Automatic Introduction of Mobility for Standard-based Frameworks

Grégory Haïk, Jean-Pierre Briot, Christian Queinnec

Laboratoire d'Informatique de Paris 6
Université Pierre et Marie Curie
4, place Jussieu 75252 Paris cedex 5 FRANCE
{Gregory.Haik | Jean-Pierre.Briot | Christian.Queinnec}@lip6.fr

Abstract. The computerization of industrial design processes raises software engineering problems that are addressed by distributed component frameworks. But these frameworks are constrained by a set of antagonistic constraints, between performances and reusability of the components. In order to take up this challenge, we study how mobile code technology enables the improvement of performances without harming the components' reusability. Our approach relies on a transparent, totally automatic introduction of mobility into the programs. This transformation is a local optimization which is based on a static analysis. It is implemented within a compiler. An experimental study shows how the approach can be helpful for increasing the efficiency of the framework, enabling the usage of standards that – as for today – lack of efficiency.

1 Introduction

System engineering, industrial design and manufacturing have been totally transformed, since 1950, by the raise of computer software, especially Computer-Aided Design (CAD) systems and numerical simulation programs. But users are still waiting for appropriate integration frameworks [11, 5] that would link these programs all toghether. Still today, engineers have a lot of manual and repetitive, work in order to make their software environment adapted to the particular needs of a specific project.

But industrial design integration frameworks are facing difficult problems : on the one hand, they need to enable a good level of reusabililty – hence they need to rely on standardized contracts and interfaces –, and on the other hand, the applications built upon these frameworks need to be efficient. Unfortunately, using standards tends to prduce uneffcient resulting applications, especially in a distributed environment. In order to bridge the gap between performance issues and conformance to standardized interfaces, we have proposed some compilation techniques for automatic introduction of mobility into programs interacting with software components. Mobile code is used as a mean to improve locality, and consequently performances [2]. By raising up the performances of such programs, software architects can rely on standards that are usually considered as unusable because of the poor performances they imply.

In this paper, we present the results of our PhD research for efficient execution of distributed, standard-based integration frameworks, applied to industrial design applications [9]. The next section presents the motivations (section 2), and leads to the description of our approach (section 3). We will then compare our approach to related works (section 4). The following section will describe more deeply the analysis and compilation techniques and present our prototype. We will finally depict the experimental study that shows the tangible benefits of our techniques (section 6).

2 Motivations

This section will first present an example of integration framework for industrial design. Based on this example, we will explain why such frameworks tend to be distributed, and why they should rely on standards. We will conclude by showing that the usage of standards raises efficiency problems.

2.1 An Example of Framework for Intustrial Design : SALOME

SALOME [16] is a ministry-led consortium (RNTL) aimed at defining a component-based framework for integration of software systems involved in industrial design, such as CAD systems, meshing software, numerical solvers, databases of physical properties, visualisation and post-treatment software. Figure 1 shows a snapshot of the user interface of SALOME. In this example, the user has imported the geometry of a ship and meshed it with the help of a meshing component. The user would then typically assign materials to the ship geometry such as, for instance, carbon composite and aluminium ; apply forces to the structure for folding and/or torsion ; load a solver of structural mechanics – embedded in a SALOME component ; and then check by computation whether the ship structure is complying to its requirements. The user could also load a fluid mechanics solver in order to check the ship's structure in conditions of navigation. As we can see, since the usage scenarios of SALOME cannot be predicted, the framework includes a program interpreter (actually a Python interpreter), so that the user can customise the components integration according to his/her specific usage.

2.2 The Need for Distribution

One can wonder why such component-based frameworks are distributed. Indeed, some companies have developed integration frameworks for industrial design in a monolithic, mono-process configuration. However, there are several reasons to make such frameworks distributed. The first reason is the typical size of data and computation timings needed by these applications. If a single-station implementation is feasible for small problems, it becomes unrealistic when the user want to refine the simulation, which leads to bigger problems. Then, distribution can

