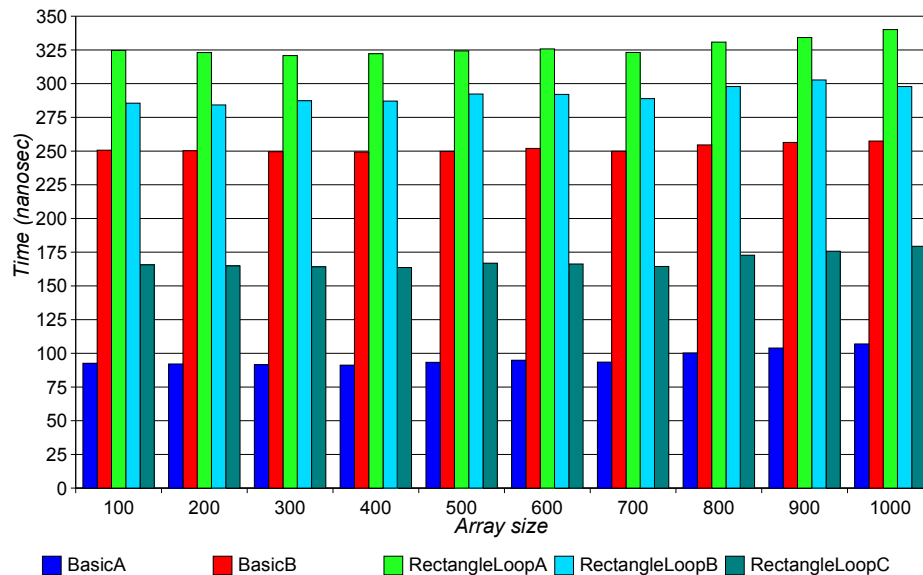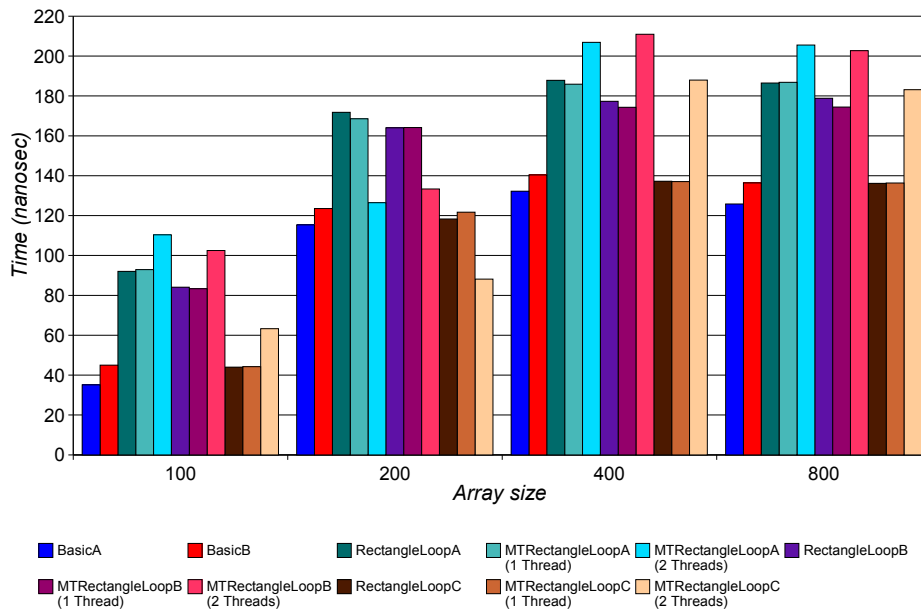Figure 5.15: Performance results for the Red-Black application, using object-oriented loops, with an aspect for parallelisation, on Sun JVM 1.5.0/Pentium-III.
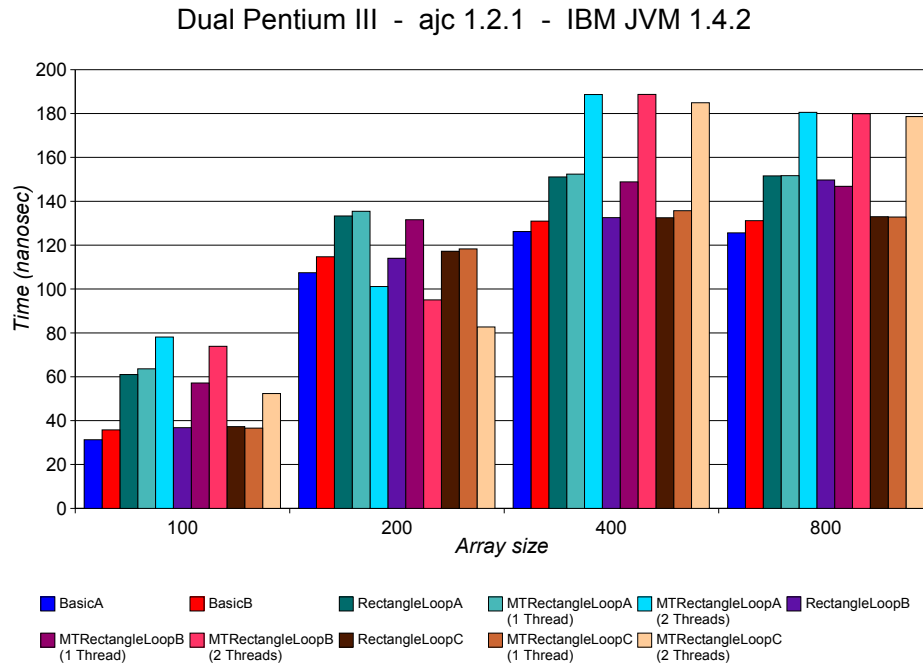
Figure 5.16: Performance results for the Red-Black application, using object-oriented loops, in parallel, on IBM JVM 1.4.2/Pentium-III.

(including a pool of one). On the 4-processor machine, the overhead for parallelising using 1 to 3 threads is consistent with the cost of synchronisation. More surprisingly, the cost of using four threads is much higher than the others. This is most likely explained by the last thread competing on the fourth and final processor with the internal threads of the JVM. Under Unix, JVMs create several processes —even for single-threaded applications— for handling system functions such as memory management and just-in-time compilation. These processes compete for processor time with the application threads.

5.6.2   LOOPSAJ approach

The Red-Black algorithm works on a square array of size $(N + 2) \times (N + 2)$. The iterations only modify the values $u_{i,j}$ for $(i, j) \in [1..N]^2$. Subsequently, the loop is not working on an array from *0* to *length*, so it is not possible to write a pointcut based on the array type. The pointcuts used for advising loops in this model are therefore `cflow`-based.

Note that the pointcut "`loop() && !cflowbelow(loop())`" selects the outermost loop only, while "`loop() && cflowbelow(loop())`" selects all the inner loops, regardless of their degree of nesting.

In the Red-Black algorithm, the loops that are to be parallelised are the second inner loops, labelled `iloop1` and `iloop2` in Listing 5.18. The labels on the loops in this listing are only included for reference from this text, and do not have any effect on the loop recognition or on the produced code.

In order to prevent the outermost loop (labelled `testconvergenceloop`) from being recognised to be of the form `for(int k=MIN; k<MAX; k+=STRIDE)`, which would bind arguments to the `args(..)` pointcut, `methodBasicA` has been slightly modified, and this outermost loop has been rewritten so that it decreases the `iterations` counter. This counter was placed here purely for the sake of the tests; otherwise, the loop would be of the form: `while(error(u,solution)>tolerance)`, and would not have been matched by a pointcut of the form "`loop() && args(min, max, stride, ..)`". A similar modification has been made to `methodBasicB`.

Listing 5.18: Red/Black test-case: methodBasicA.

```java
public void methodBasicA (double u[][]) {
    final int iterations = maxIterations;
    final int N1 = N+1 ;

    testconvergenceloop:
    while (iterations > 0 ) {
      // Iterates through the red points.
      iloop1:
      for (int i = 1; i < N1; i++)
        jloop1:
        for (int j = (2 - (i % 2)); j < N1; j += 2)
          u[i][j] = /* ... */

      // Iterates through the black points.
      iloop2:
      for (int i = 1; i < N1; i++)
        jloop2:
        for (int j = (1 + (i % 2)); j < N1; j += 2)
          u[i][j] = /* ... */

      iterations--;
    }
}
```

The pointcuts shown in Listing 5.19 select the loops to be parallelised in `method-BasicA` and `methodBasicB`. Pointcut `loopmaybeunderloopwithnoargs` matches the loops that are not within a loop that would itself have arguments, but that may be within a loop without arguments. This restriction is represented by the following term in `loopmaybeunderloopwithnoargs`: "`!cflowbelow(loop() && args(*,*,*,..))`".

Listing 5.19: Pointcuts for parallelising loops in the Red-Black algorithm (basic methods A and B).

```
pointcut methodsAB():
    withincode(* ExampleRedBlack.methodBasicA(..)) ||
    withincode(* ExampleRedBlack.methodBasicB(..)) ;

pointcut loopmaybeunderloopwithnoargs():
    loop() && !cflowbelow(loop() && args(*,*,*,..)) ;

pointcut loopsAB():
    loopmaybeunderloopwithnoargs() && methodsAB();
```

Refactoring can also be a means of selecting the second innermost loop, by extracting a method for the content of the outermost loop, as shown in Listing 5.20. The corresponding pointcuts are written as shown in Listing 5.21.

`methodBasicA`, `methodBasicB` and `methodBasicC` have been compiled without any aspect (used as a reference), with an aspect executing only `proceed()` (to show the cost of weaving), and with an aspect executing the loop in a thread pool (to show the cost of parallelising).

Listing 5.20: Red/Black test-case: methodBasicC.

```
public void methodBasicC (final double u[][]) {
    int maxIterations = /* ... */
    int iterations = 0;
    final int N1 = N+1 ;

    while (iterations < maxIterations) {
        // Iterates over the whole array.
        innerMethodBasicC(u, N1);

        iterations++;
    }
}

private void innerMethodBasicC(final double[][] u, final int N1) {
    // Iterates through the red points.
    for (int i = 1; i < N1; i++)
      for (int j = (2 - (i % 2)); j < N1; j += 2)
        u[i][j] = /* ... */

    // Iterates through the black points.
    for (int i = 1; i < N1; i++)
      for (int j = (1 + (i % 2)); j < N1; j += 2)
        u[i][j] = /* ... */
  }
```

Listing 5.21: Pointcuts for parallelising loops in the Red-Black algorithm (method-BasicC).

```
pointcut methodsC(): withincode(* ExampleRedBlack.methodBasicC(..)) ;

pointcut innerMethodsC():
    withincode(* ExampleRedBlack.innerMethodBasicC(..)) ;

pointcut loopsC():
     loop() && !cflowbelow(loop() && innerMethodsC())
    && innerMethodsC() ;
```

5.6.2.1 Cost of weaving

The results of execution on the Athlon machine are presented in Figures 5.17 and 5.18, using the IBM JVM 1.4.2 and the Sun JVM 1.5.0 (server mode), respectively.
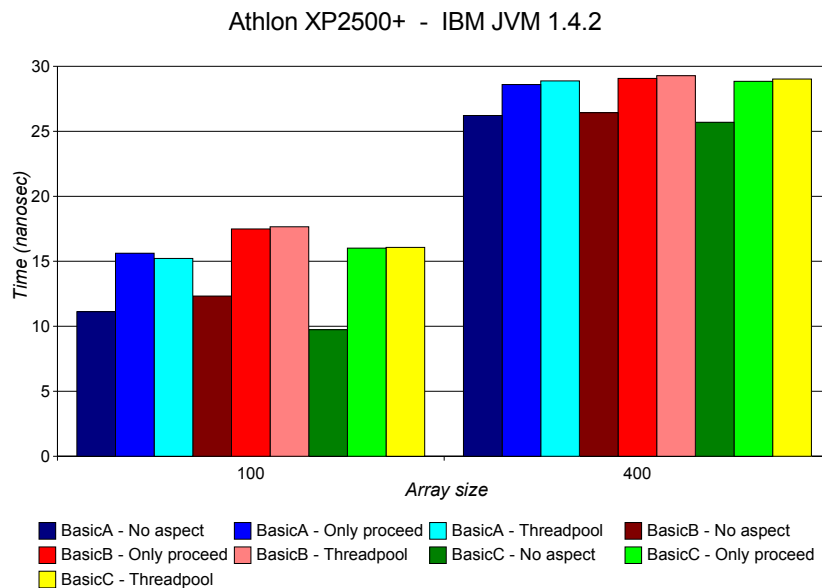
Figure 5.17: Performance results for the Red-Black application, using LOOPSAJ, without parallelism, on IBM JVM 1.4.2/Athlon.

The penalty for weaving aspects (with just `proceed()`, and even with a pool of 1 thread) appears to be similar with both JVM configurations, and, in both cases, it is more visible on the smaller array sizes. The overhead ranges from about 10% (for `methodBasicA` with array size 400, on the IBM JVM) to 70% (for `methodBasicA` with array size 100, on the Sun JVM).
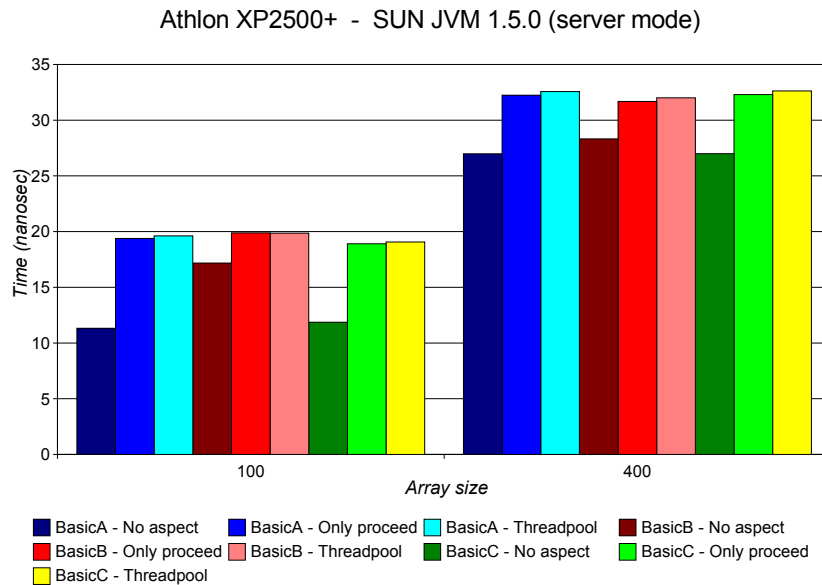
Figure 5.18: Performance results for the Red-Black application, using LoopsAJ, without parallelism, on Sun JVM 1.5.0 (server)/Athlon.

5.6.2.2 Cost of parallelising

Figures 5.19 and 5.20 presents results obtained with parallelism.

