CHAPTER VII

RELATED WORK

The objective of this chapter is to put the research described in this dissertation into perspective by showing how it relates to existing work. First, we discuss directly related work as pertaining to each of the three software tools explored by this dissertation. Then we identify how this work on separating distribution concerns utilizes approaches and techniques from different research areas. Finally, we outline how this work could benefit or influence various areas of research and practice.

7.1 Directly Related Work

Because this dissertation takes a three-pronged approach to separating distribution concerns, consisting of middleware with copy-restore semantics, a program generator for distribution, and an automatic partitioning system, we similarly discuss directly related work as pertaining to each of these software tools.

7.1.1 NRMI

7.1.1.1 Performance Improvement Work

Several efforts aimed at providing a more efficient implementation of the facilities offered by standard RMI [80]. Krishnaswamy et al. [45] achieve RMI performance improvements by replacing TCP with UDP and by utilizing object caching. Upon receiving a remote call, a remote object is transferred to and cached on the caller site. In order for the runtime to implement a consistency protocol, the programmer must identify whether a remote method is read-only (e.g., will only read the object state) or not, by including the throwing of "read" or "write" exceptions. That is, instead of transferring the data to a read-only remote method, the server object is moved to the data instead, which results in better performance in some cases.

Several systems improve the performance of RMI by using a more efficient serialization mechanism. KaRMI [65] uses a serialization implementation based on explicit routines for writing and reading instance variables along with more efficient buffer management.

Maassen et al.'s work [53][54] takes an alternative approach by using native code compilation to support compile and run time generation of marshalling code. It is interesting to observe that most of the optimizations aimed at improving the performance of the standard RMI and call-by-copy can be successfully applied to NRMI and call-by-copy-restore. Furthermore, such optimizations would be even more beneficial to NRMI due to its heavier use of serialization and networking.

7.1.1.2 Usability Improvement Work

Thiruvathukal et al. [85] propose an alternative approach to implementing a remote procedure call mechanism call for Java based on reflection. The proposed approach employs the reflective capabilities of the Java languages to invoke methods remotely. This simplifies the programming model since a class does not have to be declared `Remote` for its instances to receive remote calls.

While CORBA does not currently support object serialization, the OMG has been reviewing the possibilities of making such support available in some future version of IIOP [62]. If object serialization becomes standardized, both call-by-copy and call-by-copy-restore can be implemented enabling [in] and [in out] parameters passing semantics for objects.

The systems research literature identifies Distributed Shared Memory (DSM) systems as a primary research direction aimed at making distributed computing easier. Section 7.1.3 discusses DSM systems in greater detail. However, in comparison with NRMI, DSM systems can be viewed as sophisticated implementations of call-by-reference semantics, to be contrasted with the naive “remote pointer” approach shown in Figure 2-3 on page 18. On the other hand, the focus of DSM systems is very different from that of middleware. DSMs are a great enabling technology for distributed computing as a means to achieve parallelism. Thus, they have concentrated on providing correct and efficient semantics for multi-threaded execution. To achieve performance, DSM systems create complex memory consistency models and require the programmer to implicitly specify the sharing properties of data. In contrast, NRMI attempts to support natural semantics to straightforward middleware, which is always under the control of the programmer.

NRMI (and other mainstream distribution middleware systems) do not try to support “distribution for parallelism” but instead facilitate distributed computing in the case in which an application's data and input are naturally far away from the computation that needs them.

A special kind of tools that attempt to bridge the gap between DSMs and middleware are automatic partitioning tools such as our own J-Orchestra. (We discuss other automatic partitioning systems in Section 7.1.3.) Such tools split centralized programs into several distinct parts that can run on different network sites. Thus, automatic partitioning systems try to offer DSM-like behavior but with more ease of use and compatibility: Automatically partitioned applications run on existing infrastructure (e.g., DCOM or regular unmodified JVMs) but relieve the programmer from the burden of dealing with the idiosyncrasies of various middleware mechanisms.

The JavaParty system [31][65] works much like an automatic partitioning tool, but gives a little more programmatic control to the user. JavaParty is designed to ease distributed cluster programming in Java. It extends the Java language with the keyword `remote` to mark those classes that can be called remotely. The JavaParty compiler then generates the required RMI code to enable remote access. Compared to NRMI, JavaParty is much closer to a DSM system, as it incurs similar overheads and employs similar mechanisms for exploiting locality.

