Enhancing Remote Method Invocation through Type-Based Static Analysis

Carlo Ghezzi, Vincenzo Martena, and Gian Pietro Picco
Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.zza L. da Vinci, I-20133 Milano, Italy
{ghezzi,martena,picco}@elet.polimi.it

ABSTRACT

Distributed applications rely on middleware to enable components to interact with each other remotely. Thus, the overall performance of the distributed application increasingly depends on the interaction between the implementation of the various components and the features provided by the middleware. In this paper we analyze Java components and the RMI middleware provided by the language, and we discuss the opportunities for optimizing remote method invocations.

Specifically, we discuss how parameter passing among distributed objects can be optimized through fairly standard static program analysis techniques. Parameter passing can be costly: large object parameters need to be serialized and transmitted over the network. Our optimization allows only object portions to be transmitted, corresponding to what is actually used at the server side. The paper presents the program analysis technique we employ and outlines an implementation of the run-time optimization it enables.

1. INTRODUCTION

Two major trends characterize the evolution of software technology during the past decade. On the one hand, software applications are becoming increasingly distributed and decentralized. On the other, off-the-shelf components are increasingly used as building blocks in composing a distributed applications. The gluing mechanisms that support the assembly of components are provided by the middleware.

Although much progress has been achieved in the past in supporting designers while developing distributed applications, it is still true that the level of support provided for traditional centralized software is much more mature. While full support is available when designing individual components, little help is provided in the global context provided by the distributed infrastructure, in which components have to be deployed, configured, and interconnected.

As an example, consider the problem of code optimization. Traditional approaches deal with code optimization in the context of a compilation process to a predefined target architecture—very often, a single-processor machine. Automatic optimization allows the programmer to concentrate more on the correct development of the program, its ease of change, and other desirable qualities, instead of dealing with tiny details and maybe loosing control of the overall structure of the program. Consider instead the case where an application transitions from a centralized implementation to a distributed one, supported by some middleware layer. Or the case where a new application is developed using both a traditional programming language for components and some middleware layer for interconnecting them. We are not aware of automatic optimizations techniques that span over two domains: the domain of the programming language used inside a component and the domain of the middleware that is used to interconnect the components. This is true also in the case where the two domains are in the same linguistic framework, as in the case of Java and Java RMI [14].

This is exactly our topic of interest. Specifically, we consider the case where components are written in the Java programming language, and Java RMI is used as middleware. In particular, we concentrate on parameter passing across network boundaries, when object methods are invoked remotely. Java allows parameters to be serialized when passed from one node to another. Serialization allows objects to migrate over the network. This, however, may cause overhead and reduce performance. Maybe only a small part of a huge serializable object is actually used remotely, and therefore performance can be improved by transmitting only a small serialized portion of the object.

In this paper we discuss how this can be done by statically analyzing the bytecode of a Java program and then using the results to optimize the object serialization. We use fairly standard static analysis techniques. What is new in this paper is how this analysis is used and the context in which it is used. Our technique is particularly valuable in the case where off-the-shelf components are used to build a distributed application. In this case, in fact, the designer has no visibility of the internals of the components, and therefore many opportunities for hand optimizing inter-component interactions are necessarily missed. Moreover, since our technique is aimed at reducing the communication overhead, it is particularly useful in bandwidth-constrained scenarios like those defined by mobile computing.

The paper is structured as follows. Section 2 provides the reader with the background about Java serialization and RMI. Section 3 discusses the motivation for our work and

states the problem, by relying on a reference example that is used in the rest of the paper to illustrate our technique. Section 4 presents the details of our program analysis approach, and Section 5 shows how its results can be exploited at run-time by redefining the serialization by relying only on the Java API. Section 6 discusses benefits and limitations of our techniques, and suggests possible enhancements and exploitations. Section 8 reports about a prototype tool suite supporting we are currently building to support our approach. Section 7 briefly surveys related work. Section 9 ends the paper with brief concluding remarks.

2. BACKGROUND

In this section we provide the reader with the necessary background on object serialization and remote method invocation. The presentation is focused on Java, albeit most of the concepts can be found in similar platforms, e.g., in the Remoting API of Microsoft .NET [24].