Although the overhead of "parallelising on 1 thread" is not really visible, the cost is not negligible when several threads are used. In particular, the smaller the array (and, therefore, the shorter the overall computation), the less negligible this overhead is. This is particularly visible for size 100 on the Sun Sparc machine: there is an overhead of about 700% when running on 4 threads (which means that it takes twice as long to get the result on 4 processors as it takes on a single processor). This bad result can be explained by the creation of `Runnables` for each iteration. Indeed, this cost seems to decrease when the array size increases. The best performance gain obtained in this set of results is with the IBM JVM on the dual Pentium-III machine for size 200: when the array is split into two blocks, each block fits into the cache of each processor, which leads to a negative overhead. (There is a similar effect when using the object-oriented loops—see Figure 5.16.)

Dual Pentium III - methodBasicA - IBM JVM 1.4.2
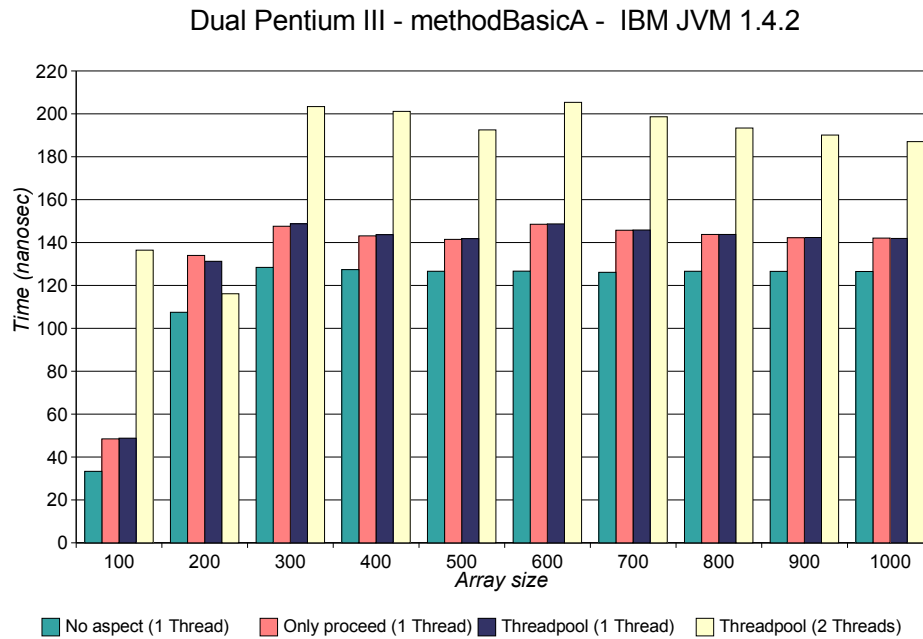
Figure 5.19: Performance results for the Red-Black application, using LoopsAJ, with an aspect for parallelisation, on IBM JVM 1.4.2/Pentium-III.

Sun Sparc (4 processors) - methodBasicA - SUN JVM 1.5.0 (client mode)
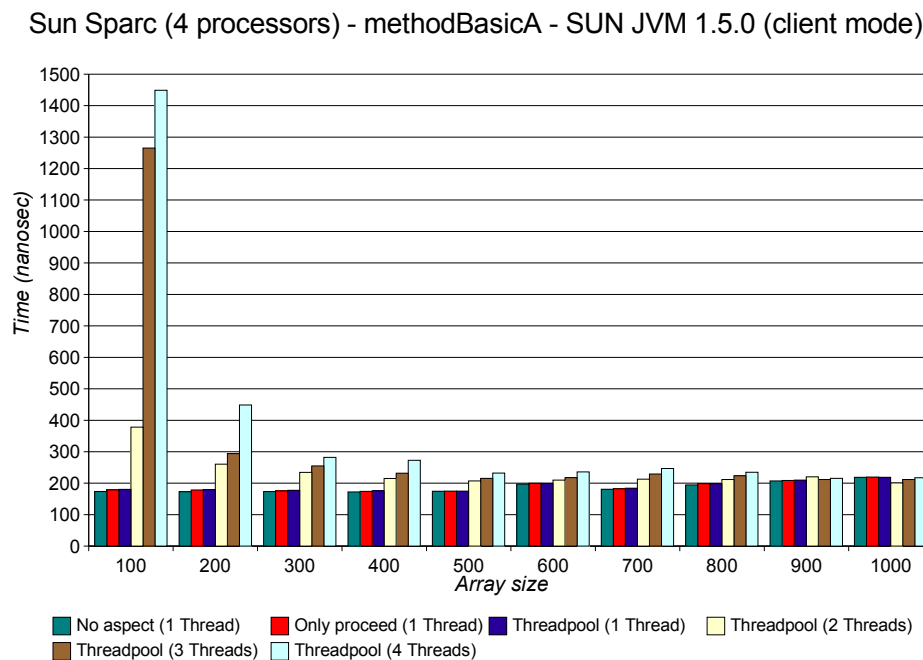
Figure 5.20: Performance results for the Red-Black application, using LoopsAJ, with an aspect for parallelisation, on SUN JVM 1.5.0 (client mode)/Sparc.

5.6.3   Performance comparison

The general trend for the performance of the object-oriented loops is ordered as follows, from worst to best: model `RectangleLoopA`, model `RectangleLoopB` and model `RectangleLoopC`.

Figure 5.21 shows a comparison between the versions compiled without aspects (both with `ajc` and `abc`), the refactored versions with the three `RectangleLoopX` models, and the version where LOOPSAJ is used to weave a `proceed()` advice around the loop to be parallelised. In terms of cost of refactoring and cost of weaving directly into the loop, the LOOPSAJ version always performs better than models `RectangleLoopA` and `RectangleLoopB`, and nearly as well as the best refactored model, `RectangleLoopC`.

Figures 5.22, 5.23 and 5.24 show a comparison of the same versions with parallelisation aspects, on three different configurations of machine and JVM. The "sequential" and the "1-thread" versions differ in that the sequential results show the cost of just weaving and refactoring, as above, whereas the 1-thread results show the cost of refactoring and weaving a parallelisation advice configured to use only 1 thread.

In parallel, the performance of the version using the LOOPSAJ aspects appears to be in-between the performance of the versions using models `RectangleLoopB` and `RectangleLoopC`, as for the sequential version. However, the overhead due to parallelisation with the LOOPSAJ aspect increases with the number of threads used. As shown in Section 5.6.2.2, in the LOOPSAJ model, an instance of `Runnable` is created for each iteration of the parallelised loop. This incurs a cost which is more visible on small tasks, i.e. on smaller array sizes, or on smaller portions of arrays (that is, when the work for the whole array is divided onto more threads). Moreover, this overhead makes the parallelisation on all the processors with the LOOPSAJ model worse than using the refactored object-oriented loops, on this kind of test-case. As mentioned in Section 5.6.1.2, when the number of threads used for the computation is the same as (or greater than) the number of processors available, these threads have to share resources with other threads used internally in the JVM, such as the just-in-time compiler and the garbage collector. When using the LOOPSAJ aspect, creating instances of `Runnable` for each iteration not only incurs an overhead per-se, but also increases the load on the garbage collector, which has to dispose of these new objects after each iteration as well. This is the most likely cause of the much larger overhead that occurs when using all the processors. In this

case, the results obtained using the LOOPSAJ aspect are substantially degraded, and even worse than the results obtained with the worst-performing object-oriented loop (`RectangleLoopA`) on the dual Pentium-III.
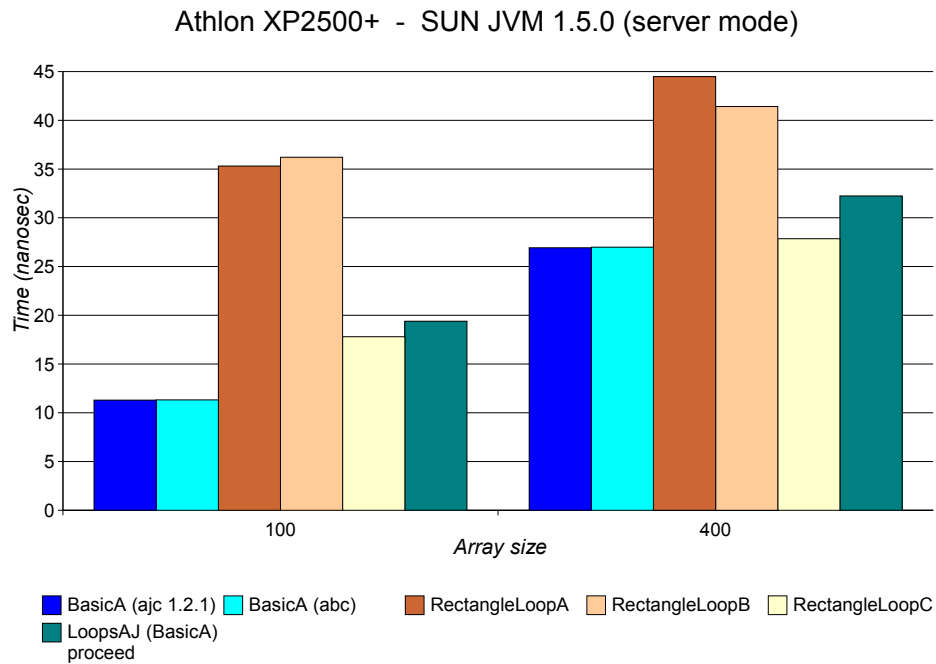
Figure 5.21: Comparison of object-oriented loops and loop join point (Red-Black algorithm), single Athlon, Sun JVM 1.5.0 (server mode).
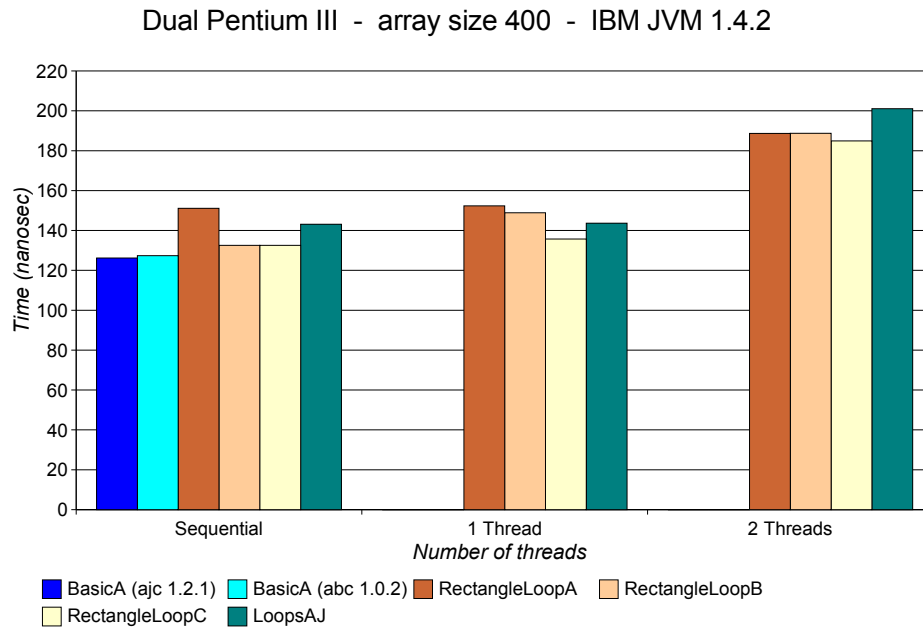
Figure 5.22: Comparison of object-oriented loops and loop join point (Red-Black algorithm), dual Pentium-III, using the IBM JVM 1.4.2.
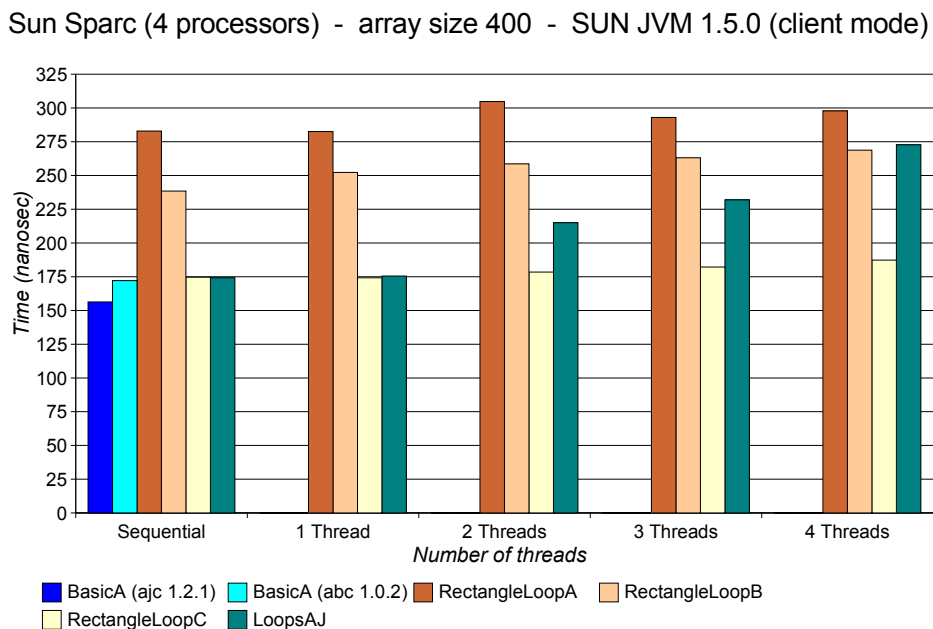
Figure 5.23: Comparison of object-oriented loops and loop join point (Red-Black algorithm), 4-processor Sun Sparc, using the Sun JVM 1.5.0 (client).

Sun Sparc (4 processors) - size 400 - SUN JVM 1.5.0 (64-bit server mode)
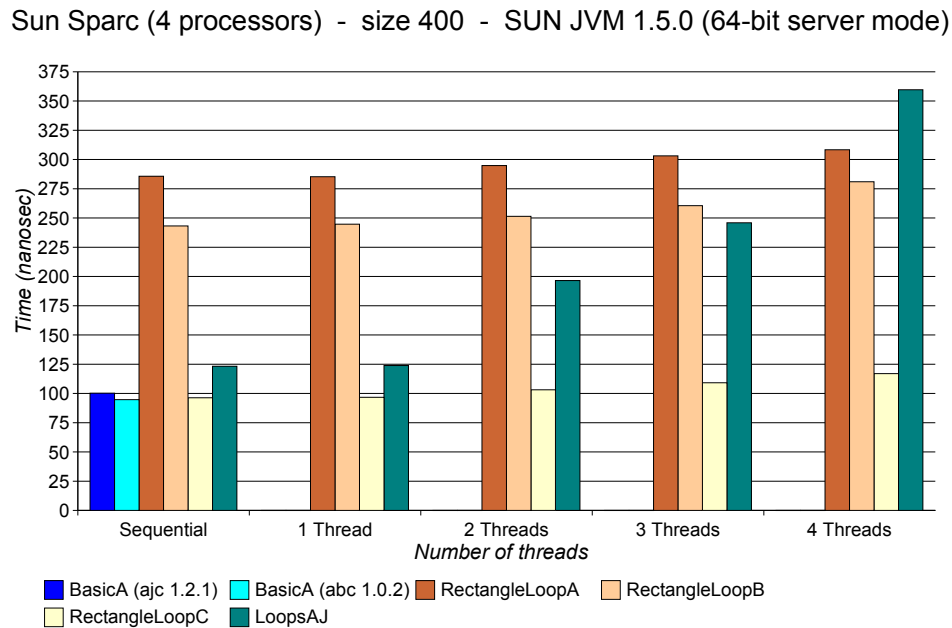
Figure 5.24: Comparison of object-oriented loops and loop join point (Red-Black algorithm), 4-processor Sun Sparc, using the Sun JVM 1.5.0 (64-bit server).