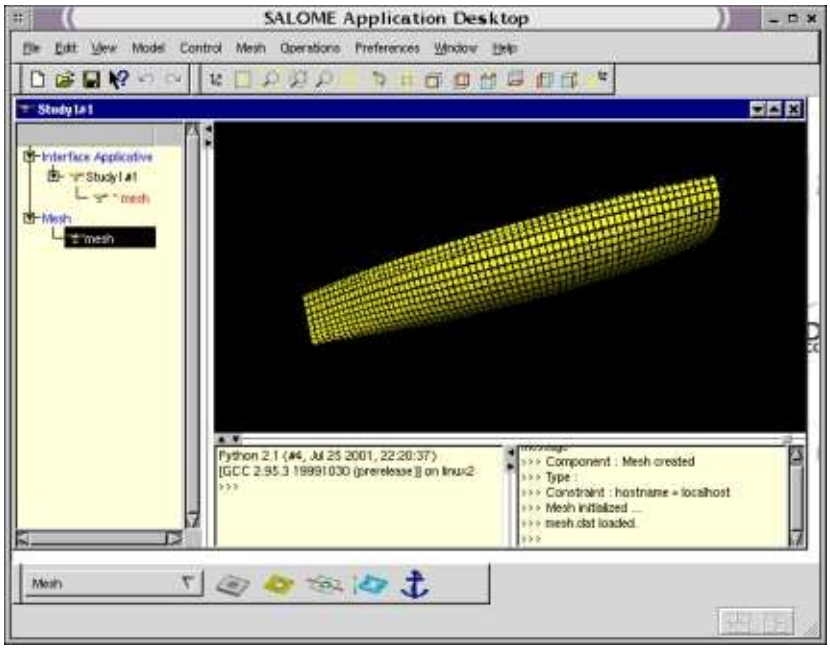

Fig. 1. Meshing of a ship structure in SALOME

help to manage more data (distributed storage), and to speed up computation (parallelism). The second reason is that the software systems encapsulated in components may have constraints on the underlying hardware and/or operating system. For instance, the visualization tool would require a good 3D GPU, a numerical solver could be specifically developed for a particular data parallel architecture, and so on. Then distribution is a mean to easily satisfy a set of constraints expressed by the different components. Finally, such component-based industrial design frameworks are aimed, at least on mid-term, to enable a large scale co-operation between engineers in different, distant locations.

2.3 The Need for Standardization

We will develop here a short remark about the definition of contracts and interfaces between the components, and between a component and the framework. On the first hand, considering the number of different components to be integrated, the number of human actors (users, vendors, integrators...), the number of different usage scenarios, it seems necessary to standardize the interfaces upon which the frameworks are built. Indeed, without this effort, the components are condemned to be incompatible one with the other. On the other hand, all the actors of this application domain are not necessarily enthusiastic regarding standards. There is a blatant lack of confidence in major standardization organizations like OMG. Some people believe that the standards produced by such

organizations are unusable : if they are too exhaustive, then no software component can implement them completely ; if they are too small then they become useless. Although this description is caricaturized, the question raised by the inherent flaws of the standardization processes must be addressed thoroughly. To illustrate this lack of confidence, let us examine the case of CAD Services.

2.4 An Example of Standardization Process : OMG *CAD Services*

In 2001, the OMG has established a working group aimed at creating a set of standard interfaces for accessing CAD systems encapsulated in a software component. The list of contributors proves that this standardization effort meets a real users' need (Boeing, Nasa, GE, Ford, ...). CAD systems vendors, such as 3DS and OpenCascade, have also participated in this effort. The group has worked for two years, and frequent meetings were held all around the world, where harsh discussions and debates took place : there exists a natural antagonism between users and vendors, since users push the standard to be as exhaustive as possible while vendors want it thinner so they can implement it more easily. Moreover, there is another natural antagonism between the different vendors, who try to make the standard ontology as close as possible to their own products ontology.

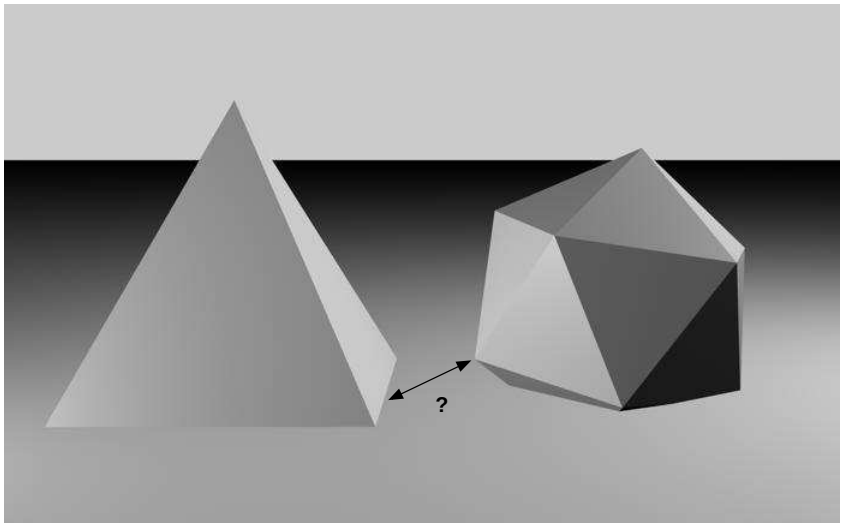

Fig. 2. A Pair of solids hosted by a CAD server

As a result of these technical and logistical impediments, the final result of this effort is disappointing. The working group has only achieved to release a standard – called *CAD Services* [13, 3] – for geometric shapes warehouses. Shapes are defined in terms of very basic and fine-grained concepts such as points, vertices, faces, and solids. Moreover, CAD Services hardly includes any of