2.1 Object Serialization

Object serialization is the process of flattening an object, whose data is structured according to the composition relationship, into a stream of bytes. This is fundamental to perform input/output with objects, and in particular for saving them on persistent storage, or transferring them across a network link.

In Java, the serialization process is accomplished by using two special input/output streams, `ObjectInputStream` and `ObjectOutputStream`. When an object reference r is written to the latter, the Java run-time recursively serializes the attributes of r, the attributes of the attributes of r (if they are object references), and so on, until the whole graph of objects rooted at r has been serialized. If an attribute value is `null`, this value is written to the serialization stream. Similarly, if an attribute value has a primitive type (e.g., `int`) it is serialized by using a default format. Class descriptors are also inserted in the serialization stream to provide the receiving side with enough information to locate the correct type at deserialization time. Deserialization essentially proceeds backwards, by extracting information from the serialization stream and reconstructing the object graph accordingly.

Interestingly, serialization preserves aliases within a single serialization stream. For instance, if two object references pointing to the same object are written one after the other without closing the stream, when deserialized they will still point to the same copy of the original object.

The aforementioned process, however, requires the object reference to belong to a type that has been explicitly declared to be serializable, which in Java is accomplished by implementing the `java.io.Serializable` interface. This interface does not contain any method or constant, and serves the only purpose of tagging the implementing class as serializable. In addition, the programmer retains control over the fraction of the object graph that must be serialized. In fact, although by default all the object attributes are serialized, attributes that are prepended by the `transient` keyword are not. When the object is reconstructed by deserialization, transient attributes are set to the language default for their type.

The object serialization API provides means for the programmer to redefine many facets of the serialization behavior, e.g., changing the policy to locate the object code, replacing an object with another one, and so on. In particular, a class may specify its own (de)serialization through two private methods `writeObject` and `readObject` which, if defined, are used in place of the aforementioned process. As we describe in Section 5, this feature can be used to implement our optimization in RMI applications.

2.2 Remote Method Invocation

Remote method invocation enables the development of distributed applications by using essentially the same programming constructs employed in a local setting, and can be regarded as an evolution of remote procedure call recast in the object-oriented paradigm.

Although our work deals with Java, the results enjoy wider applicability, since several middleware support some flavor of remote method invocation.

2.2.1 Basic Concepts

In RMI, a line is drawn between *remote objects* and *non-remote objects*. A "remote object is one whose methods can be invoked from another Java virtual machine, potentially on a different host" ([14], §2.2), i.e., a potential target of a remote method invocation. Remote objects are defined programmatically by any class that implements the `java.rmi.Remote` interface or a subtype thereof. All the other objects are simply called non-remote objects.

The remote interface plays a role analogous to interfaces written in an *interface definition language* (IDL) like in CORBA or DCOM [5], although here interface and implementation are specified using the same language. The caller accesses the remote object only through the methods in its interface: hence, the implementation of the callee can be changed independently, although constrained by the dynamic binding rules of the language. In RMI, such an implementation can even be unknown at compilation time, and it can be dynamically transferred and linked on the fly if and when needed at run-time. This removes the need to determine in advance a proper deployment of stub code on the hosts involved in the distributed application.

To be remotely accessible, however, remote objects must be instrumented properly. In essence, the implementation of `hashCode`, `equals`, and `toString` must be redefined to take distribution into account. Moreover, the object needs to be *exported*, which essentially consists of making it known to the run-time, and able to accept incoming calls. The simplest way to meet these constraints in the implementation of a remote object is to subclass from `java.rmi.server.UnicastRemoteObject`, which provides all the necessary functionality. In this case, the object is automatically and implicitly exported to the run-time support upon invocation of the constructor. Otherwise, it is the programmer's responsibility to meet the aforementioned constraints in the definition of her own class implementing `Remote`, and to export it explicitly by using the static method `exportObject` provided by `UnicastRemoteObject`.