5.7 Test-case: the Crypt application

The crypt application, part of the Java Grande Forum benchmark suite, is described in Section 3.1.1. The main computation is located in method `cipher_idea`, which is called twice by method Do, as shown in Listing 5.22.

5.7.1 AspectJ approach: minor refactoring

As described in Section 3.1.1, `cipher_idea` can be parallelised after a small modification of the code. This small modification only adds a couple of method call indirections, and its effect on performance falls far below the precision of the timers, and is thus not easily measurable. The aspect for Java-thread parallelisation, as presented in Listing 3.4, is woven into this application for the tests presented in Section 5.7.3.

5.7.2 LOOPSAJ approach

Method `cipher_idea` contains a nest of two loops. Both are recognised by the `loop()` pointcut. However, only the outer loop is iterating over an array. As

Listing 5.22: Implementation of `cipher_idea` and Do.

```java
private void cipher_idea(byte[] text1, byte[] text2, int[] key) {
    /* Declaration of local variables and initialisations */
    ...

    for (int i = 0; i < text1.length; i += 8) {
        int r = 8 ;

        /* ... */

        do {
            /*  ... */
        } while (r-- != 0) ;

        /* ... */
    }
}

void Do() {
    // Start the stopwatch.
    JGFInstrumentor.startTimer("Section2:Crypt:Kernel");

    cipher_idea(plain1, crypt1, Z);     // Encrypt plain1.
    cipher_idea(crypt1, plain2, DK);    // Decrypt.

    // Stop the stopwatch.
    JGFInstrumentor.stopTimer("Section2:Crypt:Kernel");
}
```

described in Section 5.3, there are two ways of writing a pointcut that will match the outer loop:

- a data-based pointcut, which will match the array of type `byte[]`;

- a `cflow`-based pointcut, which will internally use a counter.

5.7.3 Performance comparison

This section presents a comparison between five approaches:

- the reference sequential version, which is the original sequential version from the Java Grande Forum benchmarks;

- two versions, using a data-based and a `cflow`-based aspect (as described in Section 5.7.2), that only proceed with the original execution (for measuring the cost of weaving);

- the reference multithreaded version, which is the original Java threads version from the Java Grande Forum benchmarks;

- the refactored version, as described in Section 5.7.1, into which aspect `MultiThreadCrypt` (see Listing 3.4) has been woven;

- two versions with an aspect for parallelisation, using a data-based and a `cflow`-based aspect, respectively.

Figures 5.25 and 5.26 present speed results[12] for these five approaches on a Sparc 4-processor machine, using the Sun JVM 1.4.2 in client mode and the Sun JVM 1.5.0 in 64-bit server mode, respectively. The difference between all the versions is mostly less than 2%. When parallelising, the smaller array size (SizeA) appears to incur a bigger overhead than the larger sizes, even with the original multithreaded version from the Java Grande Forum benchmarks. The version parallelised on 1 thread with the LOOPSAJ data-based version performs slightly better than the refactored version that uses an AspectJ-only aspect, and performs as well as the sequential version without aspects.

In this kind of example, where the parallelised loop is executed only a few times, choosing between the tangled multithreaded version, the version refactored for use

---

[12]Note that all the results presented in Section 5.6 are *time* results, while all the results presented in Section 5.7 are *speed* results.

Sun Sparc (4 processors) - SUN JVM 1.4.2 (client mode)
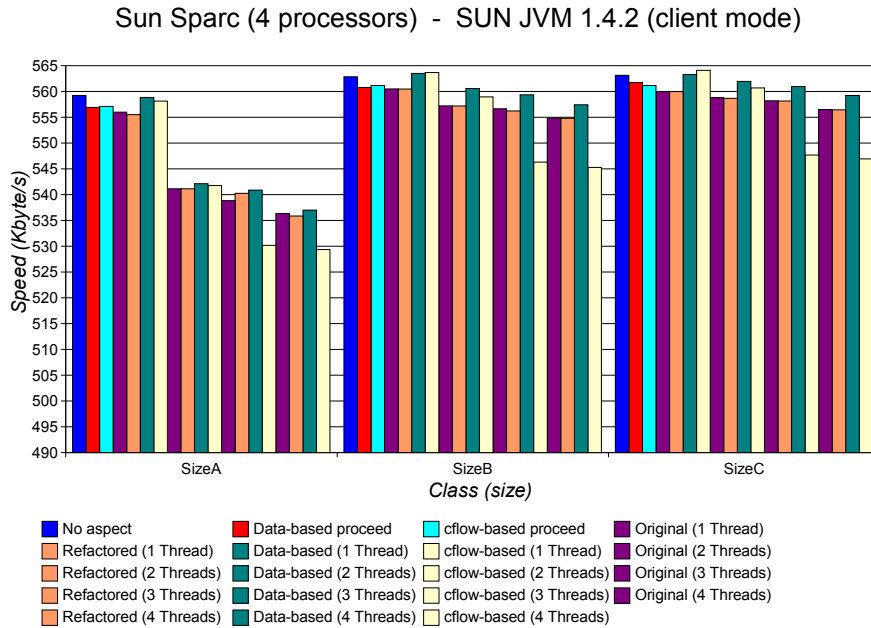
Figure 5.25: Comparison between the original, the refactored and the loop join point version of Crypt, in parallel on a 4-processor Sun Sparc, using the Sun JVM 1.5.0 (client). It should be noted that the origin on this graph is not zero and that, unlike the results presented in Section 5.6, these are *speed* results.

Sun Sparc (4 processors) - SUN JVM 1.5.0 (64-bit server mode)
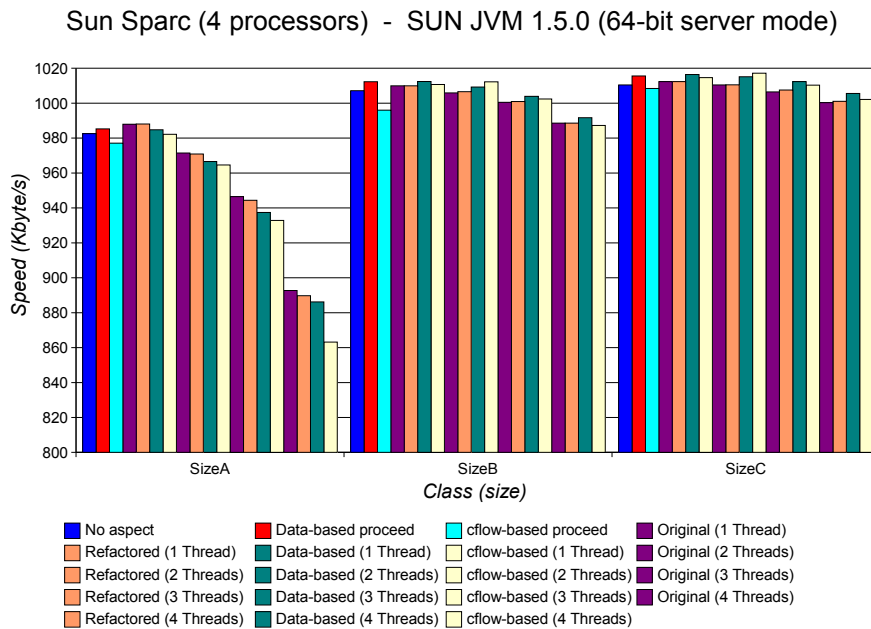
Figure 5.26: Comparison between the original, the refactored and the loop join point version of Crypt, in parallel on a 4-processor Sun Sparc, using the Sun JVM 1.5.0 (64-bit server). It should be noted that the origin on this graph is not zero.

with AspectJ, or the version advising the loop directly with LOOPSAJ is not really significant in terms of performance, except with the `cflow`-based pointcut with LOOPSAJ where the parallelised loop contains other loops. The choice between these versions is mostly a matter of design and readability of the application and of the units from which it is constituted.

5.8   Summary

This chapter has investigated two facets of the use of aspects for numerical computing: their impact on flexibility and their impact on performance.

As demonstrated throughout this chapter, and more particularly in Section 5.1, aspects can provide scientific applications with a flexible design. This flexibility is gained from two perspectives:

- the decoupling of the numerical concern from the implementation of the parallelisation; and

- the re-usability of the abstract parallelisation aspects, which could be incorporated into many applications.

However, this flexibility comes at a cost which depends heavily on the JVM used (because the examples are based on Java) and on the selection mechanism used. The performance of JVMs is a general problem addressed by the major vendors and virtual-machine researchers. It is not specific to aspects for high-performance computing, nor even to AOP.

The implementation of the selection mechanism is an area for which specific optimisations could be achieved, but often seem not to be. In particular, all the values of the counters used in the `cflow`-based LOOPSAJ examples, and the corresponding runtime tests, could be computed at compile-time. Indeed, the loop join point cannot be a regular entry point at the interface of a Java class. Therefore, wherever the counter is incremented or decremented could be predicted theoretically in circumstances where this mechanism is used as a means of selecting a particular degree of nesting. Such optimisation would be related to the `pcflow` pointcut envisaged by Gregor Kiczales in his keynote speech at the AOSD conference in 2003 [Kic03]. The `pcflow` pointcut would "predict" the control-flow at compile-time, based on heuristics, and perform what `cflow` currently does at runtime.

Another major disadvantage of the `cflow`-based selection is that it only permits the selection of either the outer loop or all the inner loops. No particular degree of nesting can be pin-pointed. A possible way to solve this problem would be to extend AspectJ with a new pointcut `cflowlevel(int, `*`Pointcut`*`)`. This pointcut could be implemented in a manner similar to the original `cflow` counter implementation; the runtime test would check whether the value of the counter is that given as the first parameter to `cflowlevel`, instead of simply comparing the counter value against zero.

In terms of performance, the selection mechanism would probably benefit from the techniques employed in SCoPE[13] [AM05], in which certain runtime tests in the pointcut expressions are eliminated when their outcome can be predicted at compile-time.

For parallelisation, the Java Thread model, which requires the creation of instances of `Runnable`, makes it less advantageous for small tasks to use the loop join point approach than the refactored approach, since it may be possible to adapt refactoring so that fewer instances of `Runnable` are created. However, when more substantial tasks are used, the loop join point approach gives performance results that can compete with the best refactored versions for aspects, and even with tangled parallel implementations.

---

[13]Static Conditional Pointcut Evaluator for AspectJ.

Chapter 6

Conclusions

6.1 Contributions

The work presented in this thesis contributes to improving the programming models in scientific computing in order to decouple the various concerns that must be dealt with in scientific software.

The two primary concerns of scientific computing are the numerical models (that is, what the application is supposed to do) and the speed of their executions (because, coincidentally, numerical models tend to require a large amount of computing power). Using programming languages such as Fortran, C++ or Java, Section 1.3 has identified that code for these two concerns become tangled one with the other. This thesis presents a novel step towards resolving the tensions between clarity of the description of numerical models and details of the (mostly parallel) implementation for high performance.

As mentioned in Chapter 2, one of the first examples in the literature of AOP involved encapsulating the optimisation concern separately from the numerical concern, in a graphic application programmed in Lisp. However, further developments in AOP were driven away from the domain of fine-grained optimisation and from Lisp, to embrace languages that have become more appealing for a wider audience in the software industry.

The relative success of AOP, as a programming model, is largely due to the success of AspectJ. Because AspectJ is Java-based, its success is also closely linked to the current popularity of Java.

Although the use of Java in the numerical computing community is still uncommon and perhaps controversial, it is growing, almost certainly due to substantial

improvements in the performance of the underlying Java Virtual Machines. Therefore, AspectJ has proved to be a suitable platform to experiment with AOP for scientific computing, and has provided the work reported in this thesis with both theoretical and practical results.

Chapter 3 provides mechanisms for using AspectJ in order to encapsulate the parallelisation concern in an aspect, that is, separate from the numerical code. This is achieved with varying degrees of success. The major problem encountered lies in the underlying abstraction that is utilised to describe the numerical problem, and the ability that aspects have to define and act upon recognition of this abstraction.

The prime mechanism upon which AOP is based is the concept of a join point. The challenge for this thesis has been to formulate the concept of join point as an abstraction that can make the link between the numerical model and the implementation of its performance. As a result, the model for a join point for loops is proposed, in Chapter 4.

6.1.1 Contributions to scientific computing

The major contribution of this thesis has been to demonstrate the possibility of using aspects for decoupling completely the implementation of the parallelisation from the implementation of the numerical models. As shown throughout Chapters 3, 4 and 5 (more particularly in Section 5.1), the example applications of the techniques presented in this thesis make it possible to use the same base numerical components with different parallelisation aspects, and the same aspects with different numerical components. In such systems, the numerical code that forms the base components is totally decoupled from the code implementing the parallelisation. The benefits in terms of reusability are immediate. The readability is also improved, compared with implementations that are not aspect-oriented, since the numerical concern and the parallelisation concern are not at all tangled. Reasoning with aspects might incur a certain learning curve, but this can be eased by the use of tools that assist the development process.

6.1.2 Contributions to aspect-oriented programming

As far as the AOP community is concerned, the thesis demonstrates the need for, and the possibility of making, aspects capable of handling complex behaviours,

which are not necessarily limited to acting upon actions occurring at the (named) interface of the objects or traditional modules.

The join point for loops presented in Chapter 4 is a join point that could not be targetted by any combination of AspectJ pointcuts. Although loops are usually part of the basic programming toolkit, providing mechanisms for AspectJ-like languages to be able to handle loops requires in-depth analysis of the base program. Providing AspectJ with the ability to intervene at loop-level cannot be solely addressed by enhancing pointcut expressiveness, that is, by improving the mechanisms for expressing complex pointcuts by combining primitive pointcut descriptors. This thesis makes the case for providing aspect-oriented tools with fine-grained join points, corresponding to primitive pointcut descriptors on their own, and encompassing complex behaviours.

In addition, LOOPSAJ —the extension to `abc` that provides AspectJ with a join point for loops, as presented in Chapter 4— has been released[1] under an open source license. This both demonstrates the practical implementation realised for this thesis and gives other researchers willing to explore new join points a comprehensive example of a complex extension.

6.1.3 Performance evaluation

The performance results obtained in Chapter 5 depend on both the use of aspects and the use of Java. Because AspectJ relies on the introduction of new accessory methods into classes to implement the aspects, some degradation of the performance was to be expected. However, results on the fastest JVM show that the overhead of using aspects can be substantially reduced to a point where it is almost unnoticeable. Aspects can therefore contribute to improving speed of execution (via parallelisation) without introducing a penalty due to the higher abstraction, as long as the underlying compiler (whether static or just-in-time) optimises away this overhead. Optimisations of Java code and Java Virtual Machines are an on-going subject of research. Improvements in this domain will definitely be beneficial to Java-based aspect-oriented code. In addition, optimising the specific causes of overheads introduced by aspects is also an on-going subject of research [ACH+04b, ACH+05a] which has already provided successful results.