Doorastha [19] represents another piece of work on making distributed programming more natural. Doorastha allows the user to annotate a centralized program to turn it into a distributed application. Although Doorastha allows fine-grained control without needing to write complex serialization routines, the choice of remote calling semantics is limited to call-by-copy and call-by-reference implemented through RMI remote pointers or object mobility. Call-by-copy-restore can be introduced orthogonally in a framework like Doorastha. In practice, we expect that call-by-copy-restore will often be sufficient instead of the costlier, DSM-like call-by-reference semantics.

Finally, we should mention that approaches that hide the fact that a network is present have often been criticized (e.g., see the well-known Waldo et al. “manifesto” on the subject [96]). The main point of criticism has been that distributed systems fundamentally differ from centralized systems because of the possibility of partial failure, which needs to be handled differently for each application. The “network transparency” offered by NRMI does not violate this principle in any way. Identically to regular RMI, NRMI remote methods throw remote exceptions that the programmer is responsible for catching. Thus, programmers are always aware of the network's existence, but with NRMI they often do not need to program differently, except to concentrate on the important parts of distributed computing such as handling partial failure.

7.1.2 GOTECH

Directly related work for GOTECH includes other tools that help in adding distribution, but without taking away from the programmer the control and responsibility of the distribution process. Such tools are “distributed programming aids”: they help do the tedious tasks that the programmer would otherwise need to do manually and that would “pollute” the code describing the application logic. Nevertheless, the programmer is still responsible for ensuring that the tools do the right job for the application at hand.

Indirectly related work includes mostly application partitioning tools and Distributed Shared Memory systems. Such tools offer a higher-level interface. Their user does not necessarily program the distributed application, but rather offers hints to improve its performance. These tools have a higher correctness responsibility: they attempt to correctly distribute any application although they usually result in loss of efficiency and are applicable in fewer situations than the “distributed programming aids”.

Many domain-specific languages have been proposed to aid distributed programming, and some of them [40][52] were key examples in the early steps of Aspect Oriented Programming. Such domain-specific languages for distribution are described in detail later, in Section 7.1.3, but we can make general observations regarding the GOTECH framework's advantages:

- it is an easy to evolve tool, based on widely used aspect-oriented infrastructure (AspectJ and XDoclet). Inspecting and changing the functionality of our XDoclet templates is much easier than changing the code for any of the above domain-specific tools.
- it employs NRMI as a unique way to support a remote call semantics that is closer to local execution. NRMI is applicable to many common scenarios, eliminating the need for explicitly updating data when changes are introduced by remote calls.
- GOTECH targets EJBs as a distribution substrate. This is a more complex, industrial-strength technology than the middleware used by previous systems.

Both DSMs and partitioning systems operate at a much higher level than tools like GOTECH. They strive for correct distributed execution of all applications and give the programmer much less control over distribution choices. Therefore, similarly to very high-level languages, these tools are valuable for the cases for which they are applicable, but these cases are a small part of the general distributed computing landscape. In contrast, GOTECH assists the programer in generating tedious code that would otherwise be intertwined with the application logic.

Finally, other researchers have examined the suitability of aspect-oriented techniques for different domains. For example, Kienzle and Guerraoui [43] examined the suitability of aspect-oriented tools for separating transaction logic from application logic. Separating transaction processing from application logic is very hard, and possible only under very strict assumptions about the application. These findings of Kienzle and Guerraoui are consistent with longtime observations of the database community.

7.1.3 J-Orchestra

Much research work is closely related to J-Orchestra, either in terms of goals or in terms of methodologies, and we discuss some of this work next.

Several recent systems other than J-Orchestra can also be classified as automatic partitioning tools. In the Java world, the closest approaches are the Addistant [84] and Pangaea [74][75] systems. The Coign system [33] has promoted the idea of automatic partitioning for applications based on COM components.

All three systems do not address the problem of distribution in the presence of unmodifiable code. Coign is the only one of these systems to have a claim at scalability, but the applications partitioned by Coign consist of independent components to begin with. Coign does not address the hard problems of application partitioning, which have to do with pointers and aliasing: components cannot share data through memory pointers. Such components are deemed non-distributable and are located on the same machine. Practical experience with Coign showed that this is a severe limitation for the only real-world application included in Coign's example set (the Microsoft PhotoDraw program). The overall Coign approach would not be feasible for applications written in a general-purpose language (like Java, C, C#, or C++) in which pointers are prevalent, unless these applications have been developed following a strict component-based implementation methodology.

The Pangaea system [74][75] has very similar goals to J-Orchestra. Pangaea, however, includes no support for making Java system classes remotely accessible. Thus, Pangaea cannot be used for resource-driven distribution, as most real-world resources (e.g., sound, graphics, file system) are hidden behind system code. Pangaea utilizes interesting static analyses to aid partitioning tasks (e.g., object placement) but these analyses ignore unmodifable (system) code.

