Chapter 5

Conclusions

This chapter serves two purposes. We will summarize and critically evaluate the achievements of the Pangaea project in section 5.1. Based on this, we will then open up our perspective and draw some general conclusions concerning distribution transparency and the design of programming languages in section 5.2.

5.1 Project Summary

The goal of the Pangaea project, as we stated it in the introduction, was to show that

> *...it is both desirable and feasible to distribute object-oriented programs automatically, and static analysis of a program's source code is an indispensable means to that end.*

In our review of the evolution of distributed object systems in chapter 2, we observed a general trend towards higher degrees of distribution transparency, which is only hampered by the conservative nature of industry standards, and the fact that mainstream languages are usually not designed with distribution in mind. We have dismissed the general and "philosophical" arguments against distribution transparency in section 2.3.1, and may therefore conclude that distribution transparency is *desirable*, and is being demonstrated as *feasible* in ever higher degrees.

We consider *automatic* distribution a natural step beyond *transparent* distribution. The term *distribution transparency* commonly refers to hiding the distribution aspect from the source code of a program, yet it is unspecified *how* and *where* distribution is then decided upon and implemented. As we have seen, many systems ultimately do leave this task to the programmer, e.g. by leaving *essential* distribution aspects, such as the placement of objects, visible in the source code, or by externalizing them into a special configuration language. Our term *automatic distribution* stresses the fact that ultimately, every aspect of distribution should be subject to as high a degree of automation as possible.

There are in fact systems that follow this idea, namely the *implicit* distribution platforms that we described in section 2.2. Due to their high degree of abstraction, they do however suffer from efficiency problems. *Program analysis*, both at compile-time and at run-time, is needed to remedy this, and is in fact used in increasing degrees in these projects.

The Pangaea system, which we described in chapter 3, follows a similar approach that is however different in two key aspects:

- We introduce a novel approach to static, compile-time analysis of object-oriented programs, which provides a macroscopic view of an entire program in object-oriented terms. This

perspective on a program lends itself well to automatic and manual finding of distribution strategies, and includes information which is critical for distribution-related optimizations, and is impossible to determine dynamically at run-time.

- We use existing, *explicit* distribution technology as a back-end, resulting in finer control over communications patterns within a program. Due to its front-end/back-end architecture, Pangaea can thus be considered a *distributing compiler*.

We will now look at the results obtained in the individual parts of Pangaea in turn.

The *object graph analysis* algorithm that we describe in section 3.1, due to its object-oriented perspective, provides an excellent tool for program understanding in its own right. The two distribution-related optimizations that it allows are *scope analysis*, which helps to decide which objects need to be remotely invokable, or monitored by the run-time system, and *immutability analysis*, which allows to copy objects in a distributed setting without affecting the program's semantics.

We have validated our algorithm on programs of up to 10,000 lines of code, and obtained satisfactory performance. The weak side of it is however that it will not scale to programs that are orders of magnitude larger. This is due both to the cubic complexity of the algorithm itself, and the fact that the resulting graphs, even when they are ultimately obtained, will become large and unwieldy. What is needed are techniques for *modularizing* the analysis. One approach would be to introduce subgraphs for subsystems, and annotate them with data-flow information on a subsystem level. This information could be obtained in independent runs of the algorithm, and stored for later use, so that ultimately the analyses of individual subsystems could be combined efficiently.

The object graph lends itself well to finding and expressing distribution strategies for a program. The area of *automatic strategy-finding* is however the one that is least explored in our current system. The use of graph analysis algorithms, in particular graph partitioning, is a promising area for future research. The current system does however implement an automatic graph assignment scheme for concurrent programs that was found effective and sufficient in two of our case studies.

Constructing Pangaea's *back-end implementation layer* led to a critical evaluation of the capabilities of the supported distribution platforms. *CORBA* was found to be a heavy-weight technology that requires many distribution-related source code transformations. An elaborate infrastructure had to be created in Pangaea's run-time system to allow, for example, remote object creation. *JavaParty* is much easier to use from a programmer's point of view, and thus puts much less work on Pangaea's back-end adapter. However, JavaParty's "remote" classes do impose a significant divergence from local Java semantics, which is difficult to work around given the expressiveness of the system (e.g. local objects cannot be passed by reference to remotely invokable objects). Furthermore, JavaParty's communication substrate does not fully optimize local invocations of remotely invokable objects, which renders object migration almost useless. (On the other hand, the different semantics of "remote" classes give JavaParty hardly any choice to do otherwise.) *Doorastha* is the only platform that does optimize local invocations completely, and gives the programmer enough control over parameter passing semantics to at least mitigate the semantic problems that arise from this. Additionally, although Doorastha only has weak object migration (like JavaParty), it is the only platform on which migration is "sufficiently strong" to work well with Pangaea's asynchronous migration subsystem.

Pangaea's code generators demonstrate the feasibility of compiling a program for a particular distribution platform, using a front-end/back-end architecture similar to that of a classical compiler. Several semantics-preserving transformations are carried out successfully, for example

wrapping arrays into objects to make them remotely invokable or migratable. The *Visitor*-based code generating mechanism has however been stretched to its utmost limits to achieve this. More far-reaching source code transformations, for example to "correct" JavaParty's remote invocation semantics completely, would only be possible if the abstract syntax trees of a program could be modified directly.

Nonetheless, the array transformation in particular turned out to be surprisingly inexpensive at run-time. This is due to the fact that array accesses are not very efficient in Java to begin with; an additional indirection via method invocation does therefore not cause a significant slowdown.