---

[1] LOOPSAJ can be obtained from: `http://www.cs.manchester.ac.uk/cnc/projects/loopsaj/`.

6.2   Critique

A join point such as the join point for loops provides aspect-orientation with more expressiveness regarding fine-grained and complex behaviours. However, the practical problems related to the loop selection (see Section 4.5) show that having fine-grained join points is not sufficient unless these are accompanied by more expressive pointcut description mechanisms.

Pointcut expressiveness is a problem that has been generally identified in AOP [OMB05]. Relying on name patterns for writing pointcuts is convenient but unsatisfactory. For example, matching the methods that "set" some values in a class —such as `setX(int X)`, `setY(int Y)`, and `setXY(int X, int Y)` in a `Point` class— can be done using a pointcut descriptor based on the regular expression "`set*`", but, unfortunately, this mechanism relies on compliance with naming conventions for the components of an application. Even without taking foreign languages into consideration, pattern matching may have undesired side-effects. The expression "`set*`" might also match methods such as "`setup()`", which might not be intended. Since the loop join point is not associated with a named signature (except via its arguments), pointcut expressiveness becomes particularly important for selecting loops.

For practical reasons, in particular for being able to execute test programs, the base model used in this thesis has been that of AspectJ. The weaving model of AspectJ used since version 1.1 is entirely based on the bytecode. As a result, this model works even on third-party classes for which the source code is not available. However, this constraint also implies a loss of the abstraction used by the programmer in the source code. In particular, this implies that potential loop labels or other forms of loosely coupled annotations[2] are lost, whereas they might have helped improve the selection mechanisms if source-based weaving had been available. A source-based approach has also been investigated (see Appendix B), but the underlying models and tools were not mature enough to be used as a basis for the final work. This problem might have a hybrid solution that would keep

---

[2] Recent developments in AspectJ have introduced the ability to write pointcut descriptors based on Java 5 annotations. Although this may seem a step back towards explicit annotations and compiler directives, this in fact provides a higher abstraction so long as these annotation do not describe the implementation explicitly. In the case of refactored methods, annotations such as `GraphicRoutine` or even `Parallelisable` might be acceptable, but annotations such as `ParallelisableUsingMPIon4Machines` would be too descriptive and render the use of aspects useless.

more information about the source-code in the bytecode representation (or in the intermediate representation used within the compiler), at the expense of the possibility to use aspects on arbitrary bytecodes.

More expressiveness could also be gained by using genericity and, in particular, fine-grained genericity [KR05].

6.3   Related and future work

Although the applications presented in this thesis are focussed on performance improvement through parallelisation, the join point for loops presented in Chapter 4 could be applied to other concerns, such as performance monitoring and profiling, using the methods described in [DHS+03, Bod05], or checkpoint-and-restart mechanisms, which may require substantial aspects. Moreover, simple timers are trivial but useful to implement using aspects.

This thesis has provided the basis for aspect-oriented handling of performance at application level. However, the performance of an application also depends on other layers of the system. The next step in this line of research could consist of trying to address concerns that cut across all the elements of the system, from hardware to application, via operating system and libraries. Encapsulating the vertical-profiling concern [HSDH04] into aspects could be an example of a cross-system concern.

The main aspect-oriented tools currently rely on object-oriented designs. This thesis has shown that it is possible to use AspectJ to intervene at a fundamentally procedural element —the loop— either by turning the loop into an object (in Section 3.2) or by providing a join point for loops (in Chapter 4). However, aspect-orientation, as a principle, need not rely on object-orientation. This is of particular importance for dealing with legacy code from the scientific community. Aspect-orientation has recently been brought to COBOL [LS05], in which billions of lines of legacy code have been written, and are still being written.[3] The main language used by the scientific community, Fortran, has evolved from Fortran 77, procedural, to Fortran 90, which contains elements of object-orientation. Perhaps Fortran 2015 will be aspect-oriented?

---

[3]This formulation may seem awkward, but the code written today is the legacy code of tomorrow.

Appendix A

AspectJ syntax guide

AspectJ[1] is an aspect-oriented extension to Java. The language is fully compatible with pure Java. However, it introduces new kinds of structures and new keywords to write aspects. This appendix presents a summary of the syntax of AspectJ version 1.2.

"AspectJ adds to Java just one new concept, a join point – and that's really just a name for an existing Java concept. It adds to Java only a few new constructs: pointcuts, advice, inter-type declarations and aspects. Pointcuts and advice dynamically affect program flow, inter-type declarations statically affect a program's class hierarchy, and aspects encapsulate these new constructs.

A join point *is a well-defined point in the program flow. A* pointcut *picks out certain join points and values at those points. A piece of* advice *is code that is executed when a join point is reached. These are the dynamic parts of AspectJ.*

AspectJ also has different kinds of inter-type declarations *that allow the programmer to modify a program's static structure, namely, the members of its classes and the relationship between classes.*

AspectJ's aspects *are the unit of modularity for crosscutting concerns. They behave somewhat like Java classes, but may also include pointcuts, advice and inter-type declarations"*[asp].

---

[1]`http://www.eclipse.org/aspectj/`

A.1 General structure of aspects

In AspectJ, aspects are syntactically similar to Java classes. Aspects are defined via the "aspect" keyword, where "class" would have been used to define a class in Java. Aspects can contain several categories of members:

- Java classes, methods, and fields, in the same way as they would be contained in a class;

- *inter-type declarations (ITD)* (also know as *introductions*) make it possible to intervene in the structure of other classes or aspects, by adding new members (see Section A.2);

- *pointcut descriptors*: these can be named and are formed by combinations of conjunctions and disjunctions of pointcut expressions—including primitive pointcuts (see Section A.3);

- *pieces of advice*: *before*, *after* or *around* pieces of advice can be considered as the aspect equivalents of methods. They do not have names, but they contain a pointcut (named or anonymous). They contain the Java instructions to execute when encountering join points matched by their pointcut (see Section A.4); and

- *declarations* (which are beyond the scope of this appendix).

A.2 Inter-type declarations

Inter-type declarations make it possible to add new members (fields and methods) to classes, via an aspect. A basic example is shown in Listing A.1. Without aspect ToStringAspect, class Test does not override method toString() defined in Object. Aspect ToStringAspect introduces method Test.toString() into class Test. This is a modification of the structure of the class that is visible throughout the system.

Listing A.1: Inter-type declaration example.

```
public class Test {
    int value = 10 ;

    public static void main(String[] args) {
        Test t = new Test() ;
        System.out.println("Test: "+t) ;
    }
}

public aspect ToStringAspect {
    public String Test.toString() {
        return Integer.toString(value) ;
    }
}
```

A.3   Pointcut descriptors

The following is an extract from the AspectJ programming guide [asp].

A pointcut is a program element that picks out join points and exposes data from the execution context of those join points. Pointcuts are used primarily by advice. They can be composed with boolean operators to build up other pointcuts. The primitive pointcuts and combinators provided by the language are:

call(MethodPattern) Picks out each method call join point whose signature matches MethodPattern.

execution(MethodPattern) Picks out each method execution join point whose signature matches MethodPattern.

get(FieldPattern) Picks out each field reference join point whose signature matches FieldPattern. [Note that references to constant fields (static final fields bound to a constant string object or primitive value) are not join points, since Java requires them to be inlined.]

set(FieldPattern) Picks out each field set join point whose signature matches FieldPattern. [Note that the initializations of constant fields (static final fields where the initializer is a constant string object or primitive value) are not join points, since Java requires their references to be inlined.]

call(ConstructorPattern) Picks out each constructor call join point whose signature matches ConstructorPattern.

execution(ConstructorPattern) Picks out each constructor execution join point whose signature matches ConstructorPattern.

initialization(ConstructorPattern) Picks out each object initialization join point whose signature matches ConstructorPattern.

preinitialization(ConstructorPattern) Picks out each object pre-initialization join point whose signature matches ConstructorPattern.

staticinitialization(TypePattern) Picks out each static initializer execution join point whose signature matches TypePattern.

handler(TypePattern) Picks out each exception handler join point whose signature matches TypePattern.

adviceexecution() Picks out all advice execution join points.

within(TypePattern) Picks out each join point where the executing code is defined in a type matched by TypePattern.

withincode(MethodPattern) Picks out each join point where the executing code is defined in a method whose signature matches MethodPattern.

withincode(ConstructorPattern) Picks out each join point where the executing code is defined in a constructor whose signature matches ConstructorPattern.

cflow(Pointcut) Picks out each join point in the control flow of any join point P picked out by Pointcut, including P itself.

cflowbelow(Pointcut) Picks out each join point in the control flow of any join point P picked out by Pointcut, but not P itself.

this(Type or Id) Picks out each join point where the currently executing object (the object bound to this) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any join points from static contexts.

target(Type or Id) Picks out each join point where the target object (the object on which a call or field operation is applied to) is an instance of Type, or of the type of the identifier Id (which must be bound in the enclosing advice or pointcut definition). Will not match any calls, gets, or sets of static members.

`args(Type or Id, ...)` Picks out each join point where the arguments are instances of a type of the appropriate type pattern or identifier.

`PointcutId(TypePattern or Id, ...)` Picks out each join point that is picked out by the user-defined pointcut designator named by `PointcutId`.

`if(BooleanExpression)` Picks out each join point where the boolean expression evaluates to `true`. The boolean expression used can only access static members, parameters exposed by the enclosing pointcut or advice, and `thisJoinPoint` forms. In particular, it cannot call non-static methods on the aspect or use return values or exceptions exposed by after advice.

`! Pointcut` Picks out each join point that is not picked out by `Pointcut`.

`Pointcut0 && Pointcut1` Picks out each join point that is picked out by both `Pointcut0` and `Pointcut1`.

`Pointcut0 || Pointcut1` Picks out each join point that is picked out by either pointcuts. `Pointcut0` or `Pointcut1`.

`( Pointcut )` Picks out each join point picked out by `Pointcut`.

Pointcut definition

Pointcuts are defined and named by the programmer with the `pointcut` declaration.

```
pointcut publicIntCall(int i):
    call(public * *(int)) && args(i);
```

A named pointcut may be defined in either a class or aspect, and is treated as a member of the class or aspect where it is found. As a member, it may have an access modifier such as `public` or `private`.

```
class C {
    pointcut publicCall(int i):
        call(public * *(int)) && args(i);
}

class D {
    pointcut myPublicCall(int i):
        C.publicCall(i) && within(SomeType);
}
```

Pointcuts that are not final may be declared abstract, and defined without a body. Abstract pointcuts may only be declared within abstract aspects.

```
abstract aspect A {
    abstract pointcut publicCall(int i);
}
```

In such a case, an extending aspect may override the abstract pointcut.

```
aspect B extends A {
    pointcut publicCall(int i): call(public Foo.m(int)) && args(i);
}
```

For completeness, a pointcut with a declaration may be declared `final`.

Though named pointcut declarations appear somewhat like method declarations, and can be overridden in subaspects, they cannot be overloaded. It is an error for two pointcuts to be named with the same name in the same class or aspect declaration.

The scope of a named pointcut is the enclosing class declaration. This is different than the scope of other members; the scope of other members is the enclosing class *body*. This means that the following code is legal:

```
aspect B percflow(publicCall()) {
    pointcut publicCall(): call(public Foo.m(int));
}
```

Context exposure

Pointcuts have an interface; they expose some parts of the execution context of the join points they pick out. For example, the PublicIntCall above exposes the first argument from the receptions of all public unary integer methods. This context is exposed by providing typed formal parameters to named pointcuts and advice, like the formal parameters of a Java method. These formal parameters are bound by name matching.

On the right-hand side of advice or pointcut declarations, in certain pointcut designators, a Java identifier is allowed in place of a type or collection of types. The pointcut designators that allow this are `this`, `target`, and `args`. In all such cases, using an identifier rather than a type does two things. First, it selects join points as based on the type of the formal parameter. So the pointcut

```
pointcut intArg(int i): args(i);
```

picks out join points where an `int` (or a `byte`, `short`, or `char`; anything assignable to an `int`) is being passed as an argument. Second, though, it makes the value of that argument available to the enclosing advice or pointcut.

Values can be exposed from named pointcuts as well, so

```
pointcut publicCall(int x): call(public *.*(int)) && intArg(x);
pointcut intArg(int i): args(i);
```

is a legal way to pick out all calls to public methods accepting an int argument, and exposing that argument.

There is one special case for this kind of exposure. Exposing an argument of type Object will also match primitive typed arguments, and expose a "boxed" version of the primitive. So,

```
pointcut publicCall(): call(public *.*(..)) && args(Object);
```

will pick out all unary methods that take, as their only argument, subtypes of Object (i.e., not primitive types like int), but

```
pointcut publicCall(Object o): call(public *.*(..)) && args(o);
```

will pick out all unary methods that take any argument: And if the argument was an int, then the value passed to advice will be of type java.lang.Integer.

The "boxing" of the primitive value is based on the *original* primitive type. So in the following program

```
public class InstanceOf {

  public static void main(String[] args) {
    doInt(5);
  }

  static void doInt(int i) {  }
}

aspect IntToLong {
  pointcut el(long l) :
      execution(* doInt(..)) && args(l);

  before(Object o) : el(o) {
      System.out.println(o.getClass());
  }
}
```

The pointcut will match and expose the integer argument, but it will expose it as an Integer, not a Long.

Primitive pointcuts

Method-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture method call and execution join points.

- `call(MethodPattern)`

- `execution(MethodPattern)`

Field-related pointcuts

AspectJ provides two primitive pointcut designators designed to capture field reference and set join points:

- `get(FieldPattern)`

- `set(FieldPattern)`

All set join points are treated as having one argument, the value the field is being set to, so at a set join point, that value can be accessed with an `args` pointcut. So an aspect guarding a static integer variable x declared in type T might be written as

```
aspect GuardedX {
    static final int MAX_CHANGE = 100;
    before(int newval): set(static int T.x) && args(newval) {
        if (Math.abs(newval - T.x) > MAX_CHANGE)
            throw new RuntimeException();
    }
}
```

Object creation-related pointcuts

AspectJ provides primitive pointcut designators designed to capture the initializer execution join points of objects.

- `call(ConstructorPattern)`

- `execution(ConstructorPattern)`

- `initialization(ConstructorPattern)`

- `preinitialization(ConstructorPattern)`

Class initialization-related pointcuts

AspectJ provides one primitive pointcut designator to pick out static initializer execution join points.