There are essentially two ways through which a reference to a remote object can be acquired by the client side of a remote method invocation. The first and most straightforward one is by querying a lookup service—or *registry* in the RMI jargon. The registry is a process that binds local[1] objects to symbolic names. Figure 1 shows a code snippet

[1]For security reasons, in the current RMI implementation the registry must be on the same machine hosting the objects being bound.

```
public static void main(String[] args) {
  String name = "//localhost/printer";
  IPrinter printer = new DotMatrixPrinter();
  try {
    Naming.rebind(name, printer);
  } catch (Exception e) {
    //exception handling code
  }
}
```

Figure 1: Binding an object to an RMI registry.

```
...
IPage aPage = new MixedPage(20);
IPrinter printer = (IPrinter) Naming.lookup("//myregistry/printer");
printer.print(aPage);
...
```

Figure 2: Obtaining a remote reference and performing a method invocation.

where an object of type `DotMatrixPrinter`, implementing the remote interface `IPrinter`, is bound to the registry and made accessible through the symbolic name `printer`. Remote clients can query the registry by providing a symbolic name, and obtain a network reference to the corresponding object[2]. The remote nature of this reference is transparent to the application developer. In particular, it can be the target of a method invocation, that is handled accordingly by the RMI run-time support. An example is shown in the code snippet in Figure 2, where a client creates a page to be printed, acquires a reference to the print service exported in Figure 1 by looking up the registry, and then invokes the print service through remote method invocation.

The other means to obtain a reference to a remote object is through parameter passing, which is examined next.

2.2.2 Passing Parameters to Remote Methods

In object-oriented languages, objects are typically passed by reference in method invocations. Maintaining this choice in a distributed setting hides distribution to the programmer, and hence simplifies her task. However, when a method invocation is remote the object reference is actually stretched across the network: accesses to the formal parameter on the callee host trigger communication back to the caller site, where the appropriate actions on the actual parameter are performed. If the parameters were instead copied and transferred to the callee site, communication would be reduced, but semantics of parameter passing would be different from the one of the host language, hence making the programmer's task more complex and error prone. This tradeoff is solved in different ways by existing systems. For instance, in CORBA [16] objects are always passed by reference, while `struct`s and sequences are passed by copy. In Java RMI, objects can be passed through parameters either by reference or by copy. If the object being passed (either as an actual parameter or as a return value) in a remote method invocation is a remote object and it has been exported, then the object is passed by reference, i.e., it is accessed through the network. Instead, if the parameter is a non-remote object, or it is a remote object that has not been exported yet, it is passed by copy. In this case, however, the type of the object is required to implement the interface `java.io.Serializable`. Primitive types are always passed by copy. Hence, in Java RMI the programmer is aware of distribution, and can retain some degree of control over it, by deciding—albeit only at design time—whether an object should be copied and hence accessed locally, or instead accessed through the network. In contrast, a fully distribution-transparent approach may lead to macroscopic inefficiences, as discussed for instance in [26].

[2]What is returned is actually a *stub*, i.e., automatically generated code that acts as a local proxy to the remote object, and manages the communication necessary to support remote method invocation.

2.2.3 The Role of Serialization

The semantics of parameter passing *by copy* is defined in Java RMI by object serialization. The interplay between serialization and parameter passing, however, slightly complicates the picture. Since it is relevant to the results we present here, we hereby delve into further details.

The first issue is aliasing. Since a single serialization stream per remote method invocation is used, references to the same object in the caller are mapped in the callee into references to the same serialized copy of that object. As a particular case, two actual parameters that are aliases result in two aliased formal parameters in the callee. Hence, the two parameters are not copied independently, as usually happens in programming languages supporting parameter passing by copy.

The other issue has to do with serialized objects containing references to remote objects. In this case, the behavior of RMI is as follows ([14], §2.6.5):

- If the object being serialized is an instance of `Remote` and the object is exported to the RMI run-time, the stub for the remote object is automatically inserted in the serialization stream in place of the original object.
- If the object is an instance of `Remote` and the object is not exported to the RMI run-time, or the object is not an instance of `Remote`, the object is simply inserted in the serialization stream, provided it implements `Serializable`. If the object is not serializable, a `NotSerializableException` is raised.

In essence, this preserves the semantics of object references in presence of distribution. If the object o contains a remote object r in its object graph, the serialized copy of o will still access the original copy of r on the original node, provided that r has been exported. Otherwise, r will be treated just like any other ordinary object.

Java RMI provides a number of other features, including dynamic class loading, activation, and security. For a description of these features, we redirect the interested reader to [14], since they do not fall in the scope of this work. Instead, we now describe the problem we identified with RMI serialization, and the solution we devised.