Finally, the *asynchronous migration subsystem* is essential for adapting otherwise static placement decisions at run-time. It features a generic design in two dimensions: First, it is generic in its migration strategy, which can be configured by plugging in arbitrary "strategy" objects. Second, the mechanism by which migration is triggered on a given platform is encapsulated and can easily be replaced. The price that is paid for using the migration subsystem is its book-keeping overhead, which our case studies have shown to be significant. Here, optimizations would be most worthwhile (and conceivable). Yet, our experience seems to indicate that such a system must only be used selectively on carefully chosen objects; it cannot establish a distribution strategy for an entire program. Asynchronous object migration is thus a complement to static analysis, not a replacement.

5.2 Implications for Language Design

At several points in this thesis, we have argued that the case of distributed execution needs to be considered during language design from the very beginning. In particular, every effort should be made to ensure that *the semantics of local and remote invocations are the same.* We have seen that many systems, in particular those based on existing main stream languages, make compromises in this regard that seem small and reasonable at first sight: passing serializable objects, including arrays, by value instead of by reference, etc. The reasoning goes that the added efficiency (a cheap alternative to true object migration) is well worth the semantic inconsistency, and that programmers can easily adapt to it.

It has already been pointed out in previous work (Brose et al. 1997, Löhr 2002) that it might not be as simple as that. In fact, situations can be constructed and are likely to occur in practice where invocation semantics become *unpredictable* as a result, since it cannot be known whether an object is invoked directly or via a proxy. Our current work reinforces this point by adding another reason: as soon as true, perhaps asynchronous, object migration is introduced, it is essential to fully optimize the local case. An object that is moved to another address space for efficiency reasons *must* be invoked directly in that address space for the migration to pay off. Otherwise, if the proxy is kept, at most network latency can be eliminated, and this still leaves invocations orders of magnitude slower than true local ones (see our performance study of JavaParty vs. Doorastha on page 100). If however remote semantics differ from local semantics, then there is compelling reason to *keep* the proxy even as an object is moved into the local address space, simply to "simulate" the remote invocation semantics.

We thus have a contradiction that can only be solved by taking distribution transparency more seriously and unifying the semantics of local and remote invocations. The problem is perhaps underlined, if not reinforced by the common misnomer of "remote", or even *"remoted,"* objects or types. Inherent in this is the misunderstanding that "remoteness" were a property of an object, when in fact it is a *relation* between two objects A and B. The corresponding property of an object is to be "remotely invokable" (it is perhaps best to call such objects "global objects"), and another property of them is that they ought to be *locally invokable* as well,

and in an efficient manner. Ultimately, this can only be achieved by incorporating distribution considerations into language design, rather than to add distribution technology merely as a kind of additional "API".

Another area that requires careful thought is *static binding to entities* in object-oriented languages, realized for example by *static type members* in Java. (This is related to, but distinct from static vs. dynamic binding of individual method calls.) The concept is somewhat alien to a pure object-oriented approach, which is why purists often eschew it. Eiffel, for example, goes to great lengths trying to avoid static binding, and yet has to introduce the construct of "once" functions to finally achieve it, for example to bind to external I/O facilities.

Static type members in Java are a much more pragmatic approach to solve the same problem, yet it needs to be rethought in the context of distribution. Should static members be unique per host, or unique within the entire system? (See the "exercise in staticness" we had to solve when setting up Pangaea's run-time system on different platforms, page 93.) Some confusion here seems to result from the fact that Java closely associates static members with the classes that define them. Classes, containing the code of instances, need to be replicated on each node where instances should reside. This is however not necessarily true for their static members.

Pangaea's approach is to separate the two concepts more thoroughly, by splitting a Java class into a static type (comprising the static members) and a dynamic type (comprising the instance members). It is thus possible to specify distribution strategies orthogonally for each of these types. JavaParty does something similar by always allocating the static members of "remote" classes in a separate entity on a single host, without however allowing any further control. Doorastha allows the programmer to specify an individual policy for each static member. A greater similarity in this area is desirable, which reinforces our suggestion to consider distribution already at language design time.

Pangaea identifies and exploits *immutable objects* for efficient distribution; other systems such as cJVM do this as well. Still, it must be admitted that many programmers do not usually *make* their objects immutable, if only because they are not aware of the potential for optimization in the distributed case. Language designers could help here by making it *easy* and perhaps *rewarding* to define immutable types. For example, a class modifier could be introduced to declare a class as immutable, and the compiler could statically check it. Vice versa, classes could be immutable by default and an additional modifier be required to declare them as *mutable* — although we don't believe that this will necessarily have an effect on programmers' habits. (*Final* members and *final* classes are a similar case; the potential for optimization is often left unused.) Ultimately, this could perhaps only be changed by defining languages so that types will *naturally* come out as immutable unless the programmer has an explicit desire to do otherwise, similar to the way languages from a functional background make it at least *hard* for the programmer to produce side-effects.

A similar argument can be made for *scope analysis*, Pangaea's second large area of optimization. Declaring a reference field to be private, or using a local variable, does not imply anything about where the reference stored in that field may come from, or where it may eventually end up in the system. Static analysis algorithms, whether they are used for distribution purposes or not, essentially face the difficult task of tracking the flow of references within the program. It would be worthwhile if programming languages had a means of expressing and restricting the scope of references, not just of names. *Confined types*, as suggested by Vitek and Bokowski (2001) are a step in this direction. Still, programmers can only be expected to embrace such facilities if the language makes it *easy*, *rewarding*, or even *inevitable* to use them.