- `staticinitialization(TypePattern)`

Exception handler execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of exception handlers:

- `handler(TypePattern)`

All handler join points are treated as having one argument, the value of the exception being handled. That value can be accessed with an `args` pointcut. So an aspect used to put `FooException` objects into some normal form before they are handled could be written as

```
aspect NormalizeFooException {
    before(FooException e): handler(FooException) && args(e) {
        e.normalize();
    }
}
```

Advice execution-related pointcuts

AspectJ provides one primitive pointcut designator to capture execution of advice

- `adviceexecution()`

This can be used, for example, to filter out any join point in the control flow of advice from a particular aspect.

```
aspect TraceStuff {
    pointcut myAdvice(): adviceexecution() && within(TraceStuff);

    before(): call(* *(..)) && !cflow(myAdvice) {
        // do something
    }
}
```

State-based pointcuts

Many concerns cut across the dynamic times when an object of a particular type is executing, being operated on, or being passed around. AspectJ provides primitive pointcuts that capture join points at these times. These pointcuts use the dynamic types of their objects to pick out join points. They may also be used to expose the objects used for discrimination.

- `this(Type or Id)`

- `target(Type or Id)`

The `this` pointcut picks out each join point where the currently executing object (the object bound to `this`) is an instance of a particular type. The `target` pointcut picks out each join point where the target object (the object on which a method is called or a field is accessed) is an instance of a particular type. Note that `target` should be understood to be the object the current join point is transfering control to. This means that the target object is the same as the current object at a method execution join point, for example, but may be different at a method call join point.

- `args(Type or Id or "..", ...)`

The args pointcut picks out each join point where the arguments are instances of some types. Each element in the comma-separated list is one of four things. If it is a type name, then the argument in that position must be an instance of that type. If it is an identifier, then that identifier must be bound in the enclosing advice or pointcut declaration, and so the argument in that position must be an instance of the type of the identifier (or of any type if the identifier is typed to Object). If it is the "*" wildcard, then any argument will match, and if it is the special wildcard "..", then any number of arguments will match, just like in signature patterns. So the pointcut

```
args(int, .., String)
```

will pick out all join points where the first argument is an `int` and the last is a `String`.

Control flow-based pointcuts

Some concerns cut across the control flow of the program. The `cflow` and `cflowbelow` primitive pointcut designators capture join points based on control flow.

- `cflow(Pointcut)`

- `cflowbelow(Pointcut)`

The `cflow` pointcut picks out all join points that occur between entry and exit of each join point P picked out by `Pointcut`, including P itself. Hence, it picks out the join points *in* the control flow of the join points picked out by `Pointcut`.

The `cflowbelow` pointcut picks out all join points that occur between entry and exit of each join point P picked out by `Pointcut`, but not including P itself. Hence, it picks out the join points *below* the control flow of the join points picked out by `Pointcut`.

Context exposure from control flows The `cflow` and `cflowbelow` pointcuts may expose context state through enclosed `this`, `target`, and `args` pointcuts.

Anytime such state is accessed, it is accessed through the *most recent* control flow that matched. So the "current arg" that would be printed by the following program is zero, even though it is in many control flows.

```java
class Test {
    public static void main(String[] args) {
        fact(5);
    }
    static int fact(int x) {
        if (x == 0) {
            System.err.println("bottoming out");
            return 1;
        }
        else return x * fact(x - 1);
    }
}


aspect A {
    pointcut entry(int i): call(int fact(int)) && args(i);
    pointcut writing(): call(void println(String)) && ! within(A);

    before(int i): writing() && cflow(entry(i)) {
        System.err.println("Current arg is " + i);
    }
}
```

It is an error to expose such state through *negated* control flow pointcuts, such as within `!cflowbelow(P)`.

Program text-based pointcuts

While many concerns cut across the runtime structure of the program, some must deal with the lexical structure. AspectJ allows aspects to pick out join points based on where their associated code is defined.

- `within(TypePattern)`

- `withincode(MethodPattern)`

- `withincode(ConstructorPattern)`

The `within` pointcut picks out each join point where the code executing is defined in the declaration of one of the types in `TypePattern`. This includes the class initialization, object initialization, and method and constructor execution join points for the type, as well as any join points associated with the statements and expressions of the type. It also includes any join points that are associated with code in a type's nested types, and that type's default constructor, if there is one.

The `withincode` pointcuts picks out each join point where the code executing is defined in the declaration of a particular method or constructor. This includes the method or constructor execution join point as well as any join points associated with the statements and expressions of the method or constructor. It also includes any join points that are associated with code in a method or constructor's local or anonymous types.

Expression-based pointcuts

- `if(BooleanExpression)`

The if pointcut picks out join points based on a dynamic property. It's syntax takes an expression, which must evaluate to a boolean true or false. Within this expression, the `thisJoinPoint` object is available. So one (extremely inefficient) way of picking out all call join points would be to use the pointcut

```
if(thisJoinPoint.getKind().equals("call"))
```

Note that the order of evaluation for pointcut expression components at a join point is undefined. Writing `if` pointcuts that have side-effects is considered bad style and may also lead to potentially confusing or even changing behavior with regard to when or if the test code will run.

Signatures

One very important property of a join point is its signature, which is used by many of AspectJ's pointcut designators to select particular join points.

Methods

Join points associated with methods typically have method signatures, consisting of a method name, parameter types, return type, the types of the declared (checked) exceptions, and some type that the method could be called on (below called the "qualifying type").

At a method call join point, the signature is a method signature whose qualifying type is the static type used to *access* the method. This means that the signature for the join point created from the call `((Integer)i).toString()` is different than that for the call `((Object)i).toString()`, even if `i` is the same variable.

At a method execution join point, the signature is a method signature whose qualifying type is the declaring type of the method.

Fields

Join points associated with fields typically have field signatures, consisting of a field name and a field type. A field reference join point has such a signature, and no parameters. A field set join point has such a signature, but has a has a single parameter whose type is the same as the field type.

Constructors

Join points associated with constructors typically have constructor signatures, consisting of a parameter types, the types of the declared (checked) exceptions, and the declaring type.

At a constructor call join point, the signature is the constructor signature of the called constructor. At a constructor execution join point, the signature is the constructor signature of the currently executing constructor.

At object initialization and pre-initialization join points, the signature is the constructor signature for the constructor that started this initialization: the first constructor entered during this type's initialization of this object.

Others

At a handler execution join point, the signature is composed of the exception type that the handler handles.

At an advice execution join point, the signature is composed of the aspect type, the parameter types of the advice, the return type (void for all but around advice) and the types of the declared (checked) exceptions.

Matching

The `withincode`, `call`, `execution`, `get`, and `set` primitive pointcut designators all use signature patterns to determine the join points they describe. A signature pattern is an abstract description of one or more join-point signatures. Signature patterns are intended to match very closely the same kind of things one would write when declaring individual members and constructors.

Method declarations in Java include method names, method parameters, return types, modifiers like static or private, and throws clauses, while constructor declarations omit the return type and replace the method name with the class name. The start of a particular method declaration, in class `Test`, for example, might be

```
class C {
    public final void foo() throws ArrayOutOfBoundsException { ... }
}
```

In AspectJ, method signature patterns have all these, but most elements can be replaced by wildcards. So

```
call(public final void C.foo() throws ArrayOutOfBoundsException)
```

picks out call join points to that method, and the pointcut

```
call(public final void *.*() throws ArrayOutOfBoundsException)
```

picks out all call join points to methods, regardless of their name name or which class they are defined on, so long as they take no arguments, return no value, are both `public` and `final`, and are declared to throw `ArrayOutOfBounds` exceptions.

The defining type name, if not present, defaults to *, so another way of writing that pointcut would be

```
call(public final void *() throws ArrayOutOfBoundsException)
```

Formal parameter lists can use the wildcard .. to indicate zero or more arguments, so

```
execution(void m(..))
```

picks out execution join points for void methods named `m`, of any number of arguments, while

```
execution(void m(.., int))
```

picks out execution join points for void methods named `m` whose last parameter is of type `int`.

The modifiers also form part of the signature pattern. If an AspectJ signature pattern should match methods without a particular modifier, such as all non-public methods, the appropriate modifier should be negated with the `!` operator. So,

```
withincode(!public void foo())
```

picks out all join points associated with code in null non-public void methods named `foo`, while

```
withincode(void foo())
```

picks out all join points associated with code in null void methods named `foo`, regardless of access modifier.

Method names may contain the `*` wildcard, indicating any number of characters in the method name. So

```
call(int *())
```

picks out all call join points to `int` methods regardless of name, but

```
call(int get*())
```

picks out all call join points to `int` methods where the method name starts with the characters "get".

AspectJ uses the `new` keyword for constructor signature patterns rather than using a particular class name. So the execution join points of private null constructor of a class C defined to throw an ArithmeticException can be picked out with

```
execution(private C.new() throws ArithmeticException)
```

Matching based on the declaring type

The signature-matching pointcuts all specify a declaring type, but the meaning varies slightly for each join point signature, in line with Java semantics.

When matching for pointcuts `withincode`, `get`, and `set`, the declaring type is the class that contains the declaration.

When matching method-call join points, the declaring type is the static type used to access the method. A common mistake is to specify a declaring type for the `call` pointcut that is a subtype of the originally-declaring type. For example, given the class

```
class Service implements Runnable {
  public void run() { ... }
}
```

the following pointcut

```
call(void Service.run())
```

would fail to pick out the join point for the code

```
((Runnable) new Service()).run();
```

Specifying the originally-declaring type is correct, but would pick out any such call (here, calls to the `run()` method of any Runnable). In this situation, consider instead picking out the target type:

```
call(void run()) && target(Service)
```

When matching method-execution join points, if the execution pointcut method signature specifies a declaring type, the pointcut will only match methods declared in that type, or methods that override methods declared in or inherited by that type. So the pointcut

```
execution(public void Middle.*())
```

picks out all method executions for public methods returning void and having no arguments that are either declared in, or inherited by, Middle, even if those methods are overridden in a subclass of Middle. So the pointcut would pick out the method-execution join point for Sub.m() in this code:

```
class Super {
  protected void m() { ... }
}
class Middle extends Super {
}
class Sub extends Middle {
  public void m() { ... }
}
```

A.4 Advice

The following is an extract from the AspectJ programming guide [asp].

> Advice defines pieces of aspect implementation that execute at well-defined points in the execution of the program. Those points can be given either by named pointcuts (like the ones you've seen above) or by anonymous pointcuts. Here is an example of an advice on a named pointcut:
>
> ```
> pointcut setter(Point p1, int newval): target(p1) && args(newval)
>                                         (call(void setX(int) ||
>                                          call(void setY(int)));
>
>
> before(Point p1, int newval): setter(p1, newval) {
>     System.out.println("About to set something in " + p1 +
>                        " to the new value " + newval);
> }
> ```
>
> And here is exactly the same example, but using an anonymous pointcut:
>
> ```
> before(Point p1, int newval): target(p1) && args(newval)
>                               (call(void setX(int)) ||
>                                call(void setY(int))) {
>     System.out.println("About to set something in " + p1 +
>                        " to the new value " + newval);
> }
> ```
>
> Here are examples of the different advice:
> This before advice runs just before the join points picked out by the (anonymous) pointcut:
>
> ```
> before(Point p, int x): target(p) && args(x) && call(void setX(int)) {
>     if (!p.assertX(x)) return;
> }
> ```
>
> This after advice runs just after each join point picked out by the (anonymous) pointcut, regardless of whether it returns normally or throws an exception:
>
> ```
> after(Point p, int x): target(p) && args(x) && call(void setX(int)) {
>     if (!p.assertX(x)) throw new PostConditionViolation();
> }
> ```
>
> This after returning advice runs just after each join point picked out by the (anonymous) pointcut, but only if it returns normally. The return value can be accessed, and is named x here. After the advice runs, the return value is returned:

```
after(Point p) returning(int x): target(p) && call(int getX()) {
    System.out.println("Returning int value " + x + " for p = " + p);
}
```

This after throwing advice runs just after each join point picked out by the (anonymous) pointcut, but only when it throws an exception of type Exception. Here the exception value can be accessed with the name e. The advice re-raises the exception after it's done:

```
after() throwing(Exception e): target(Point) && call(void setX(int)) {
    System.out.println(e);
}
```

This around advice traps the execution of the join point; it runs *instead* of the join point. The original action associated with the join point can be invoked through the special proceed call:

```
void around(Point p, int x): target(p)
                          && args(x)
                          && call(void setX(int)) {
    if (p.assertX(x)) proceed(p, x);
    p.releaseResources();
}
```

Appendix B

A source-code and structural approach

B.1 Join point as point in the structure of the program

The definition of *join point* in the Aspect-Oriented Software Development book [FECA04] is about *"well-defined places in the structure or the execution of a program"*, whereas the AspectJ definition talks exclusively about points in the execution of a program[1].

Chapters 3 and 4 have used an approach based on the AspectJ model. This appendix presents another line of investigation which did not lead to experimental results due to the lack of underlying tools. The tools that might have been used (JTransformer[2] [Rho03, Bar03] and LogicAJ[3] [RK04, KRH, KR05]) were research prototypes not mature enough for reliable experimentation at the time of writing.

JTransformer (see Section B.2.1) provides a low-level representation of the source code and source-code transformations aimed at programmers familiar with Prolog [Bra90]. LogicAJ (see Section B.2.2) provides a higher-level abstraction, aimed at programmer familiar with Java and AspectJ.

---

[1]In fact, even AspectJ has got pointcut descriptors to match join points according to the structure of the program: `within` and `withincode` are pointcut descriptors utilised to refer to a subset of the lexical scope of the program.

[2]`http://roots.iai.uni-bonn.de/research/jtransformer/`

[3]`http://roots.iai.uni-bonn.de/research/logicaj/`

B.2   Tools

B.2.1   JTransformer

JTransformer is a transformation engine[4] that uses Prolog in order to transform the abstract syntax tree (AST) of Java units. Each node of the syntax tree is represented by a Prolog fact. It is then possible to use Prolog predicates to add, remove and modify the base of facts, that is, to change the AST.

Using JTransformer, the Java "`for`" construct would be represented by a fact of the following form:

```
forLoopT(#id,#parent,#enclMethod,[#init_1,...]
          #condition,[#step_1,...],#body)
```

where:

- `#id` would be a unique identifier (all nodes have an identifier),

- `#parent` would be the identifier of the parent node (usually the enclosing block),

- `#enclMethod` would be the identifier of the enclosing method,

- `[#init_1,...]` would be the list of identifiers of the initialisation statements,

- `#condition` would be the identifiers of the condition expression,

- `[#step_1,...]` would be the list of identifiers of the updating statements, and

- `#body` would be the identifiers of the loop body.

Other Java constructs are represented similarly, for example with `assignopT` (for assignment statements) or `localDefT` (for statements defining local variables). The Java method shown in Listing B.1 is represented by the Prolog facts shown in Listing B.2, which are sorted in the order of the AST in Listing B.3.

JTransformer could be used as a back-end for a tool that provides *"fine-grained genericity"* [KR05], that is, genericity at any level of the syntax tree, including Java loop constructs.

---

[4]JTransformer comes in the form of a plugin for Eclipse.