3. MOTIVATION, PROBLEM STATEMENT, AND REFERENCE EXAMPLE

RMI provides a simple and powerful mechanism for building distributed applications, whose semantics strikes a reasonable balance between expressiveness and efficient use of communication. One of the known weaknesses of RMI, however, is its reliance upon object serialization. In Java—and in object-oriented languages in general—objects are often

```
public interface IPrinter extends Remote {
  public void print(Page page) throws RemoteException;
}

public interface IPage extends Serializable {
  public IPageElement[] getWholePage();
  public IPageElement[] getTextElements();
  public IPageElement[] getGraphicElements();
}

public interface IPageElement extends Serializable {
  public void print();
}
```

Figure 3: Interfaces for a simple print service.

very structured: composition may quickly lead to pretty large object graphs. This is usually not a problem when the use of an object is limited to a given host, and even a poor use of the composition relation usually does not bear immediate negative consequence on the performance of the overall application. Nevertheless, when the object must be transferred to another host, e.g., during a remote method invocation, the degree of structuring of an object has a great impact on performance. An overhead is introduced in terms of both computation, since both serialization and deserialization require the object graph to be recursively navigated, and communication, since large objects obviously result in large serialization streams being transmitted.

Most of the existing approaches focus on reducing the computational overhead introduced by serialization [10, 1, 9, 2, 17]. They aim at improving the Java/RMI run-time, without considering the application code running and, in particular, how the object is used after deserialization. In this paper, we take the complementary approach of exploiting the knowledge about the use of serialized objects on the server side to achieve opportunities of further optimizations, in particular for reducing the size of the serialized object and hence reducing the network traffic.

Different uses of a deserialized object may stress different portions of the object. For instance, different method invocations may access different subsets of the object fields. Unfortunately, in Java an object is always serialized in the same way, regardless of its use after deserialization. The only degrees of freedom left to the programmer are the ability to mark some of the fields as `transient` and the ability to redefine serialization. The former enables the programmer to tune serialization only at the class level rather than the object level; still, serialization cannot be tailored according to how the object is going to be used (which may vary from call to call). The latter can be used in principle to achieve the optimization we are suggesting, but it would require the programmer to track down and manage a prohibitive amount of information.

Our approach, instead, consists of (a) performing static analysis to derive information about the portions of serializable parameters that need to be transmitted at each call point, and (b) use this information at run-time to drive the serialization process accordingly.

In what follows, we define a simple reference example, which will be used throughout the paper. The example deals with a print service. The service is provided through an `IPrinter` interface, shown in Figure 3. Clients are expected to invoke the only method `print()` by passing the page to

```
public class DotMatrixPrinter extends UnicastRemoteObject
                              implements IPrinter {
  private PrintStream out = new DotMatrixPrintStream();
  public DotMatrixPrinter() throws RemoteException { super(); }
  public void print(IPage page) throws RemoteException {
    IPageElement[] text = page.getTextElements();
    if (text != null)
      for (int i = 0; i < text.length; i++)
        text[i].print(out);
  }
}
public class Plotter extends UnicastRemoteObject
                     implements IPrinter {
  private PrintStream out = new PlotterPrintStream();
  public Plotter() throws RemoteException { super(); }
  public void print(IPage page) throws RemoteException {
    IPageElement[] graphics = page.getGraphicElements();
    if (graphics != null)
      for (int i = 0; i < graphics.length; i++)
        graphics[i].print(out);
  }
}
public class InkJetPrinter extends UnicastRemoteObject
                           implements IPrinter {
  private PrintStream out = new InkJetPrintStream();
  public InkJetPrinter() throws RemoteException { super(); }
  public void print(IPage page) throws RemoteException {
    IPageElement[] elements = page.getWholePage();
    if (elements != null)
      for (int i = 0; i < elements.length; i++)
        elements[i].print(out);
  }
}
```

Figure 4: Implementations of the printer interface.

be printed as a parameter[3]. Pages are made up of page elements; both are manipulated through the other two interfaces shown in Figure 3. A fundamental characteristics of pages, in our example, is that they can contain text and/or graphical elements. The methods exported by the `IPage` interface allow one to retrieve either or both. A `PageElement` object exports a single method `print()`, which is invoked by the receiving `IPrinter` and causes the actual printing of the element on the printing device.