The JavaParty [31][66] system is closely related to J-Orchestra. The similarity is not so evident in the objectives, since JavaParty only aims to support manual partitioning and does not deal with system classes. However, the implementation techniques of JavaParty are very similar to the ones of J-Orchestra, especially for the newest versions of JavaParty [31]. For distributed synchronization, JavaParty relies on KaRMI, a drop-in replacement for RMI, that maintains correct multithreaded execution over the network efficiently. In contrast, J-Orchestra implements distributed synchronization on top of standard middleware.

J-Orchestra bears similarity with such diverse systems as DIAMONDS [16], FarGo [32] and AdJava [23]. DIAMONDS clusters are similar to J-Orchestra anchored and mobile groups. FarGo groups are similar to J-Orchestra anchored groups. Notably, however, FarGo has focused on grouping classes together and moving them as a group. In fact, groups of J-Orchestra objects that are all anchored by choice could well move, as long as all objects in the group move. We have not yet investigated such mobile groups, however.

The pioneering work at MCC in the early 90s identified classes as suitable entities for performing resource allocation in distributed systems. The experimental system described in reference [15] uses class profiling as a guide for assigning objects to the nodes of a distributed system. J-Orchestra has fully explored the idea of resource-based partitioning at the class or group of classes level of granularity, demonstrating the feasibility and scalability of the approach.

Automatic partitioning is essentially a distributed shared memory (DSM) technique. Nevertheless, automatic partitioning differs from traditional DSMs in several ways. First, automatic partitioning systems such as J-Orchestra do not change the runtime system, but only the application. Traditional DSM systems like Munin [14], Orca [5][6], and, in the Java world, cJVM [3][4], and Java/DSM [102] use a specialized run-time environment in order to detect access to remote data and ensure data consistency. Also, DSMs have usually focused on parallel applications and require programmer intervention to achieve high-performance. In contrast, automatic partitioning concentrates on resource-driven distribution, which introduces a new set of problems (e.g., the problem of distributing around unmodifable system code, as discussed earlier). Among distributed shared memory systems, the ones most closely resembling the J-Orchestra approach are object-based DSMs, like Orca [5][6].

Mobile object systems, like Emerald [11][39] have formed the inspiration for many of the J-Orchestra ideas on object mobility scenarios. The novelty of J-Orchestra is not in the object mobility ideas but in the rewrite that allows them to be applied to an oblivious centralized application.

Both the D [52] and the Doorastha [19] systems allow the user to easily annotate a centralized program to turn it into a distributed application. Although these systems are higher-level than explicit distributed programming, they are significantly lower-level than J-Orchestra. The entire burden is shifted to the programmer to specify which semantics is valid for a specific class (e.g., whether objects are mobile, whether they can be passed by-copy, and so forth). Programming in this way requires complete understanding of the application behavior and can be error-prone: a slight error in an annotation may cause insidious inconsistency errors.

7.2 Related Research Areas

This dissertation explores new software tools for separating distribution concerns, placing the work on the intersection of software engineering, programming languages, and distributed systems. While acknowledging that centralized and distributed programming model are fundamentally different and should be treated as such (e.g., as stated in the well-known "Note" by Waldo et al. [96]), we, nevertheless, recognize that evolving a distributed program by introducing distribution to an existing centralized program has become a common programming task [44]. Thus, this research draws its motivation from the inherent difficulties of the programming task of transforming a centralized, monolithic program into a distributed program.

This research is related to software engineering. First, it identifies the limits of automating the process of introducing distribution to existing centralized programs. Second, it provides better software tools, thereby contributing to improving programmers's productivity. Finally, it explores new software engineering approaches for emerging domains such as ubiquitous computing.

This research is related to programming languages. This research takes advantage of several techniques and approaches that were first utilized to address various challenges in programming language implementation. Specifically, the NRMI algorithm for call-by-copy-restore is influenced by copying garbage collectors [100]. Several implementation facets of the J-Orchestra system (e.g., maintaining the semantics of various language entities in a distributed environment, bridging the local/remote differences in parameters passing semantics, introducing indirection in the presence of unmodifiable code in the runtime system) have commonalities with issues in programming languages research. GOTECH uses code generation, which has branched away from programming languages into an independent research area [50].