Listing B.1: Sample Java method.

```
public void test() {
    int[] a = new int[10] ;
    int[] b = new int[10] ;
    for (int i = 0 ; i < 10 ; i++) {
        a[i] = 0 ;
    }
    for (int i = 0 ; i < 10 ; i++) {
        b[i] = 0 ;
    }
}
```

B.2.2   LogicAJ

LogicAJ[5] is an extension to the AspectJ language that provides genericity and interference analysis. It is a source-to-source transformation engine that uses JTransformer as a back-end. It is in fact a completely different implementation that is not fully compatible with AspectJ but has different objectives, in particular genericity through the use of *logic meta-variables* (LMV). "*A* meta-variable *is a variable that ranges over syntactic entities of the base language. [...] A* logic variable *is a variable that can only be bound to values by the evaluation of predicates that take the variable as an argument. [... A* logic meta-variable *...] is both, a logic variable and a meta-variable*" [KR05].

Listing B.4 [Rho04] shows an example application [6] of LogicAJ. The variables whose names start with a question mark are the LMVs (the double question mark is a variation for LMVs matching several parameters). Although the syntax looks similar to that of AspectJ, binding the LMVs in the pointcut follows the mode of reasoning of Prolog (i.e. first order predicate logic). In this example, the Prolog *goal* (or *pointcut* in LogicAJ's terminology) will be satisfied if and only if there is a matching combination of `?Class`, `??args` and `?mock`, where there exists a call to the constructor of `?Class` taking `??args` as arguments, there exists a string `?mock` made of the concatenation of the name of the class `?Class` and `"Mock"`, and there exists a class named by the string value of `?mock`.

Unfortunately, at the time of writing, LogicAJ is still in development, and only partially working alpha versions have been released.

---

[5]LogicAJ comes in the form of a plugin for Eclipse.
[6]This particular example has been taken from the LogicAJ example files.

Listing B.2: JTransformer/Prolog facts for the method in Listing B.1.

```
assignT(15646, 15645, 15448, 15647, 15648).
assignT(15660, 15659, 15448, 15661, 15662).
blockT(15628, 15448, 15448, [15629, 15630, 15631, 15632]).
blockT(15640, 15631, 15448, [15645]).
blockT(15654, 15632, 15448, [15659]).
execT(15645, 15640, 15448, 15646).
execT(15659, 15654, 15448, 15660).
forLoopT(15631, 15628, 15448, [15637], 15638, [15639], 15640).
forLoopT(15632, 15628, 15448, [15651], 15652, [15653], 15654).
identT(15642, 15638, 15448, i, 15637).
identT(15644, 15639, 15448, i, 15637).
identT(15649, 15647, 15448, i, 15637).
identT(15650, 15647, 15448, a, 15629).
identT(15656, 15652, 15448, i, 15651).
identT(15658, 15653, 15448, i, 15651).
identT(15663, 15661, 15448, i, 15651).
identT(15664, 15661, 15448, b, 15630).
indexedT(15647, 15646, 15448, 15649, 15650).
indexedT(15661, 15660, 15448, 15663, 15664).
literalT(15634, 15633, 15448, type(basic, int, 0), 10).
literalT(15636, 15635, 15448, type(basic, int, 0), 10).
literalT(15641, 15637, 15448, type(basic, int, 0), 0).
literalT(15643, 15638, 15448, type(basic, int, 0), 10).
literalT(15648, 15646, 15448, type(basic, int, 0), 0).
literalT(15655, 15651, 15448, type(basic, int, 0), 0).
literalT(15657, 15652, 15448, type(basic, int, 0), 10).
literalT(15662, 15660, 15448, type(basic, int, 0), 0).
localDefT(15629, 15628, 15448, type(basic, int, 1), a, 15633).
localDefT(15630, 15628, 15448, type(basic, int, 1), b, 15635).
localDefT(15637, 15631, 15448, type(basic,int,0), i, 15641).
localDefT(15651, 15632, 15448, type(basic,int,0), i, 15655).
methodDefT(15448, 15431, test, [], type(basic, void, 0),
                                 type(basic, void, 0), [], 15628).
modifierT(15448, public).
newArrayT(15633, 15629, 15448, [15634], [], type(basic,int,1)).
newArrayT(15635, 15630, 15448, [15636], [], type(basic,int,1)).
operationT(15638, 15631, 15448, [15642, 15643], <, 0).
operationT(15639, 15631, 15448, [15644], ++, 1).
operationT(15652, 15632, 15448, [15656, 15657], <, 0).
operationT(15653, 15632, 15448, [15658], ++, 1).
```

Listing B.3: JTransformer/Prolog facts for the method in Listing B.1 (sorted according to the syntax tree).

```
methodDefT(15448, 15431, test, [], type(basic, void, 0),
                                   type(basic, void, 0), [], 15628)
/modifierT(15448, public)/
  []
  []
  blockT(15628, 15448, 15448, [15629, 15630, 15631, 15632])
    [ localDefT(15629, 15628, 15448, type(basic, int, 1), a, 15633)
        newArrayT(15633, 15629, 15448, [15634], [], type(basic,int,1))
          [ literalT(15634, 15633, 15448, type(basic, int, 0), 10) ]
          []
      localDefT(15630, 15628, 15448, type(basic, int, 1), b, 15635)
        newArrayT(15635, 15630, 15448, [15636], [], type(basic,int,1))
          [ literalT(15636, 15635, 15448, type(basic, int, 0), 10) ]
          []
      forLoopT(15631, 15628, 15448, [15637], 15638, [15639], 15640)
        [ localDefT(15637, 15631, 15448, type(basic,int,0), i, 15641)
            literalT(15641, 15637, 15448, type(basic, int, 0), 0) ]
        operationT(15638, 15631, 15448, [15642, 15643], <, 0)
          [ identT(15642, 15638, 15448, i, 15637)
            literalT(15643, 15638, 15448, type(basic, int, 0), 10)
          ]
        [ operationT(15639, 15631, 15448, [15644], ++, 1)
            [ identT(15644, 15639, 15448, i, 15637) ]
        ]
        blockT(15640, 15631, 15448, [15645])
          [ execT(15645, 15640, 15448, 15646)
              assignT(15646, 15645, 15448, 15647, 15648)
                indexedT(15647, 15646, 15448, 15649, 15650)
                  identT(15649, 15647, 15448, i, 15637)
                  identT(15650, 15647, 15448, a, 15629)
                literalT(15648, 15646, 15448, type(basic, int, 0), 0)
          ]
      forLoopT(15632, 15628, 15448, [15651], 15652, [15653], 15654)
        [ localDefT(15651, 15632, 15448, type(basic,int,0), i, 15655)
            literalT(15655, 15651, 15448, type(basic, int, 0), 0)
        ]
        operationT(15652, 15632, 15448, [15656, 15657], <, 0)
          [ identT(15656, 15652, 15448, i, 15651)
            literalT(15657, 15652, 15448, type(basic, int, 0), 10)
          ]
        [ operationT(15653, 15632, 15448, [15658], ++, 1)
            [ identT(15658, 15653, 15448, i, 15651) ]
        ]
        blockT(15654, 15632, 15448, [15659])
          [ execT(15659, 15654, 15448, 15660)
              assignT(15660, 15659, 15448, 15661, 15662)
                indexedT(15661, 15660, 15448, 15663, 15664)
                  identT(15663, 15661, 15448, i, 15651)
                  identT(15664, 15661, 15448, b, 15630)
                literalT(15662, 15660, 15448, type(basic, int, 0), 0)
          ]
    ]
```

Listing B.4: Mock objects using LogicAJ.

```
public aspect MockAspect {
   Object around(?mock, ??args):
      // Intercept constructor invocations.
      // Bind ?class to the name of the instantiated class
      // and ?args to the argument list of the invocation

      call(?Class.new(..)) && args(??args) &&

      // Check if a class with name ?class+"Mock" exists
      concat(?Class, "Mock", ?mock) && class(?mock)
    {
      // return instance of mock class
      // includes weave time check for constructor existence
      return new ?mock(??args);
    }
}
```

B.3   Loop fusion

A typical loop optimisation that could not be performed using an AspectJ join point model (because AspectJ join points are points in the *execution* of the program) is the fusion of loops. The AspectJ model does not make it possible to combine two "proceed()" statements corresponding to two different join points (it would not make sense in the AspectJ advice model).

This section sketches an example application of JTransformer that could be used as a basis for an aspect-oriented system permitting aspects to perform loop fusion on Java-code (this has otherwise been done in Lisp [KLM+97]).

The two main facets of this problem are the implementation of the transformations and the abstraction that needs to be provided to enable programmers of aspects to utilise this transformation.

Listing B.5 shows a Prolog rule that could implement the merging of two loops. This rule, mergeForLoops(ForLoopID1, ForLoopID2, NewLoop), is designed in three steps which consists of:

1. matching the two for loops and their components (initialisers, conditions, updating statements and body);

2. checking that the iteration spaces of the two loops are equivalent, via the initialisers, the conditions and the updating statements; and

3. merging the loop bodies into a new block, and creating a new loop. (This is a simplified version that does not take into account the proper insertion of the new loop or the deletion of the merged loops.)

Listing B.5: Merging two loops using JTransformer.

```prolog
% This goal represents the merging of loops ForLoopID1 and ForLoopID2
% to form a loop with identifier NewLoop.
mergeForLoops(ForLoopID1, ForLoopID2, NewLoop) :-
    % The following 2 rules match two loops with the same parent node.
  forLoopT(ForLoopID1, _Parent,_Method, Init1, Cond1, Update1, Body1),
  forLoopT(ForLoopID2, _Parent,_Method, Init2, Cond2, Update2, Body2),

    % The following rule check that the iteration spaces of the
    % loops are equivalent.
  equiv([Init1, Cond1, Update1], [Init2, Cond2, Update2]),
  !,

    % Creates a new identifier (for a block)
  new_id(NewBlock), !,
    % Creates a new identifier (for a loop)
  new_id(NewLoop), !,
    % The following rule adds a new fact to the prolog base. This
    % represents a block merging the two bodies of the original loops.
  assert(blockT(NewBlock, NewLoop, _EnclMethod, [Body1, Body2])),

    % The following rule adds a new fact to the prolog base. This
    % represents a loop with the original iteration space and the
    % merged body newly created.
  assert(forLoopT(NewLoop, _Parent, _EnclMethod, Init1,
                                      Cond1, Update1, NewBlock)).
```

The most laborious task regarding the implementation is the writing of the rules that check the equivalence of two iteration spaces (equiv([Init1, Cond1, Update1], [Init2, Cond2, Update2])). This example is incomplete, but the difficulty lies in implementing a mechanism capable of checking that (int i=0 ; i<10; i++) in the two loops in Listing B.1 have the same semantics, based on the AST. Having a mechanism flexible enough that would work, for example, if the second loop was using another variable name, if the variable was declared outside the loop, or if the loop was using the while construct is a non-trivial task. This would require semantic analyses using the AST as a starting point. This would more or less consist of writing a Java compiler (or the equivalent of the Soot framework), in Prolog.

Moreover, the abstraction to use for such transformations would be difficult to integrate in the AspectJ advice model. In AspectJ, the advice is executed when the join point is encountered. For merging loops, this is too late, since, by definition, merging two loops requires anticipation of the occurrence of a second loop after the first one. Using an AspectJ advice to do something before, after, or around the occurrence of a given loop is of no use in this respect. The LogicAJ introduction model, which has the potential to perform complex operations on the whole AST, could perhaps help overcome this problem.

B.4   Aspects for refactoring

Another possible use of JTransformer and LogicAJ would consist of writing aspects that perform the refactorings presented in [HG04] and in Section 3.2. Indeed, if LogicAJ was capable of matching loops, its introduction mechanism should make it possible to create new classes based on the content of these loops. Therefore, it should be possible to write aspects, not only for parallelising, but also for refactoring double loop nests into their object-oriented counterparts (as is done manually in Section 3.2), for example. A possible design for this solution could consist of providing LogicAJ with the means of using its introduction mechanism to let the aspect programmer explicitly handle closures at certain join points.

This approach could also circumvent the performance problem due to the creation of many instances of `Runnable` objects exactly at the loop join point when parallelising using LOOPSAJ, as shown in Chapter 5.

A prototype implementation of semi-automated loop refactorings has been written using an early version of JTransformer in the context of this thesis. However, it did not fully address the problem of recognising semanticaly equivalent codes written differently (for example, `while`- and `for`-loops), and LogicAJ was not available at the time. Without a higher-level abstraction, it is hardly realistic to imagine software engineers used to Java writing Prolog-based transformations of the AST to improve the design of their applications.

Appendix C

Listings

C.1 Object-oriented loops

C.1.1 RectangleLoopA

Listing C.1: Class `RectangleLoopA` and interface `Runnable2DLoopBody`.

```java
/**
 * This class represents a loop body that has two parameters
 * (typically i and j).
 * It is meant to be used in conjunction with RectangleLoopA.
 */

public interface Runnable2DLoopBody {
    /**
     * Classes implementing this interface have to implement
     * this method 'run(int, int)' that will be representing
     * the body of a double nested loop.
     * Programmers should pay attention to the order of the
     * indexes (when used with RectangleLoopA, for example)
     * for a better use of the cache lines.
     *
     * @param i Outer loop index when using RectangleLoopA.
     * @param j Inner loop index when using RectangleLoopA.
     */
    void run(int i, int j);
}



/**
 * This class is used for providing a model of double nested loops.
 * The body of the loop must be encapsulated in the run(int, int)
 * method of an instance of Runnable2DLoopBody. The run() method of
 * this class will iterate through the loop body over a rectangle
 * whose boundaries are given as arguments to the constructor.
 */
```

```java
public class RectangleLoopA implements Runnable {
    final protected Runnable2DLoopBody loopBody;

    final private int minI;
    final private int maxI;
    final private int minJ;
    final private int maxJ;

    /**
     * Constructs the loop.
     * @param loopBody Instance of Runnable2DLoopBody that represents
     *            the body of the loop to be executed.
     * @param minI Minimal value for the outer loop index (inclusive).
     * @param maxI Maximal value for the outer loop index (inclusive).
     * @param minJ Minimal value for the inner loop index (inclusive).
     * @param maxJ Maximal value for the inner loop index (inclusive).
     */
    public RectangleLoopA(
        Runnable2DLoopBody loopBody,
        int minI,
        int maxI,
        int minJ,
        int maxJ) {

        this.loopBody = loopBody ;
        this.minI = minI;
        this.maxI = maxI;
        this.minJ = minJ;
        this.maxJ = maxJ;
    }

    /**
     * This method does the iterations through the loop body according
     * to the values given in the constructor.
     */
    public void run() {
        for (int i = minI; i <= maxI; i++)
            for (int j = minJ; j <= maxJ; j++)
                loopBody.run(i, j);
    }
}
```