The code in Figure 3 constitutes a reasonable API for our printing service, and one that decouples sharply the service interface from its implementation. A client can request a page printout regardless of the target device, without any need to change the client implementation. It will be up to the `IPrinter` server to do the printing according to its own capabilities, i.e., printing only text, only graphics, or both. Nevertheless, it is precisely this (desirable) separation of concerns that backfires in terms of communication performance. To understand why, let us consider what happens when multiple printing devices with different capabilities are available. In Figure 4 we show three possible implementations of the `IPrinter` interface, meant to be used with a dot-matrix printer, a plotter, and an ink-jet printer, respectively. All of them export the `print()` method. However, a dot-matrix printer can only print text-only pages, a plotter can only print graphics, while an ink-jet printer can print both.

The implementation of pages and page elements must be

[3]Clearly, this is unrealistic, as printing usually involves documents, which are in turn composed of pages. Nevertheless, we choose to focus on a single page in order to keep the example simple and compact. Further improvements would also be possible from an object-oriented programming style.

```
public class TextPage implements IPage {
  private TextElement[] pageElements;
  public TextPage(int pageDimension) {
    pageElements = new TextElement[pageDimension];
  }
  public IPageElement[] getWholePage() { return pageElements; }
  public IPageElement[] getTextElements() { return pageElements; }
  public IPageElement[] getGraphicElements() { return null; }
}
public class GraphicPage implements IPage {
  private GraphicElement[] pageElements;
  public GraphicPage(int pageDimension) {
    pageElements = new GraphicElement[pageDimension];
  }
  public IPageElement[] getWholePage() { return pageElements; }
  public IPageElement[] getTextElements() { return null; }
  public IPageElement[] getGraphicElements() { return pageElements; }
}
public class MixedPage implements IPage {
  private IPageElement[] pageElements;
  public MixedPage(int pageDimension) {
    pageElements = new IPageElement[pageDimension];
  }
  public IPageElement[] getWholePage() { return pageElements; }
  public IPageElement[] getTextElements() {
    TextElement[] text = new TextElement[pageElements.length];
    int j = 0;
    for (int i = 0; i < pageElements.length; i++)
      if (pageElements[i] instanceof TextElement)
        text[j++] = pageElements[i];
    return text;
  }
  public IPageElement[] getGraphicElements() {
    GraphicElement[] graphics =
      new GraphicElement[pageElements.length];
    int j = 0;
    for (int i = 0; i < pageElements.length; i++)
      if (pageElements[i] instanceof GraphicElement)
        graphics[j++] = pageElements[i];
    return graphics;
  }
}
```

Figure 5: Implementations of a page.

```
public class GraphicElement implements IPageElement {
  private int[][] colors;
  public GraphicElement(int dimension1, int dimension2) {
    colors = new int[dimension1][dimension2];
  }
  public void print(PrintStream out) {
    for (int i = 0; i < colours.length; i++)
      for (int j = 0; j < colours[i].length; j++)
        out.print(colors[i][j]);
  }
}
public class TextElement implements IPageElement {
  private char[] characters;
  public TextElement(int dimension) {
    characters = new char[dimension];
  }
  public void print(PrintStream out) {
    for (int i = 0; i < characters.length; i++)
      out.print(characters[i]);
  }
}
```

Figure 6: Implementations of a page element.

defined accordingly. Figure 5 shows three possible implementations of `IPage` supporting text, graphics, and composite pages, respectively. A `TextPage` contains textual elements only, a `GraphicPage` contains graphic elements only, and a `MixedPage` contains both. In this latter case, the methods `getTextElements` and `getGraphicElements` are defined to return only the appropriate subset of page elements. Figure 6 shows a possible implementation of the `IPageElement` interface. The class `TextElement` contains an array of characters representing the text fragment associated with the element. Similarly, `GraphicElements` contains a color matrix representing, with some simplification, the picture to be drawn. It is interesting to note that, while printing is invoked by the page element through its `print` method, the page element does not have any knowledge about the innards of the printing process: it simply dumps its data to the print stream `out`, that has been bound to the appropriate implementation passed by the calling server object. Again, this enables a full decoupling between the document logic contained in the page element and the printing logic contained in the server.

A user may print the same page on different printers at different times, e.g., depending on proximity. Clearly the result may be somewhat degraded with respect to the page content. For example, a page containing text and graphics can still be printed on a dot-matrix printer. In this case, however, only the text will be printed.

