Chapter 1

Introduction

This dissertation shows that it is both desirable and feasible to *distribute object-oriented programs automatically*, and that *static analysis* of a program's source code is an indispensable means to that end. To demonstrate this, we have built a system named *Pangaea*, which analyzes the source code of *Java* programs, and distributes them automatically using *arbitrary middleware platforms* as a back-end. At run-time, Pangaea uses a *generic migration subsystem* that can monitor the interactions between selected objects and may migrate them automatically to reduce network communication, thus complementing the static placement decisions.

The name *Pangaea* (greek: "the entire earth") derives from the ancient continent that was postulated by Alfred Wegener (1915). The entire land mass of the earth was centralized in this super-continent until about 200 million years ago, when it broke up and drifted apart to create the distributed world we know today.

The Pangaea system is targeted both towards inherently distributed programs, such as interactive client/server applications on the Internet, and towards concurrent programs to be run in parallel on loosely coupled machines such as clusters of workstations. For both categories of programs, the distribution aspect considerably increases the complexity of programming. Automatic distribution, as performed in Pangaea, is a means to reduce this complexity. It allows programmers to write programs in a centralized version first, without any regard to distribution issues. After the program has been completed and tested on a single machine, Pangaea can automatically create a distributed version of it, following both automatic optimization decisions and constraints specified by the programmer.

Pangaea does not modify the execution logic of a program. It does therefore not attempt to "parallelize" sequential algorithms. If parallelism is desired, we leave it to the programmer to formulate the algorithm in a concurrent way, using the standard model of threads and objects of the Java language. It is the distribution aspect that Pangaea handles automatically, e.g. by placing objects and using remote communication mechanisms to execute the program on a distributed set of machines. Likewise, Pangaea cannot convert a centralized program into a client/server architecture where multiple clients can access a single or multiple servers. Leaving the execution logic unchanged, Pangaea can only split a program into two or more parts, for example a GUI front-end and a database back-end. With some extensions to our system, it would however be possible for a programmer to provide the execution logic of a multi-client program, and let Pangaea again handle the distribution aspect.

This thesis makes three main research contributions. The first is a set of static analysis algorithms that allow our system to estimate a program's run-time structure and dynamic behavior. Based on this information, abstract distribution strategies for the program can be selected and specified, which is done partly by the programmer, and partly by automatic tools that assist him.

The second contribution is that we show how the abstract distribution strategy can automatically be implemented on a given middleware platform. Pangaea achieves this by re-generating the source code of the centralized program, changing or adding whatever is necessary to express the distribution strategy that was specified during analysis. We can thus consider Pangaea a *distributing compiler*: the input language of this compiler is Java, and the target language is the particular Java dialect or configuration language required by the middleware platform to be used. The third contribution is that we show how a generic run-time system can monitor interactions between objects, and re-adjust their placement automatically. Our implementation and case studies allow us to evaluate the cost of such a migration system, suggesting to which extent it is useful and necessary to complement static analysis.

Terminology

A *distributed program* is a program that runs on two or more separate computers, where *separate* means that these computers do not share memory. As a special case, this means that a program that runs in multiple, separate address spaces on a single CPU may also be considered a distributed program. Normally however, the term refers to applications running on loosely coupled machines connected by a network that has much higher latency and lower throughput than a single machine's internal bus system.

When referring to the machines that execute a distributed program, we will use the terms *node*, *host*, *processor*, *CPU*, and *machine* interchangeably. Each of these terms carries some additional flavours of meaning that justify its use in a given context. The technical term used in Pangaea for the different parts of a distributed program is *partition*, although we will sometimes replace it with one of the above terms for clarity. Nothing in this usage implies a difference from the above definition of a distributed program.

A program that is not distributed is called a *centralized* program, and the process of turning a given, centralized program into a distributed program (preserving its execution logic and semantics), is called *to distribute* the program. There are two sharply distinguished reasons why we would want a program to be distributed:

- to enable parallel execution of a *concurrent algorithm*,
- to satisfy the constraints of an *inherently distributed setting.*

A *concurrent program* is a program that is formulated as a set of tasks that may run independent of each other. Such a program can either be executed on a single processor, e.g. by time-slicing, or it can be executed *in parallel* on several processors, which may result in a

	Sequential	Concurrent
Not inherently distributed	—	Concurrent Computation
Inherently distributed	Client/Server Program	Mixed

Table 1.1: Categories of Programs

speedup compared to single-processor execution. (We are following the definitions in Briot et al. (1998) here.)

An *inherently distributed application* is a program that, in order to fulfill its functional purpose, must run on two or more computers that are spatially separated. We call such an environment an *inherently distributed setting.* As an example, consider a banking system where a central database computer is connected to a set of remote ATM terminals, each of which is equipped with a local processor. The software for such a system can be regarded as a distributed program, where certain parts reside in each ATM terminal, and others in the central database computer. Another example for an inherently distributed setting is the Internet, and as a typical example application, we could think of a graphical interface to a database, where the interface is downloaded as a Java applet while the code that accesses the database runs on a central server host.