Listing C.2: Aspect for implementing multi-threading in RectangleLoopA.

```java
public privileged aspect MultiThreads {

    /* Initialise the number of threads on which to split the loop */
    public static int NUM_PROC ;
    public MultiThreads () {
        NUM_PROC = Integer.parseInt(System.getProperty("numproc","1")) ;
    }
```

```java
/**
 * This class is meant to replace an instance of RectangleLoopA and run the
 * loop it represents on several threads.
 */
class MTRectangleLoopA extends RectangleLoopA {
    /**
     * Array of "smaller" RectangleLoopAs.
     */
    private final Runnable[] subLoops ;

    /**
     * Splits the load on "NUM_PROC" RectangleLoops.
     *
     * @see uk.ac.man.cs.bruno.loops.RectangleLoopA
     * @param loopBody
     *              Instance of Runnable2DLoopBody that represents the body of
     *              the loop to be executed.
     * @param minI Minimal value for the outer loop index (inclusive).
     * @param maxI Maximal value for the outer loop index (inclusive).
     * @param minJ Minimal value for the inner loop index (inclusive).
     * @param maxJ Maximal value for the inner loop index (inclusive).
     */
    public MTRectangleLoopA (
        Runnable2DLoopBody loopBody,
        int minI,
        int maxI,
        int minJ,
        int maxJ) {
            super(loopBody, minI, maxI, minJ, maxJ) ;
            subLoops = new Runnable [NUM_PROC] ;
            int width =
                (int) Math.ceil(
                    ((double) (maxI - minI)) / (double) NUM_PROC);
            for (int k=0; k<NUM_PROC; k++) {
                int min = minI + k*width ;
                int max = minI + (k + 1) * width - 1;
                if (max > maxI)
                    max = maxI;
                subLoops[k] = new RectangleLoopA (loopBody, min, max, minJ, maxJ) ;
            }
        }

    /**
     * Sends all the subloops to the thread pool.
     */
    public void run () {
        threadPool.run(subLoops) ;
    }
}

/**
    This around piece of advice catches creations of instances of RectangleLoopA.
    It transparently replaces the instance of RectangleLoopA by the equivalent
    instance of MTRectangleLoopA, as defined above.
```

```
    */
    RectangleLoopA around (Runnable2DLoopBody loopBody,
                           int minI,
                           int maxI,
                           int minJ,
                           int maxJ):
        call (RectangleLoopA.new(..)) && !within(MTRectangleLoopA) &&
        args (loopBody, minI, maxI, minJ, maxJ) {
            return new MTRectangleLoopA (loopBody, minI, maxI, minJ, maxJ) ;
        }
}
```

C.1.2   RectangleLoopB

Listing C.3: Class `RectangleLoopB`.

```
/**
 * This class is used for providing a model of double nested loops.
 * The programmer must extend this class and implement the
 * loopBody (int, int) method, which will represent the loop body.
 * The run() method of this class will iterate through the loop body
 * over a rectangle whose boundaries are given as arguments to the
 * constructor.
 */

public abstract class RectangleLoopB implements Runnable {
    final public int minI;
    final public int maxI;
    final public int minJ;
    final public int maxJ;

    /**
     * Constructs the loop.
     * @param minI Minimal value for the outer loop index (inclusive).
     * @param maxI Maximal value for the outer loop index (inclusive).
     * @param minJ Minimal value for the inner loop index (inclusive).
     * @param maxJ Maximal value for the inner loop index (inclusive).
     */
    public RectangleLoopB(
        int minI,
        int maxI,
        int minJ,
        int maxJ) {
        this.minI = minI;
        this.maxI = maxI;
        this.minJ = minJ;
        this.maxJ = maxJ;
    }

    /**
     * This method has to be overridden for implementing the body of a
     * double nested loop.
```

```
     * @param i Outer loop index.
     * @param j Inner loop index.
     */
    public abstract void loopBody (int i, int j) ;


    /**
     * This method does the iterations through the loop body according
     * to the values given in the constructor.
     */
    public final void run () {
        for (int i=minI; i<=maxI; i++)
            for (int j=minJ; j<=maxJ; j++)
                loopBody (i, j) ;
    }
}
```

C.1.3   RectangleLoopC

Listing C.4: Class `RectangleLoopC`.

```
/**
 * This class is used for providing a model of double nested loops.
 * The programmer must extend this class and implement the
 * loopDoJRange (int, int) method, which will represent the loop body.
 *
 * The run() method of this class will iterate through
 * loopDoJRange (int i, int minJ, int maxJ) according to the boundary
 * values of i given in the constructor. This representation does not
 * actually implement the inner nested loop. The programmer is expected
 * to do it in loopDoJRange (@see #loopDoJRange (int, int, int)).
 */

public abstract class RectangleLoopC implements Runnable {
    final private int minI;
    final private int maxI;
    final private int minJ;
    final private int maxJ;

    /**
     * Constructs the loop.
     * @param minI Minimal value for the outer loop index (inclusive).
     * @param maxI Maximal value for the outer loop index (inclusive).
     * @param minJ Minimal value for the inner loop index (inclusive).
     * @param maxJ Maximal value for the inner loop index (inclusive).
     */
    public RectangleLoopC(
        int minI,
        int maxI,
        int minJ,
        int maxJ) {

        this.minI = minI;
```

```java
        this.maxI = maxI;
        this.minJ = minJ;
        this.maxJ = maxJ;
    }


    /**
     * The programmer is expected to implement the inner nested loop and
     * the loop body in this method.
     * The reason why the inner loop has not been hard coded in that model
     * is to avoid a method call at the most inner side of the loop, so
     * that maybe the JVM would perform a better loop optimisation (loop
     * unrolling, ...).
     *
     * Thus, <code>loopDoJRange (int i, int minJ, int maxJ)</code> should
     * contain something similar to the following:
     * <pre>
     * public void loopDoJRange (int i, int minJ, int maxJ) {
     *     for (int j=minJ; j <= maxJ; j++) {
     *         // implements the loop body depending on the values of i and j.
     *     }
     * }
     * </pre>
     *
     * @param i Current value of i, as called by loopDoIRange.
     * @param minJ If called via loopDoIRange, will be minJ given to the
     *         constructor.
     * @param maxJ If called via loopDoIRange, will be maxJ given to the
     *         constructor.
     */
    public abstract void loopDoJRange (int i, int minJ, int maxJ) ;



    /**
     * This method will iterate through loopDoJRange for values of i
     * from minI to maxI, with values of minJ and maxJ as given in the
     * constructor.
     *
     * @param minI Minimal value for the outer loop index (inclusive).
     * @param maxI Maximal value for the outer loop index (inclusive).
     */
    public void loopDoIRange (int minI, int maxI) {
        for (int i=minI; i<=maxI; i++)
            loopDoJRange(i, minJ, maxJ) ;
    }


    /**
     * This method does the iterations through the loop body according
     * to the values given in the constructor.
     */
    public final void run () {
        loopDoIRange (minI, maxI) ;
    }
}
```

Bibliography

[ACH+04a] PAVEL AVGUSTINOV, ASKE SIMON CHRISTENSEN, LAURIE HEN-
DREN, SASCHA KUZINS, JENNIFER LHOTÁK, ONDREJ LHOTÁK,
OEGE DE MOOR, DAMIEN SERENI, GANESH SITTAMPALAM, AND
JULIAN TIBBLE. abc: An extensible AspectJ compiler. *Technical
report*, The abc group, aspectbench.org, September 2004.

[ACH+04b] PAVEL AVGUSTINOV, ASKE SIMON CHRISTENSEN, LAURIE HEN-
DREN, SASCHA KUZINS, JENNIFER LHOTÁK, ONDREJ LHOTÁK,
OEGE DE MOOR, DAMIEN SERENI, GANESH SITTAMPALAM, AND
JULIAN TIBBLE. Optimising AspectJ. *Technical report*, The abc group,
aspectbench.org, November 2004.

[ACH+05a] PAVEL AVGUSTINOV, ASKE SIMON CHRISTENSEN, LAURIE HEN-
DREN, SASCHA KUZINS, JENNIFER LHOTÁK, ONDREJ LHOTÁK,
OEGE DE MOOR, DAMIEN SERENI, GANESH SITTAMPALAM, AND
JULIAN TIBBLE. Optimising AspectJ. In *PLDI '05: Proceedings of
the 2005 ACM SIGPLAN conference on Programming language de-
sign and implementation*, pages 117–128, New York, NY, USA, 2005.
ACM Press.

[ACH+05b] PAVEL AVGUSTINOV, ASKE SIMON CHRISTENSEN, LAURIE HEN-
DREN, SASCHA KUZINS, JENNIFER JENNIFER LHOTÁK, ONDREJ
LHOTÁK, OEGE DE MOOR, DAMIEN SERENI, GANESH SITTAM-
PALAM, AND JULIAN TIBBLE. abc: an extensible AspectJ compiler. In
*AOSD '05: Proceedings of the 4th international conference on Aspect-
oriented software development*, pages 87–98, New York, NY, USA,
2005. ACM Press.

[Ald05]   JONATHAN ALDRICH. Open modules: Modular reasoning about advice. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'05), Lecture Notes in Computer Science*, volume 3586, pages 144–168, 2005.

[AM05]   TOMOYUKI AOTANI AND HIDEHIKO MASUHARA. Compiling conditional pointcuts for user-level semantic pointcuts. In *Proceedings of the 4th workshop on Software-Engineering Properties of Languages and Aspect Technologies (SPLAT 2005)*, March 2005.

[AOS]   AOSD STEERING COMMITTEE. *AOSD web site*. `http://www.aosd.net/`.

[Art00]   JOHN K. ARTHUR. Java as an environment for scientific computing. *Lecture Notes in Computational Science and Engineering 10, Advances in Software Tools for Scientific Computing*, pages 179–196, 2000.

[asp]   *AspectJ web site*. `http://www.eclipse.org/aspectj/`.

[ASU85]   ALFRED V. AHO, RAVI SETHI, AND JEFFREY D. ULLMAN. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[BA01]   LODEWIJK BERGMANS AND MEHMET AKSITS. Composing crosscutting concerns using composition filters. *Communications of the ACM*, 44(10):51–57, 2001.

[Bar03]   UWE BARDEY. Abhängigkeitsanalyse von Softwaretransformationen. Diploma thesis, CS Dept. III, University of Bonn, Germany, Feb 2003.

[BB04]   GILAD BRACHA AND JOSHUA BLOCH. JSR 201: Extending the Java^{TM} programming language with enumerations, autoboxing, enhanced for loops and static import, September 2004. `http://jcp.org/en/jsr/detail?id=201`.

[BBKW98]   AART J. C. BIK, PETER J. H. BRINKHAUS, PETER M. W. KNIJNENBURG, AND HARRY A. G. WIJSHOFF. The automatic generation of sparse primitives. *ACM Transactions on Mathematical Software (TOMS)*, 24(2):190–225, 1998.

[BCF⁺98] MARK BAKER, BRYAN CARPENTER, GEOFFREY FOX, SUNG HOON KO, AND XINYING LI. mpiJava: A Java interface to MPI. In *First UK Workshop on Java for High Performance Network Computing, Europar '98*, September 1998.

[BG97] AART J.C. BIK AND DENNIS B. GANNON. javab – a prototype bytecode parallelization tool. *Technical Report TR489*, Computer Science Department, Indiana University, 1997.

[BK00] J. M. BULL AND M. E. KAMBITES. JOMP – an OpenMP-like interface for Java. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 44–53. ACM Press, 2000.

[BMPP01] RONALD F. BOISVERT, JOSÉ MOREIRA, MICHAEL PHILIPPSEN, AND ROLDAN POZO. Java and numerical computing. *Computing in Science & Engineering [see also IEEE Computational Science and Engineering]*, 3(2):18–24, 2001.

[Bod05] RON BODKIN. Performance monitoring with AspectJ, part 1. In *IBM developerWorks AOP@Work series*. September 2005.

[Bra90] IVAN BRATKO. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 2nd edition, 1990.

[BSPF01] J. M. BULL, L. A. SMITH, L. POTTAGE, AND R. FREEMAN. Benchmarking Java against C and fortran for scientific applications. In *Proceedings of ACM Java Grande/ISCOPE Conference*, pages 97–105, 2001.

[BSW⁺00] J. BULL, L. SMITH, M. WESTHEAD, D. HENTY, AND R. DAVEY. Benchmarking Java Grande applications. In *Proceedings of the Second International Conference on The Practical Applications of Java*, pages 63–73, 2000.

[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.

[CCFL98] BRYAN CARPENTER, YUH-JYE CHANG, GEOFFREY FOX, AND XIAOMING LI. Java as a language for scientific parallel programming.

Lecture Notes in Computer Science volume 1366, Languages and Compilers for Parallel Computing, pages 340–354, 1998. Proceedings of the 10th International Workshop, LCPC'97 Minneapolis, Minnesota, USA.

[CCHW05] ADRIAN COLYER, ANDY CLEMENT, GEORGE HARLEY, AND MATTHEW WEBSTER. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Pearson Education, 2005.

[CDK+01] ROHIT CHANDRA, LEONARDO DAGUM, DAVE KOHR, DROR MAYDAN, JEFF MCDONALD, AND RAMESH MENON. *Parallel Programming in OpenMP*. Morgan Kaufmann Publishers, 2001.

[CGJ+00] BRYAN CARPENTER, VLADIMIR GETOV, GLENN JUDD, ANTHONY SKJELLUM, AND GEOFFREY FOX. MPJ: MPI-like message passing for Java. *Concurrency: Practice and Experience*, 12(11):1019–1038, 2000.

[CKF+01] YVONNE COADY, GREGOR KICZALES, MIKE FEELEY, NORM HUTCHINSON, AND JOON SUAN ONG. Structuring operating system aspects: using AOP to improve OS structure modularity. *Communications of the ACM*, 44(10):79–82, 2001.

[Cli05] CURTIS CLIFTON. *A design discipline and language features for modular reasoning in aspect-oriented programs*. PhD thesis, Department of Computer Science, Iowa State University, 2005.

[CRP01] DENIS CAROMEL, JOHN REYNDERS, AND MICHAEL PHILIPPSEN, editors. *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*. ACM Press, 2001.

[CWH00] MARY CAMPIONE, KATHY WALRATH, AND ALISON HUML. *The Java(TM) Tutorial: A Short Course on the Basics*, chapter 8—Threads: Doing Two or More Tasks At Once. Addison-Wesley, `http://java.sun.com/docs/books/tutorial/essential/threads/`, 3rd edition, 2000.

[DBMS] JACK DONGARRA, JIM BUNCH, CLEVE MOLER, AND PETE STEWART. *LINPACK web site*. `http://www.netlib.org/linpack/`.

[DHS⁺03] JONATHAN DAVIES, NICK HUISMANS, RORY SLANEY, SIAN WHIT-
ING, MATTHEW WEBSTER, AND ROBERT BERRY. Aspect oriented
profiler. In *Practitioner Reports, AOSD'2003 conference, Boston*. 2003.

[Dij76] EDSGER W. DIJKSTRA. *A Discipline of Programming*. Prentice-Hall,
1976.

[DT04] RÉMI DOUENCE AND LUC TEBOUL. A pointcut language for
control-flow. In *Proceedings of the 3rd International Conference on
Generative Programming and Component Engineering (GPCE'04)*,
pages 95–114, 2004.

[EAK⁺01] TZILLA ELRAD, MEHMET AKSITS, GREGOR KICZALES, KARL LIEBER-
HERR, AND HAROLD OSSHER. Discussing aspects of AOP. *Com-
munications of the ACM*, 44(10):33–38, 2001.