In this example, the definition of the page contents lies entirely with the client, while the definition of how such content is manipulated on the server lies entirely on the latter. This is compliant with the principles of object-oriented design, and information hiding at large.

This implementation choice, unfortunately, may raise performance problems at execution time. Consider, for example, the case where a composite page is to be printed on a current printer that, due to dynamic binding, happens to be a dot-matrix printer. Although only textual elements are actually accessed by the printer, all page elements are actually transferred to the server. This is an example where serialization and transmission of a large unused portion of an object generates unnecessary overhead. The key point here is that the client code does not have any means to avoid this unnecessary serialization—unless information hiding is broken. Our assumption was that the client must be able to print *the same page* on any printer, regardless of the printer implementation.

We can generalize from this example. Often the client has no control over the server's behavior. The actual kind of server may change over time, due to dynamic binding, and different servers, though presenting the same interface, may differ in their internal behaviors. Internal behaviors are not visible, either because this has been a deliberate design choice, as in the example, or because we are dealing with an off-the-shelf component, whose implementation has been made by a third party.

In either case, and back to the example, the client "sees" a printer only through the `IPrinter` interface: it has no means to know whether it supports text and/or graphics. All a client can do is to serialize the whole page content, potentially including elements that cannot be printed by the target printer object.

The next section presents a program analysis technique that enables run-time optimization of remote method invocations in an RMI application. Situations like the one we described can be detected automatically during a static analysis phase that determines which fields of a given object involved in a remote method invocation are actually used by the distributed application and which are not. The results of the analysis can be exploited by automatically generating application code that selects the best serialization steps on a

per-invocation basis, by skipping the serialization of unused objects. The combination of the two leads to a programming support environment that enables the programmer to write applications without giving up the benefits of encapsulation and information hiding, and yet transparently optimizes the use of communication resources. Needless to say, the optimization is performed without compromising the correctness of the application, i.e., it guarantees that there will never be an attempt to access an object that has not been serialized.

The next section describes our analysis technique in detail. Section 5 contains instead a description of how the analysis results can be exploited for improved run-time serialization by relying only on the Java API, i.e., without modifying the Java Virtual Machine.

4. TYPE-BASED STATIC ANALYSIS

This section describes our static analysis technique in a stepwise manner. We begin by describing the overall analysis strategy, then introduce the notion of concrete graph, which is central to our approach, and conclude by describing the details of the analysis.

4.1 Overall View

Our goal is to identify, for each remote method invocation `r=o.m(`$p_1, \ldots, p_n$`)` and for each serializable parameter $p_i \in \{r, p_1, \ldots, p_n\}$, which attributes of p_i need to be copied through serialization and passed to the remote target object. To achieve this goal, we focus our analysis on types that can be instantiated directly, e.g., through a `new` operation. These types, which we call *concrete types*, include all the primitive types, and do not include any abstract class or interface.

Our analysis technique is structured in the following phases:

1. Given the overall set of types constituting the distributed application, determine:
   (a) the set $\mathcal{R}$ of *remote types*, i.e., types that extend or implement the `Remote` interface;
   (b) the set $\mathcal{S}$ of *serializable types*. This includes primitive types (e.g., `int`) and reference types that extend or implement[4] the `Serializable` interface. The declaration of an array `T[]` causes the insertion in $\mathcal{S}$ of both `T` and the type "array of `T`".
2. From the above sets, compute the set of *concrete* remote and serializable types, i.e., the subsets $\mathcal{R}_c \subseteq \mathcal{R}$ and $\mathcal{S}_c \subseteq \mathcal{S}$ containing only concrete types, according to the definition above.
3. For each class belonging to $\mathcal{R}_c$, identify the set $\mathcal{M}$ of methods that can be invoked remotely. For a given class in $\mathcal{R}_c$, $\mathcal{M}$ includes all the methods implementing (directly or through overriding) the corresponding method declarations provided by the interfaces extending `Remote` and implemented by the class.
4. For each remote invocation of a method $m \in \mathcal{M}$, identify the set of parameters $\mathcal{P}_m$ that must be serialized, i.e., those for which at least one dynamic type belongs to $\mathcal{S}_c$.
5. For each parameter $p \in \mathcal{P}_m$, identify the attributes of p for which serialization can be safely skipped.