the most common algorithms usually applied to these data. Consequently, when users want to manipulate geometrical data hosted by such a CAD server, they need to implement their algorithms on the client side, while fine-grained data is accessed on the server side. In a distributed environment, this is particularly inefficient. For instance, consider a CAD server hosting the two solids shown in Figure 2. Consider that the solids have an elctrical potential difference, and that the user wants to know whether an electric arc can occur between the two solids. For this, he has to compute the minimal distance from one solid to the other. But unfortunately, the standard does not include any operation for this algorithm. As we will see in section 6, we have implemented this example : a client is located in Paris, while the CAD server is in Nice (1000 km, ping round trip time 20 ms). With a simplified algorithm implemented in the client and a pair of very small shapes made of 117 vertices hosted in the server, the computation almost takes 40 minutes.

3 Our Approach

Our solution to this problem is to transform in a completely automated manner the client program so that it sends a piece of mobile code, that we call a *mobilet*, to the server. This transformation is made in an optimizing compiler. The input of the compiler is exactly the same source code than for the example described above. The computation of the physical distance decreases from 40 minutes to just over 2 minutes. The optimized code is about 18 times faster than without our compiling techniques.

The main challenge addressed by our optimizing compiler is to identify the interesting pieces of code to be embedded in a *mobilet* and executed remotely. The result of the transformation is illustrated on Figure 3. The original code performs a big number of remote interactions between the client-side and the server-side components. By executing a *mobilet* containing part of the client code – in a dedicated hosting process located on the server side –, the remote interactions are transformed into local interactions, resulting in a sensible improvement of performances.

4 Related Works

Our approach is a variant of *automatic partitioning*. In a similar way as *automatic parallelization* transparently analyzes and transforms sequential programs in order to discover opportunities for introducing parallelism [12, 1, 6], *automatic partitioning* (or *automatic distribution*) tends to analyze and transform centralized programs in order to discover opportunities for introducing distribution. During the last few years, some research has been conducted in this domain, as reviewed below.

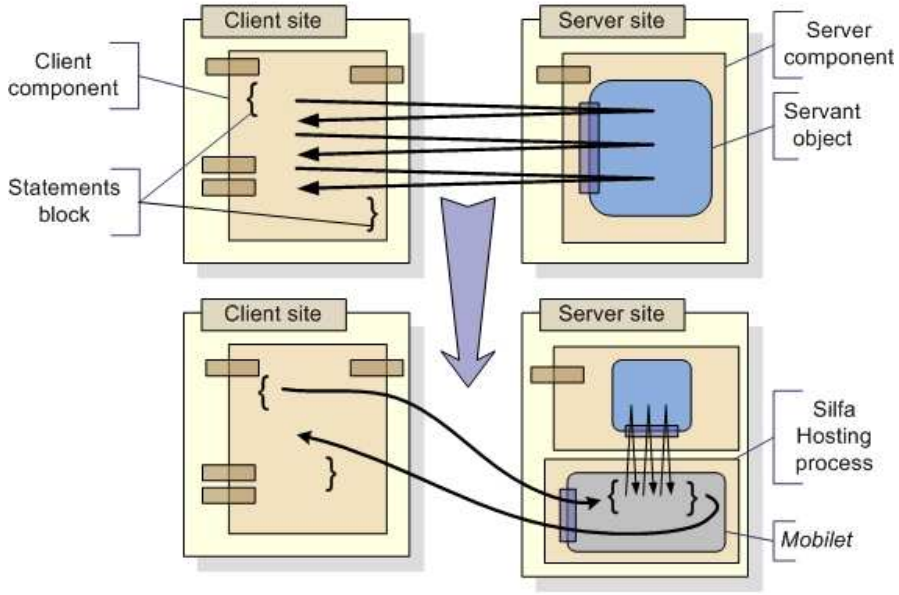

Fig. 3. Transformation of client code by the introduction of a mobilet

JavaParty [14] is an extension of Java that automatically transforms regular Java classes into remotely accessible ones. It also provides migration of these classes' instances. Users specify which objects are to be made remote/mobile by tagging their classes with a new modifier (keyword `remote`). When an object is migrated, it accesses Java API on the host where it is executed : inputs and outputs are taken from and sent to the destination host, which can be interpreted as a problem of correctness of the transformation.

Doorastha [4] is quite similar to JavaParty, but differs from it on the following issues : Java syntax is not modified, and users insert pragmas in Java comments. Thus, compatibility with genuine Java compilers is preserved. Moreover, in Doorastha's object migration model, calls to `System.out` are forwarded to the original JVM's console, which denotes consideration for the problem of correctness. Still, in Doorastha, there is no tracking of *every* such location-dependent primitives of the Java API, which leads to inconsistencies.