[EFB01] TZILLA ELRAD, ROBERT E. FILMAN, AND ATEF BADER. Aspect-
oriented programming: Introduction. *Communications of the ACM*,
44(10):29–32, 2001.

[EPC] EPCC. *Java Grande at EPCC*. http://www.epcc.ed.ac.uk/
javagrande/.

[FECA04] ROBERT E. FILMAN, TZILLA ELRAD, SIOBHÁN CLARKE, AND
MEHMET AKŞIT, editors. *Aspect-Oriented Software Development*.
Addison-Wesley, Boston, 2004.

[FF00] R. FILMAN AND D. FRIEDMAN. Aspect-oriented programming is
quantification and obliviousness. In *Workshop on Advanced Separa-
tion of Concerns, OOPSLA 2000*, 2000.

[FSS99] GEOFFREY FOX, KLAUS SCHAUSER, AND MARC SNIR, editors. *Pro-
ceedings of the ACM 1999 conference on Java Grande*. ACM Press,
1999.

[GBNT01] JEFF GRAY, TED BAPTY, SANDEEP NEEMA, AND JAMES TUCK. Han-
dling crosscutting constraints in domain-specific modeling. *Com-
munications of the ACM*, 44(10):87–93, 2001.

[GC00]   K. JOHN GOUGH AND DIANE CORNEY. Evaluating the Java Virtual Machine as a target for languages other than Java. *Lecture Notes in Computer Science 1897, Modular Programming Languages*, pages 278–290, 2000.

[gcj]   *GCJ Home page*. Free Software Foundation, Inc., `http://www.gnu.org/software/gcc/java/`.

[GJSB05] JAMES GOSLING, BILL JOY, GUY STEELE, AND GILAD BRACHA. *Java(TM) Language Specification, Third Edition*. Addison-Wesley, `http://java.sun.com/docs/books/jls/`, 2005.

[GM00]   DENNIS GANNON AND PIYUSH MEHROTRA, editors. *Proceedings of the ACM 2000 conference on Java Grande*. ACM Press, 2000.

[HG04]   BRUNO HARBULOT AND JOHN R. GURD. Using AspectJ to separate concerns in parallel scientific Java code. In *Proceedings of the 3rd international conference on Aspect-Oriented Software Development*, pages 122–131. ACM Press, 2004.

[HG05]   BRUNO HARBULOT AND JOHN R. GURD. A join point for loops in AspectJ. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, pages 11–20. TR #05-05, Department of Computer Science, Iowa State University, March 2005.

[HG06]   BRUNO HARBULOT AND JOHN R. GURD. A join point for loops in AspectJ. In *Proceedings of the 5th international conference on Aspect-Oriented Software Development (to appear)*. ACM Press, 2006.

[HH04]   ERIK HILSDALE AND JIM HUGUNIN. Advice weaving in AspectJ. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 26–35. ACM Press, 2004.

[Hot]   *Java HotSpot Technology*. Sun Microsystems, Inc., `http://java.sun.com/products/hotspot/`.

[HSDH04] MATTHIAS HAUSWIRTH, PETER F. SWEENEY, AMER DIWAN, AND MICHAEL HIND. Vertical profiling: understanding the behavior of object-oriented applications. In *OOPSLA '04: Proceedings of the 19th*

annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, pages 251–269, New York, NY, USA, 2004. ACM Press.

[IEE00]   IEEE. *IEEE Recommended practice for architectural description of software-intensive systems*. 2000. IEEE Std 1471.

[ILG⁺97]   JOHN IRWIN, JEAN-MARC LOINGTIER, JOHN R. GILBERT, GREGOR KICZALES, JOHN LAMPING, ANURAG MENDHEKAR, AND TATIANA SHPEISMAN. Aspect-oriented programming of sparse matrix code. In *International Symposium on Computing in Object-Oriented Parallel Environments (ISCOPE)*, pages 249–256, 1997.

[JGF]   JGF. *Java Grande Forum's website*. http://www.javagrande.org/.

[Jor03]   JOHN JORGENSEN. Improving the precision and correctness of exception analysis in Soot. *Technical report*, Sable Group, McGill University, Montreal, Canada, September 2003.

[KHH⁺01a]   GREGOR KICZALES, ERIK HILSDALE, JIM HUGUNIN, MIK KERSTEN, JEFFREY PALM, AND WILLIAM GRISWOLD. Getting started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[KHH⁺01b]   GREGOR KICZALES, ERIK HILSDALE, JIM HUGUNIN, MIK KERSTEN, JEFFREY PALM, AND WILLIAM G. GRISWOLD. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01), Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, 2001.

[Kic03]   GREGOR KICZALES. Keynote speech at the 2nd aspect-oriented software development conference (AOSD'2003), 2003. http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt.

[KLM⁺97]   GREGOR KICZALES, JOHN LAMPING, ANURAG MENHDHEKAR, CHRIS MAEDA, JEAN-MARC LOINGTIER, AND JOHN IRWIN. Aspect-oriented programming. In MEHMET AKŞIT AND SATOSHI MATSUOKA, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97), Jyväskylä, Finland, Lecture Notes in Computer Science*, volume 1241, pages 220–242, New York, NY, 1997. Springer-Verlag.

[KM05]   GREGOR KICZALES AND MIRA MEZINI. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[KR05]   GÜNTER KNIESEL AND TOBIAS RHO. Generic aspect languages - needs, options and challenges. In *Proceedings of the 2ème Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA 2005)*. Sep 2005.

[KRH]   GÜNTER KNIESEL, TOBIAS RHO, AND STEFAN HANENBERG. Evolvable pattern implementations need generic aspects.

[Kuz04]   SASCHA KUZINS. Efficient implementation of around-advice for the AspectBench Compiler. Master's thesis, Oxford University, UK, September 2004.

[Lad03]   RAMNIVAS LADDAD. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning, 2003.

[Lea00]   DOUG LEA. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, pages 36–43. ACM Press, 2000.

[LMGF05] MIKEL LUJÁN, GIBSON MUKARAKATE, JOHN R. GURD, AND T. L. FREEMAN. DiFoJo: A Java fork/join framework for heterogeneous networks. In *13th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP)*, pages 297–304, 2005.

[LOO01]   KARL LIEBERHERR, DOUG ORLEANS, AND JOHAN OVLINGER. Aspect-oriented programming with adaptive methods. *Communications of the ACM*, 44(10):39–41, 2001.

[Lop02]   CRISTINA VIDEIRA LOPES. Aspect-Oriented Programming: An historical perspective (what's in a name?). *Technical report*, Institute for Software Research, University of California, Irvine, December 2002.

[Lop04]   CRISTINA VIDEIRA LOPES. AOP: A historical perspective (What's in a name?). In Filman et al. [FECA04], pages 97–122.

[LS05]   RALF LÄMMEL AND KRIS DE SCHUTTER. What does aspect-oriented
            programming mean to cobol? In *AOSD '05: Proceedings of the
            4th international conference on Aspect-oriented software development*,
            pages 99–110, New York, NY, USA, 2005. ACM Press.

[LY99]   TIM LINDHOLM AND FRANK YELLIN. *The Java(TM) Virtual Ma-
            chine Specification, Second Edition*. Addison-Wesley, `http://java.`
            `sun.com/docs/books/vmspec/`, 1999.

[Mey88]   BERTRAND MEYER. *Object-oriented Sofware Construction*. Prentice-
            Hall, 1988.

[MFG02]   JOSÉ E. MOREIRA, GEOFFREY C. FOX, AND VLADIMIR GETOV, edi-
            tors. *Proceedings of the 2002 joint ACM-ISCOPE conference on Java
            Grande*. ACM Press, 2002.

[MH02]   JEROME MIECZNIKOWSKI AND LAURIE HENDREN. Decompiling
            Java bytecode: Problems, traps and pitfalls. In *Proceedings of Com-
            piler Construction, 11th International Conference, CC 2002*, pages
            111–127, 2002.

[Mie03]   JEROME MIECZNIKOWSKI. New algorithms for a Java decompiler and
            their implementation in Soot. Master's thesis, McGill University,
            Montréal, Québec, Canada, apr 2003.

[MK03]   HIDEHIKO MASUHARA AND KAZUNORI KAWAUCHI. Dataflow
            pointcut in aspect-oriented programming. *Lecture Notes in Computer
            Science 2895, Proceedings of The First Asian Symposium on Program-
            ming Languages and Systems (APLAS'03)*, pages 105–121, 2003.

[MKL97]   A. MENDHEKAR, G. KICZALES, AND J. LAMPING. RG: A case-
            study for aspect-oriented programming. *Technical Report SPL97-009
            P9710044*, Palo Alto, CA, USA, February 1997.

[MMG⁺00]   JOSE E. MOREIRA, SAMUEL P. MIDKIFF, MANISH GUPTA, PEDRO V.
            ARTIGAS, MARC SNIR, AND RICHARD D. LAWRENCE. Java pro-
            gramming for high-performance numerical computing. *IBM Systems
            Journal*, 39(1):21–, 2000.

[Mon05]   ALAN MONNOX. *Rapid J2EE Development: An Adaptive Foundation for Enterprise Applications*. Prentice-Hall, 2005.

[Muc97]   STEVEN S. MUCHNICK. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[MWB+01]   GAIL C. MURPHY, ROBERT J. WALKER, ELISA L. A. BANIASSAD, MARTIN P. ROBILLARD, ALBERT LAI, AND MIK A. KERSTEN. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, 2001.

[NEF01]   PANITI NETINANT, TZILLA ELRAD, AND MOHAMED E. FAYAD. A layered approach to building open aspect-oriented systems: a framework for the design of on-demand system demodularization. *Communications of the ACM*, 44(10):83–85, 2001.

[OMB05]   KLAUS OSTERMANN, MIRA MEZINI, AND CHRISTOPH BOCKISCH. Expressive pointcuts for increased modularity. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP'05), Lecture Notes in Computer Science*, volume 3586, pages 214–240, 2005.

[OT01]   HAROLD OSSHER AND PERI TARR. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44(10):43–50, 2001.

[OW99]   SCOTT OAKS AND HENRY WONG. *Java Threads*. O'Reilly, 2nd edition, 1999.

[Pan98]   JAVA GRANDE FORUM PANEL. Making Java work for high-end computing. *Technical report*, Java Grande Forum, Orlando, Florida, November 1998.

[PC01]   J. ANDRÉS DÍAZ PACE AND MARCELO R. CAMPO. Analyzing the role of aspects in software design. *Communications of the ACM*, 44(10):66–73, 2001.

[PRS04]   RENAUD PAWLAK, JEAN-PHILIPPE RETAILLÉ, AND LIONEL SEINTURIER. *La programmation orientée aspect pour J2EE*. 2004.

[PSR05] RENAUD PAWLAK, LIONEL SEINTURIER, AND JEAN-PHILIPPE RE-
TAILLÉ. *Foundations of AOP for J2EE Development*. Apress, 2005.

[PTVF93] W.H. PRESS, S.A. TEUKOLSKY, W.T. VETTERLING, AND B.P. FLAN-
NERY. *Numerical Recipes in C*. Cambridge University Press, Cam-
bridge, UK, 1993.

[PWBK05] DAVID J. PEARCE, MATTHEW WEBSTER, ROBERT BERRY, AND
PAUL H.J. KELLY. Profiling with AspectJ. *Submitted to Software–
Practice and Experience*, 2005.

[RH98] RAJA VALLEE-RAI AND LAURIE J. HENDREN. Jimple: Simplifying
Java bytecode for analyses and transformations. *Technical report*,
Sable Group, McGill University, Montreal, Canada, July 1998.

[Rho03] TOBIAS WINDELN (RHO). LogicAJ - eine Erweiterung von AspectJ
um logische Meta-Programmierung. Diploma thesis, Computer Sci-
ence Department III, University of Bonn, Germany, Aug 2003.

[Rho04] TOBIAS WINDELN (RHO). Static safety for generic aspect languages.
In *Student Research Extravaganza. Poster Session. International Con-
ference on Aspect-Oriented Software Development (AOSD'04), 2004,
Lancaster*. March 2004.

[RK04] TOBIAS RHO AND GÜNTER KNIESEL. Uniform genericity for as-
pect languages. *Technical report*, Computer Science Department III,
University of Bonn., December 2004.

[RS05] HRIDESH RAJAN AND KEVIN SULLIVAN. Aspect language features
for concern coverage profiling. In *AOSD '05: Proceedings of the
4th international conference on Aspect-oriented software development*,
pages 181–191, New York, NY, USA, 2005. ACM Press.

[SB01] L. A. SMITH AND J. M. BULL. A multithreaded Java Grande bench-
mark suite. In *Proceedings of the Third Workshop on Java for High
Performance Computing*, June 2001.

[SBO01] L. A. SMITH, J. M. BULL, AND J. OBDRZALEK. A parallel Java Grande
benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference
on Supercomputing (SC 2001)*, pages 97–105, November 2001.

[SGP02]   OLAF SPINCZYK, ANDREAS GAL, AND WOLFGANG SCHRÖDER-PREIKSCHAT. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, Australia, 2002. Australian Computer Society, Inc.

[SGS+05]   KEVIN SULLIVAN, WILLIAM G. GRISWOLD, YUANYUAN SONG, YUANFANG CAI, MACNEIL SHONLE, NISHIT TEWARI, AND HRIDESH RAJAN. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 166–175, New York, NY, USA, 2005. ACM Press.

[SOK+04]   TOSHIO SUGANUMA, TAKESHI OGASAWARA, KIYOKUNI KAWACHIYA, MIKIO TAKEUCHI, KAZUAKI ISHIZAKI, AKIRA KOSEKI, TATSUSHI INAGAKI, TOSHIAKI YASUE, MOTOHIRO KAWAHITO, TAMIYA ONODERA, HIDEAKI KOMATSU, AND TOSHIO NAKATANI. Evolution of a Java just-in-time compiler for IA-32 platforms. *IBM Journal of Research and Development*, 48(5/6):767–795, 2004.

[SOW+95]   MARC SNIR, STEVE W. OTTO, DAVID W. WALKER, JACK DONGARRA, AND STEVEN HUSS-LEDERMAN. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.

[SR02]   STANLEY M. SUTTON AND ISABELLE ROUVELLOU. Modeling of software concerns in Cosmos. In *Proceedings of the 1st international conference on Aspect-oriented software development*, pages 127–133. ACM Press, 2002.

[SSTP02]   TODD SMITH, SURESH SRINIVAS, PHILIPP TOMSICH, AND JINPYO PARK. Experiences with retargeting the Java HotSpot(tm) virtual machine. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'02)*, pages 119–127, 2002.

[Sul01]   GREGORY T. SULLIVAN. Aspect-oriented programming using reflection and metaobject protocols. *Communications of the ACM*, 44(10):95–97, 2001.

[Thi02]   G. K. THIRUVATHUKAL. Java at middle age: enabling Java for compu-
tational science. *Computing in Science & Engineering [see also IEEE
Computational Science and Engineering]*, 4(1):74–84, 2002.

[Tol]   ROBERT TOLKSDORF. *Programming Languages for the Java Virtual
Machine*. `http://www.robert-tolksdorf.de/vmlanguages`.