[4]According to Java inheritance rules, a class can implement an interface directly, or indirectly through a superclass that implements it directly.

$\mathcal{R} =$ {IPrinter, DotMatrixPrinter, Plotter, InkJetPrinter}
$\mathcal{R}_c =$ {DotMatrixPrinter, Plotter, InkJetPrinter}
$\mathcal{S} =$ {IPage, TextPage, GraphicPage, MixedPage, IPageElement, IPageElement[], TextElement, TextElement[], GraphicElement, GraphicElement[], int, int[], int[][], char, char[]}
$\mathcal{S}_c =$ {TextPage, GraphicPage, MixedPage, TextElement, TextElement[], GraphicElement, GraphicElement[], int, int[], int[][], char, char[]}
$\mathcal{M} =$ {print}
$\mathcal{P}_{print} =$ {aPage}

Figure 7: The sets of program elements relevant to our analysis found in our reference example.

The first three phases of the analysis can be accomplished straightforwardly through a simple inspection of the inheritance hierarchy. The fourth phase involves determining the set of remote method invocations `r=o.m(`$p_1, \ldots, p_n$`)` that are actually present in the code. This must take into account all the possible dynamic types of `o` which, again, can be accomplished with standard techniques inspecting the inheritance hierarchy. Figure 7 shows the result of phases 1 through 4 for our reference example.

Phase 5 is the most complex. It constitutes the core of our analysis and the main contribution of this paper. Given a remote method invocation in the form `r=o.m(`$p_1, \ldots, p_n$`)`, the analysis is complicated by polymorphism. In fact, given a parameter p_i, its static type T can be replaced at run-time by any subtype of T. The same holds for every attribute of T, and so on recursively. For each parameter p_i, we must determine whether each attribute that is potentially reachable through p_i must be serialized or whether we can safely avoid to do it.

The next two sections describe precisely how this can be done. Section 4.2 defines the notion of *concrete graph,* the data structure used to represent the type information associated with an object reference. Section 4.3 describes how concrete graphs are used by our analysis algorithm.

4.2 Concrete Graphs

Each parameter of a remote method invocation can be associated with one or more descriptors, called concrete graphs. Intuitively, a concrete graph associated with a reference parameter p of type T is a directed multi-graph[5] that represents the type structure of one of the possible instances of p at runtime, according to the class hierarchy. The nodes of the concrete graph are serializable types belonging to $\mathcal{S}_c$. Each edge departs from a node representing the type of an object, ends in the node representing the type of one of the object's attributes, and is labeled with the name[6] of such attribute.

The concrete graphs associated to a given parameter can be computed easily through a simple inspection of the static class hierarchy. As an example, Figure 8 shows the two con-

[5]That is, a node can be linked to another through more than one edge.

[6]Attribute names are fully specified, i.e., they must include the type where they are defined, to account for inheritance. For simplicity, however, the figures in the rest of the paper do not show the type information.

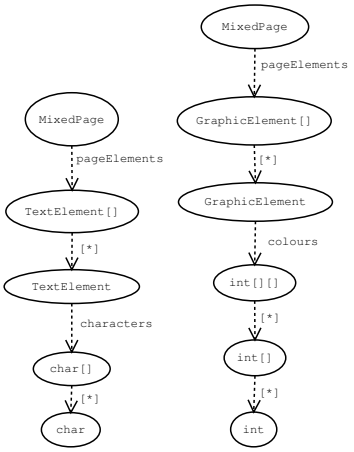


Figure 8: The possible concrete graphs for the parameter `aPage` in the invocation of `print` in Figure 2.

crete graphs for the parameter `aPage` shown in Figure 2, built using the class definitions shown earlier. The two concrete graphs differ according to the assumptions about the array attribute `pageElements` of `MixedPage`. The graph in Figure 8(a) describes the case where an element of the array, accessed through an index, is of type `TextElement`. The other graph, in Figure 8(b), describes the case where the element is instead of type `GraphicElement`. Clearly, the array can in general contain any combination of the two.

Arrays require a little more explanation. We treat indexing in the array by labelling the edge of the concrete graph with the special label `[*]`. For example, the graph in Figure 8(a) represents the case where indexing yields a `TextElement` element from the array. Moreover, Java arrays are objects containing a built-in attribute `length`. While our analysis takes into account this attribute, the concrete graphs disregard it, since its serialization cannot be redefined.