Pangaea [17] is a distribution system for Java applications that works with

both JavaParty and Doorastha as back-ends. It is based on a static analysis of Java programs that computes an approximation of the runtime object graph. The Pangaea user specifies, through a graphical user interface, which objects are tied to which hosts. From this specification, the system computes a good placement of every other objects among the anticipated runtime population, by minimizing the number of repeated remote calls.

J-Orchestra [19] and **Addistant** [18] are two automatic partitioning systems for Java bytecode. J-Orchestra distributes Java classes among the network (with the help of the user, like in Pangaea), using statistics gathered by a runtime profiler – in a calibration stage – in order to make placement decisions. The profiler is applied to the non-transformed program in order to evaluate the computational flows between classes. Moreover, classes that contain platform-specific code in native format are considered anchored to their host : they can not be made mobile. A semi-automatic process ensures that no such class will eventually be run on the wrong machine. Regarding to our notion of correctness, J-Orchestra is the only partitioning system that provides a sound mechanism for distributing code.

Coign [10] is a partitioning system for applications made of COM components. It combines typical usage scenarios, application and network profilers in order to make placement decisions, by scrutinizing inter-component communications. As Coign is designed for client-server distribution, it constrains GUI calls to remain on client side, while data storage calls remain on the server side.

Compared to the related works reviewed above, one should notice the three main contributions of our approach. First, our grain of mobility introduction is atomic : our compiler can make mobile every single statement of the original program, while systems reviewed above can only distribute COM components or Java objects. Our fine-grain mobility introduction enables to take advantage of automatic distribution for slices of code for which previous techniques would have been constrained by the including component/object.

Second, we provide a formal framework, partially based on first-class environments semantics [15], that asserts the conditions of a sound automatic distribution. Other approaches focussed on real-world languages such as Java sources, Java bytecode or binaries. Thus, the validity of the program transformations reviewed could not easily be formally proven[1], and we even consider that the majority of them are unsound. We do not present our formal framework in this paper. Interested readers should refer to [8, 9].

Finally, there is an important difference in the goal of related research and ours : previous works have focussed on the distribution of a *stand-alone*, centralized program that is to be executed on a network of computers. Distribution

[1] Because of the technicalities involved in managing real-world languages in a formal manner.

is seen as a motivation in itself, coming from the suboptimal usage of computer resources of laboratories and companies or from the fact that a particular application should be divided between a client side and a server side. On the contrary, we do not consider stand-alone programs to be candidates for transformation : we study programs that interact with other computers by RPC-like techniques. Here, distribution is not seen as a goal in itself, but rather as a mean to minimize the physical distance between a set of distributed resources and their client code.

5 A Compiler for Automatic Introduction of Mobility

This section describes the basic techniques of automatic introduction of mobility into communicating sequential programs. We will first present on a simple example the static analysis, which is aimed at *(i)* identifying the pieces of code the compiler should transform and *(ii)* gather the information required by the transformation itself. We will then describe our compiler prototype, and discuss how these techniques should be extended to enable the compilation of higher-order languages.

5.1 Identification of Relevant Pieces of Code

The left part of Figure 4 shows a simple program computing the sum of each column of a remote matrix `m`. It is made of two nested loops : the external one (variable `i`) ranging over lines, the internal one (variable `j`) over columns.

When the optimizing compiler is given such a program, if first identifies the non-movable primitives. In this example, there is only one : `printInt`. It is not movable because its effect depends on the host it is executed on : we can not move it without modifying the semantics of the program. For a given language, there are many non-movable primitives, and we suppose we statically know all of them. The next operation performed by the compiler is to propagate the non-movability property : every piece of code from which a non-movable primitive is accessible, is also marked as non-movable. Thus, the remaining, not marked, pieces of code can be moved without compromising the global behavior of the program.

Then, the compiler empirically identifies a good candidate for remote execution. A good candidate is a loop, not marked as non-movable, that performs method calls to one or more remote objects. In the example of Figure 4, the external loop does not fit this definition, since it is not movable, but the internal loop – boxed on the figure – does. The compiler will tranform it so it will be embedded in a *mobilet* in order to be remotely executed on the server hosting the matrix `m`.

For this, the compiler needs to compute the set of input and output variables of the *mobilet* : the inputs are the variables read by the boxed statement[2], and the outputs are the written variables. In our example, the inputs of the mobilet are variables `j`, `length_y`, `s`, `m`, and `i`. Outputs are variables `s`, and `j`.

[2] Omitting the non-free variables, *i.e.* those declared in the sub-block of the statement.