This research is related to distributed systems. The Remote Procedure Call (RPC) mechanism [10] remains a popular programming model for building distributed systems, even in the research domain. For example, van Nieuwpoort at al. [59] demonstrate the feasibility of using RMI for building parallel grid applications. This dissertation makes several improvements to the RPC mechanism. Specifically, NRMI is the first RPC system that offers a fully-general call-by-copy-restore semantics for linked data structures, and J-Orchestra introduces a novel technique for maintaining concurrency and synchronization constructs over RPC efficiently. In a broader sense, because this research focuses on synchronous, RPC-enabled remote communication, it is not immediately obvious how this work could advance some of the current state-of-the-art of distributed systems such as peer-to-peer systems, sensor networks, grid computing, autonomic computing, and sophisticated fault-tolerance mechanisms. Nevertheless, the software tools explored by this dissertation can be adapted to work and be beneficial for many experimental distributed systems as well. In other words, by further investing into this work on improving software tools for challenging domains, we can get closer to fulfilling the ambitious objective of addressing the disparity between the advances in high-performance system design and the practices in industrial software development.

7.3 Beneficiaries of This Research

The results of this research can benefit several areas of research and practice. Despite the fact that the primary contributions of this research are in the domain of software technologies for distributed computing, some of the ideas explored by this dissertation are applicable to other domains. We next describe in turn how this research contributes to the areas software engineering, programming languages, and distributed systems.

This research contributes to software engineering by having investigated novel software tools that facilitate challenging programming tasks. This work has demonstrated how a combination of advanced development techniques (such as code generation, code transformation, and bytecode engineering) can assist the programmer in developing complex computer systems in the field of distributed computing. At the same time, this research has developed techniques that can be of value in multiple domains. For example, Bialek at al. [9] have adapted some of the J-Orchestra techniques to partition programs so that they could support dynamic updates. As a sign of its practical value and broader impact, this research has influenced the designers of JBoss [68], the most popular open source application server. According to Marc Fleury, JBoss founder, the bytecode engineering techniques of J-Orchestra have been the inspiration for the Aspect-Oriented Programming features of JBoss 4. JBoss 4 employs bytecode engineering to add various non-functional pieces of functionality to POJOs (Plain Old Java Objects), similarly to J-Orchestra adding distribution to unaware centralized programs. Another potential beneficiary of this research is the domain of ubiquitous computing, which has been recognized as needing better software support [1]. This research has identified automatic partitioning as a promising approach to prototyping ubiquitous computing applications and also pointed out how, in this domain, semi-automatic tools could be beneficial at the development stages beyond prototyping. The GOTECH framework has introduced the approach of combining generative and aspect-oriented techniques to alleviate tedious and error-prone programming tasks, which can be of software engineering value in domains other than distributed computing. As an example, the domain-independent MAJ tool [107] has followed and improved on the GOTECH approach. In addition, because of its ease of use in the presence of complex J2EE conventions, GOTECH has been suggested as a tool for software engineering education. While discussing approaches to teaching concepts of software adaptation in distributed object computing, Gray [26] identifies the GOTECH framework as a possible tool for exposing students to applications of aspect-oriented techniques.

This research contributes to programming languages. The insights of generalizing the J-Orchestra indirection machinery can be applied to designing the runtime libraries of future virtual machines. This research has identified that the presence of unmodifiable code in the runtime system of a bytecode application can significantly hinder what kind of indirection can be safely applied to that application. If one had information on how exactly native code libraries interact among them and with the bytecode of system classes, this would allow more flexibility in designing the user-level indirection machinery. Unfortunately, the straightforward way to get such information is to analyze the source code of the runtime system. This is complicated at best and unrealistic at worst (source code may not be available). The natural avenue for extending program analysis when source code is not available is to employ programmer-supplied annotations. These annotations would form a language for communicating implementation insights. One can draw on the findings of this research to create such an annotations scheme that would reflect how exactly system classes interact with native libraries.

This research contributes to distributed systems. We have already mentioned the contributions to the RPC mechanism, a common paradigm for building distributed systems. In addition, automatic application partitioning provides a software technology answer to several difficult systems problems. For example, state-of-the-art event-delivery systems such as JECho [105][106] derive performance benefits by collocating event processing functionality with event producers. Specifically, JECho introduces a novel software abstraction called eager handlers that send parts of event handling functionality from the consumer to the supplier sites of an event. This effectively partitions event handling into two parts, the first executed by the sender and the second by the receiver. Such partitioning of events processing functionality could limit bandwidth consumption or reduce the computational costs, depending on a given cost model. With automatic application partitioning we can achieve similar benefits for larger-scale applications by placing code near the resource it manages. For example, we can partition a centralized application for distributed execution in such a way that a particular systems resource such as graphics can be produced and filtered on the same network site, while only the filtered version will be transferred over the network to the site on which it will be displayed on a graphical screen. More sophisticated scenarios for collocating resources with the code managing them can be achieved through object mobility.