Both distribution incentives, whether we want to exploit concurrency or adapt to inherent distribution, are orthogonal to another, resulting in a matrix of program categories, as shown in table 1.1. For each of these categories, there are different optimization criteria and means to achieve these criteria. This is most clearly visible when comparing the two extremes, a sequential client / server program, and a concurrent computation without any inherent distribution constraints (see table 1.2).

	Program Type	
	Sequential Client/Server Program	Concurrent Computation
Reason for distribution	adapt to environment	parallel execution
Distributed performance	slower than centralized (tolerated)	faster than centralized (desired)
Primary optimization criterion	minimize distribution penalty	speedup
How to optimize	minimize network communication	load distribution; co-locate activities and objects
Secondary criteria	resources; downloading time; security	—

Table 1.2: Incentives for Distribution

A concurrent computation that is not inherently distributed is usually an input/output-type program with little or no interactive facilities. The reason to distribute such programs is to reduce their execution time, which ideally should be inversely proportional to the number of machines used. To what extent this goal is achieved depends on the nature of the computation, the amount of concurrency inherent in the program, the distribution technology used, and the way the components of the program are actually distributed. With respect to the latter, the optimization criterion is that activities should be placed onto distinct processors, and the data that each activity operates on be co-located on the same processor (if possible), so that costly network communication is minimized.

For inherently distributed client/server type programs, distribution is not a means to an end as it is for concurrent computations; it is a *constraint* that must be met. Because such programs are usually sequential, this constraint necessarily implies a performance penalty, so that the distributed program does not run faster, but *slower* than a centralized version of it. The goal when deciding how to distribute such a program is thus to draw the distribution boundaries so that the penalty is minimized.

This thesis focuses on the distribution of *object-oriented* programs. We follow the classical definition of Peter Wegner (1987), where object-orientation is a property of programming languages involving (a) objects, (b) classes, and (c) inheritance (polymorphism). The technical part of this thesis focuses on the Java language, although our results are applicable to any object-oriented language, and they specifically include implications that would be useful in the design of *future* object-oriented languages. When referring to object-oriented constructs, we will use Java terminology as defined in Gosling et al. (2000).

Making programs run in a distributed fashion requires some sort of *distribution technology* that is often separate from the standard compiler and run-time system of a programming language. We refer to this technology as *distribution platforms*. The term *middleware* is a loosely defined concept that refers to some particular kinds of distribution platforms which Pangaea particularly focuses on; a detailed explanation will be given in chapter 2.

The term *distributed object system* refers to distribution technology for object-oriented programs. This includes object-oriented distribution platforms, but also systems that define a special, object-oriented programming language that cannot be separated from the underlying technology. Distributed object systems also include systems where the standard compiler and run-time system are completely replaced with distribution-enabled versions. For simplicity and historical reasons, we will also use the term distributed object system for systems that are not strictly object-oriented according to the Wegner definition, but can be seen as predecessors of object-oriented distribution technology.

Since distributed programming is inherently more complex than centralized programming, an important goal of distribution technology, and distributed object systems in particular, is to *hide* the distribution aspect from the programmer and to make distributed programming similar, if not identical to centralized programming. This is commonly referred to as *distribution transparency*. There is a classical definition of distribution transparency in the Reference Model for *Open Distributed Processing* (RM-ODP, 1995–1998), which breaks it down into the subcategories *access transparency*, *location transparency*, *migration transparency*, and *replication transparency*. I consider these definitions of little practical use, because they are at the same time too broad and too coarse. For example, almost no distributed object system actually satisfies even the requirements for *access transparency*, since remote object invocations do not have the same semantics as local invocations. In this thesis, we will therefore use the term distribution transparency in a more pragmatic sense, indicating *how similar* a technology makes distributed and centralized programming. We will thus speak of higher vs. lower *degrees* of distribution transparency, and spell out individual transparency deficits in detail.

Road Map

This dissertation is organized as follows.

In chapter 2, we describe the evolutionary progress in the development of distributed object systems over the past decades. This chapter is not a classical "related work" section, because rather than describing projects that do something similar as our own, we portray the technological development that ultimately leads to a system like ours being useful. Pangaea is not a distributed object system itself, it rather utilizes existing distribution technology to distribute programs. Our system also uses ideas from several other areas than those indicated in chapter 2, and we will refer to them where they are actually mentioned.

Chapter 3 describes the Pangaea system itself, following the progression of its subsystems from analysis to implementation and execution. The description is illustrated with numerous small examples for clarification.

In chapter 4, we report on several individual case studies, distributing complete programs with Pangaea and evaluating their performance.

Chapter 5 concludes this thesis and summarizes our project, as well as pointing out some implications for future design of programming languages and distribution technology.

Many concepts in this thesis are closely related and referred to in several different places. The index at the end should therefore provide a useful tool for locating definitions (usually the first mentioning of a concept) and any related aspects.