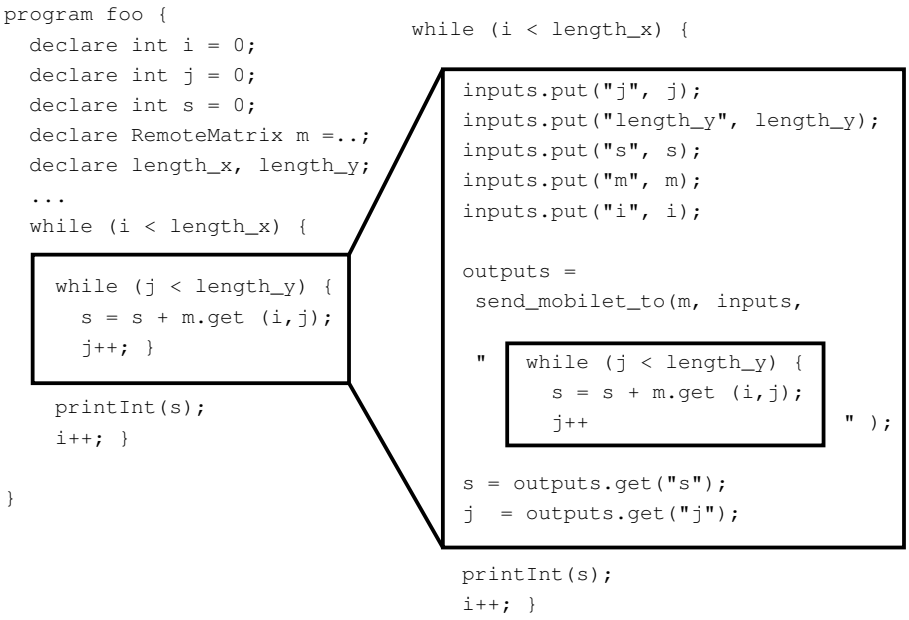

Fig. 4. Principles of code analysis and transformation

The compiler has now collected all the relevant data to produce the tranformed code, as shown on the right side of Figure 4. The boxed statement on the left side is transformed into the outer-most box on the right side. The compiler generates the filling of a variable-value pair list (namely `inputs`) containing the input chunk of the environment. It then introduces a remote execution primitive (`send_mobilet_to`) that will create a *mobilet* containing the original code – boxed on the left side –, with the input environment chunk as argument, producing an output environment chunk (`outputs`) that will be restored after the remote execution. The *mobilet* will be sent to the computer hosting the remote object reference `m` involved in the loop.

5.2 Managing Multiple Remote Calls and References in a Mobilet

When there are more than one remote object reference in the code to be sent, the following question arises : can the compiler decide what is the appropriate host to recieve and execute the *mobilet* ?

Our approach, implemented in the prototype, is to perform a *regularity test* at runtime on the references collected in the piece of code : before sending the *mobilet*, the runtime support inspects the address of a maximum of 10 references. If, and only if, all the addresses are the same, the decision is taken to send the *mobilet* to that host. This technique has a quite small, bounded overhead. Its flaw is that the compiler does not necessarily know all the references at the time

when the decision is taken. For instance, in an expression like `o.m1().m2()`, there is no possibility to anticipate, before the computation of the expression, which machine hosts the reciever of `m2`. There are other cases like this one.

Another approach is to intercept every remote call on one (or more) execution of the piece of code (without introduction of mobility), and to perform a statistical breakdown of the addresses of the receiving hosts. An empirical model can then decide, on the basis of this breakdown, whether it sounds valuable to send the *mobilet* for the next executions and where to send it. In comparison to the previous approach, the advantage of this one is that all the remote calls are taken into account. Its disadvantages are that it may be more costly because of the interceptions, and that it needs a calibration stage.

5.3 A Compiler and Analyser Prototype

Silfa : A Dedicated Toy Language. We have implemented a prototype of a compiler for automatic introduction of mobility. It compiles a dedicated toy language called Silfa – a simple imperative sequential language. Silfa users can define procedures and functions, manipulate data arrays and invoke remote operations on CORBA objects. We have decided to study a toy language since the size of the analysis code is proportional to the number of grammar rules that generate the programs : Silfa grammar is made of 45 rules, while Java's has more than 200 rules. Notice that Silfa is not object-oriented, that it is not concurrent, and that there are no pointer, and particularly no pointer to function. We will examine in section 7.1 how to extend the compiler for a higher-order language.

Compiler Implementation. The Silfa analyser and compiler is illustrated on Figure 5. When given a program, the compiler generates a set of Java source code programs, one for the main program, and several mobilets for remote execution. A regular Java compiler generates afterwards Java executable bytecode. The Silfa compiler user interface has an option to disable the generation and connection of the *mobilets*, so we can benchmark the benefits of introduction of mobility.

6 Experimental Study

The experimental study aims at showing that automatic introduction of mobility can bridge the gap between efficient and reusable distributed architectures.

6.1 Tested Applications and Experimental Settings

The first experiment we have conducted addresses the following question : there is a number of iterations performed by the *mobilets* beyond which the cost of remote execution is prohibitive, and leads to worst performances ; isn't this number too big ?

We have designed a small example to answer this question : the example is made of a server program hosting a remotely accessible two-dimensional matrix

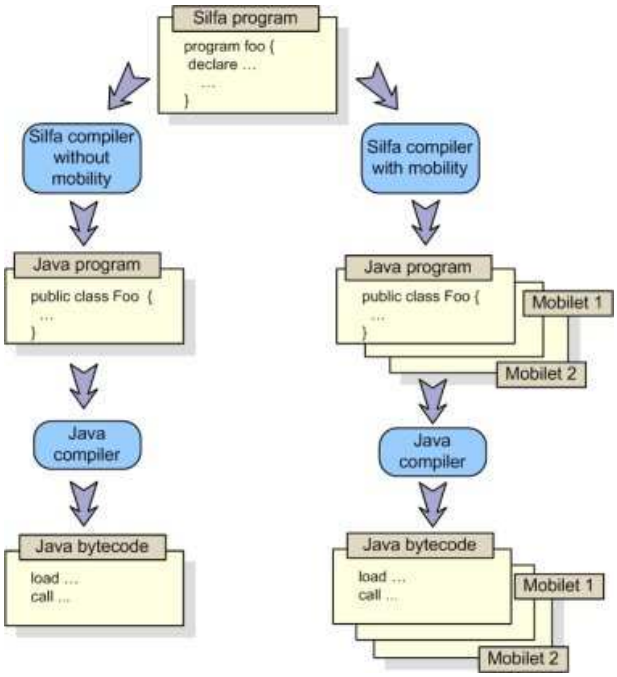

Fig. 5. Prototype of a Mobility Introduction Compiler

of integers, and a client program writing and reading the entire matrix. We noticed that for a matrix of 22×23, with a round-trip latency[3] of 1 ms, the transformed program is 50% faster than without introduction of mobility. Since such a matrix is very small compared to the typical data of industrial design applications, and since this latency is also quite small[4], we conclude that the iteration number threshold above which introduction of mobility is profitable is small enough for the targeted applications.

The second and most relevant experimentation tends to show how introduction of mobility can speed up applications that are sensitive to latency. For this, we have built an experimental network with a configurable latency : it is made of two different LANs linked by a customized Linux gateway implementing a packet delaying mechanism. This enables variation of the RTT from 225 μs (a regular LAN) to 20 ms (a high performance WAN from Paris to Nice).

Figure 6 shows this experimental network in the settings of a 3D distance computation through the interfaces of CAD Services, as presented in section 2.4.

[3] Round-trip time measured by *ping*

[4] An RTT of 1 ms is a limit timing between the performances of a bad LAN and a good WAN.

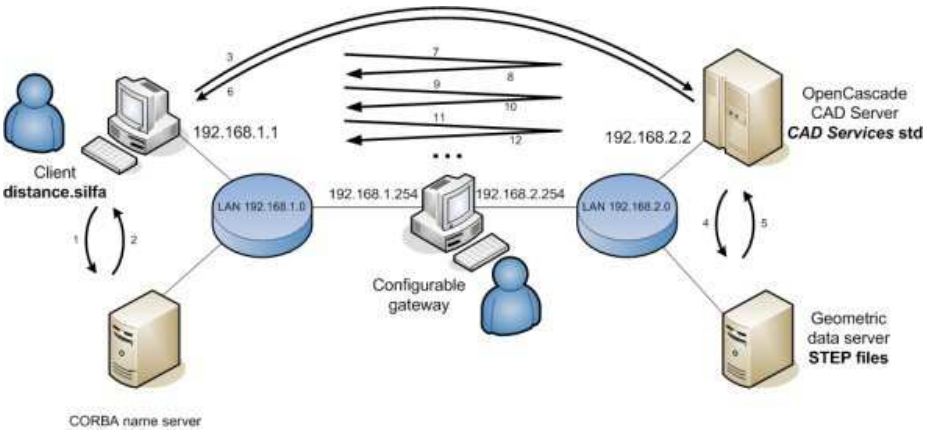

Fig. 6. Experimental network for variation of the latency

We have actually used wireframe shapes only, in order to simplify the 3D distance algorithm. The sequence executed by client program `distance.silfa` is the following :

- Perform a request to the naming service to get a reference to the CAD server (stesp 1 and 2);
- Ask the CAD server to load the two wireframe shapes (117 vertices each) from a file repository (steps 3–6);
- Range over the vertices of the first shape and compute its distance with each of the vertices of the other shape (steps 7–);
- Print the minimum of all computed distances.

This program is very simple and the size of the data (two shapes of 117 vertices) is very small. A real application would certainly generate much more remote interactions. Thus, if the experimental results are good for this example, we can expect that they would be even better for a real application.

6.2 Experimental Results

Figure 7 shows the execution timings of the 3D distance computation with and without introduction of mobility, for a latency varying from 1 to 20 ms. The first remark is that latency has a very important impact on global execution timings of the program compiled without introduction of mobility : it varies from 153 s for an RTT of 1 ms, to 2350 s for 20 ms. With our compiling techniques, the impact of latency, if not completely null, is dramatically decreased. As a result, introduction of mobility is very profitable, especially for big values of latency – and still 20 ms is not such a big latency : from Paris to Tokyo, we have noticed an average ping of more than 300 ms, between two well connected universities.

Although our compiler is designed for big latencies, it also produces a sensible optimization, for this application, in a LAN setting : with a `ping` of 225

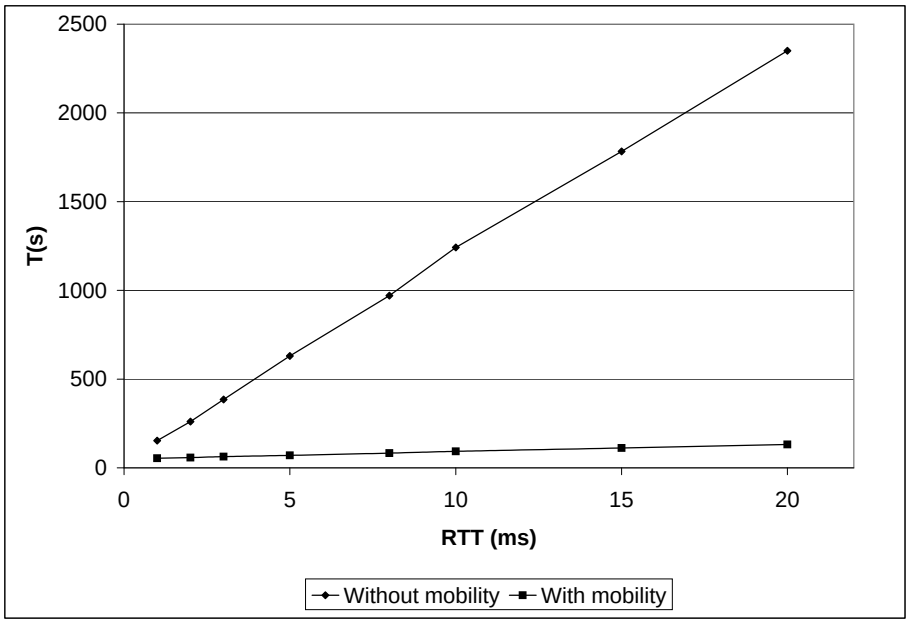

Fig. 7. Execution time *vs.* round-trip time latency (1 – 20 ms)

μs, the optimized program is 25% faster than without introduction of mobility. Conversely we noticed, as expected, that the possible speed-up depends on the applications : for the example of the remote matrix presented earlier, a `ping` of 225 μs leads to higher performances – about 20% faster – *without* introduction of mobility. It is also the case for two other applications we have tested, a ranging over a list and a ranging over a three-dimensional array.

7 Future Works

We consider two directions for continuing our research on introduction of mobility. The first direction addresses the compilation of higher-order languages. The second direction addresses the the decision process for sending a *mobilet*.

7.1 Towards a Compiler for Higher-Order Language

When the analyser computes, for instance, the movability property of expressions and statements throughout a program, it needs to consider the user-defined subroutines that are called by these expressions and statements. Indeed, a statement is not movable if it calls a non-movable procedure or function. Thus, the very first step of the analysis is to associate, for each expression and statement, the set of called subroutines. This is a kind of context-sensitive call graph. For Silfa programs, this call graph is easy to compute. But when the language is

provided with pointers to functions, first-class functions, or objects, the call graph construction is more difficult. For instance, consider an object-oriented language : for a given method call `o.m()`, the code that will be actually executed depends on the dynamic type of `o`. The knowledge of the static type of `o` and the name `m` of the method is not enough to determine the associated code. There are techniques to statically build such object-oriented call graphs [7] : the analyser alternates data flow analysis (to anticipate the dynamic types of the object references) and control flow analysis (to refine the call graph), until reaching a fixed point. We have not implemented such techniques in our prototype : we have considered that our interprocedural analysis is sufficient for our proof of concept of introduction of mobility.

Other issues have to be addressed in order to compile a mainstream object-oriented language like Java, especially concurrency and synchronisation management, and exception support.

7.2 Improvement of the Dynamic Decision for Remote Execution

We have already presented in section 5.2 a sharper decision process for choosing the appropriate machine that would recieve and execute a particular *mobilet*. In addition, the experimental study shows that the optimizing compiler, for very small latencies, may produce a slower code. We can decrease this risk by deciding whether to send the *mobilet* on the basis of the value of a metrics, that would take into account some of the different parameters affecting the speed-up : latency – of course –, by also the number of remote interactions transformed into local ones, the available computation power of the client and the server, and any other parameter that a deeper experimental study would show as relevant.

8 Concluding Remarks

We have defined a static analysis method for introducing, in a totally automated manner, mobility primitives in imperative, sequential, communicating programs. This method is implemented in an optimizing compiler designed for a simplified language. Experimental results show that the optimized programs are dramatically more efficient than non-optimized programs, as soon as the latency gets over a small threshold. Moreover, the correctness of the program transformation is formally proven [8, 9].

We believe that these kind of compilation techniques can benefit to standardization process. Indeed, we have shown that a CAD Services client can be about 18 times faster when interacting with a server located at 1000 km. One could argue that if the CAD Services standard would have included a 3D distance operation, this example would have been meaningless. It is true but it is not the point. The standard itself is not to blame : we have shown that the designers cannot anticipate all the usage scenarios of the standard. And algorithms that are not

anticipated must be implemented in the client.

If automatic introduction of mobility reaches a mature status, standard designers could partly rely on the compiler : standards could be more concise, focussing on data exchange, and would let the compiler move the non-anticipated algorithms from the client side to the server side for fast execution. It would renew confidence in standardization organizations like the OMG, since the components based on their standards would be both reusable and performant.

References

1. J.R. Allen and K. Kennedy. *PFC : A program to convert Fortran to parallel form.* Technical Report MASC TR82-6, Department of Math. Sciences, Rice University, Houston, 1982.
2. David M. Chess, Colin G. Harrison, and Aaron Kershenbaum. Mobile Agents: Are they a good idea? In *Mobile Object Systems – Toward a Programmable Internet, LNCS 1222*, pages 25–47, Berlin, Germany, 1997. Springer-Verlag.
3. R. Claus and M. Kazakov. CAD Services: An industry standard interface for mechanical CAD interoperability. In *Proceedings of Concurrent Engineering 2003 conference*, ISBN 90-5809-622-X, 2003.
4. M. Dahm. The doorastha system. *Technical report B-1-2000, Freie Universität Berlin*, 2000.
5. Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson. Application Frameworks. In Mohamed E. Fayad, Douglas C. Schmidt, and Ralph E. Johnson, editors, *Building Application Frameworks*, pages 29–54, New York, 1999. Wiley Computer Publishing.
6. Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
7. David Grove, Greg DeFouw, Jeffrey Dean, and Craig Chambers. Call graph construction in object-oriented languages. In *ACM Conference on Object-Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 108–124, 1997.
8. Grégory Haïk. Introduction of Mobility for Distributed Numerical Simulation Frameworks : a Formal Study. Technical Report LIP6 2003/008, Université Pierre et Marie Curie – Laboratoire d'Informatique de Paris 6, Paris, France, 2003.
9. Grégory Haïk. *Introduction de mobilité dans les applications de conception industrielle.* PhD thesis, Université Pierre et Marie Curie (Paris 6) – Laboratoire d'Informatique de Paris 6, 2005.
10. Galen C. Hunt and Michael L. Scott. The coign automatic distributed partitioning system. In *Operating Systems Design and Implementation*, pages 187–200, 1999.
11. Ralph E. Johnson. Components, frameworks, patterns. In *ACM SIGSOFT Symposium on Software Reusability*, pages 10–17, 1997.
12. D.J. Kuck, Y. Muraoka, and S.C. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speed-up. In *IEEE Transactions on Computers C-21*, pages 1293–1310, 1972.
13. Object Management Group. Computer Aided Design Services Specification, OMG document formal/05-01-07, version 1.2, 2005.
14. Michael Philippsen and Matthias Zenger. JavaParty — transparent remote objects in Java. *Concurrency: Practice and Experience*, 9(11):1225–1242, nov 1997.

15. Christian Queinnec and David De Roure. Sharing code through first-class environments. In *Proceedings of ICFP'96 — ACM SIGPLAN International Conference on Functional Programming*, pages 251–261, Philadelphia (Pennsylvania, USA), 1996.
16. Salome. http://www.salome-platform.org/.
17. André Spiegel. Automatic distribution in pangaea. *Proc. Workshop on Communications-Based Systems, CBS 2000*, April 2000.
18. Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" Java software. *Lecture Notes in Computer Science*, 2072:236–256, 2001.
19. E. Tilevich and Y. Smaragdakis. J-orchestra: Automatic java application partitioning, 2002.