Formally, a concrete graph can be represented as a tuple $\langle \mathcal{N}, \mathcal{E}, \mathcal{S}_c, \mathcal{A}, \mathit{type}, \mathit{attr} \rangle$. $\mathcal{N}$ and $\mathcal{E}$ are respectively the set of nodes and edges of the graph, with

$$\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N} \times \mathcal{A}$$

where $\mathcal{A}$ is the set of attribute names. $\mathcal{S}_c$ is the set of serializable concrete types. The functions *type* and *attr* map the object structure on the concrete graph:

$$\begin{aligned} \mathit{type}: &\quad \mathcal{N} \rightarrow \mathcal{S}_c \\ \mathit{attr}: &\quad \mathcal{E} \rightarrow \mathcal{A} \end{aligned}$$

The function *type* associates a serializable type to each node of the concrete graph, with the constraint that

$$\nexists n_1, n_2 \mid \mathit{type}(n_1) = \mathit{type}(n_2)$$

i.e., each type appears in the concrete graph exactly once. The function *attr* allows one to retrieve the name associated to an edge. For instance, if n_1 and n_2 are the first two nodes of the concrete graph in Figure 8(a), and e the first edge, then $\mathit{type}(n_1) =$ `MixedPage` and $\mathit{attr}(e) =$ `pageElements`.

Intuitively, concrete graphs are used to optimize serialization as follows. First, we assume that, for all remote invocations, each serializable parameter has its associated set of concrete graphs. Static analysis is then performed by examining each concrete graph and determining, for each field, whether it is used on the receiving side, and hence should be serialized[7]. This information is recorded by suitably annotating the edges of the concrete graph. At run-time, since the dynamic type of each node of the object graph to be transferred is known, the information stored in the concrete graph gives all the information needed to define which attributes should be serialized and then transmitted.

4.3 The Analysis in Detail

Armed with this knowledge, we can now describe the core of our technique, i.e., phase 5 of the program analysis described in Section 4.1. To simplify the presentation, we focus on method invocations with a single input parameter. The analysis can be adapted easily to method signatures with arbitrary arity and types[8]. Similarly, we limit the discussion to the serialization of input parameters. The analysis of whether the serialization of the result value can be optimized can be achieved straightforwardly by using the same technique described here, but analyzing the client code using the return value instead of the server code using the input parameters.

Exploiting the concrete graph. The analysis of a given method `m(p)` starts by building the concrete graphs associated to p. Then, it analyzes the control flow of `m`. As the analysis walks through the body of `m`, it "decorates" each concrete graph of p by keeping track of whether a given attribute can be serialized or not, based on how the control flow has used the attribute thus far. This information is derived incrementally as the control flow is examined, and relies on the definition of two labelling functions that map each edge of a concrete graph to a boolean value:

$$\begin{aligned} \mathit{defined}: &\quad \mathcal{E} \rightarrow \{\mathit{true}, \mathit{false}\} \\ \mathit{skip}: &\quad \mathcal{E} \rightarrow \{\mathit{true}, \mathit{false}\} \end{aligned}$$

The value returned by $\mathit{defined}(e)$ is *true* if the attribute associated with the edge e (i.e., $\mathit{attr}(e)$) has been already assigned a value at a given point in the analysis. Instead, the value of $\mathit{skip}(e)$ is *true* if the attribute $\mathit{attr}(e)$ can be safely skipped during the serialization process of the parameter p associated to the concrete graph. Essentially, the value of *skip* eventually holds the final result of the analysis, while the value of *defined* is relevant only while the analysis is being performed.

Analyzing the control flow. To inspect the control flow of the invoked method, our analysis exploits a standard data-flow framework as described in [15]. In this approach, the control flow of a program is described by a *control flow graph,* where nodes represent program statements and edges represent the transfer of control from one statement to another. As an example, Figure 9 shows the control flow graph

[7]Clearly, in the case of recursive types only an approximation is possible.

[8]A simple way to do this is to represent parameters as attributes of a fake, single parameter. Incidentally, this solution has the additional benefit of providing a way to detect parameters whose serialization can be avoided *entirely.*

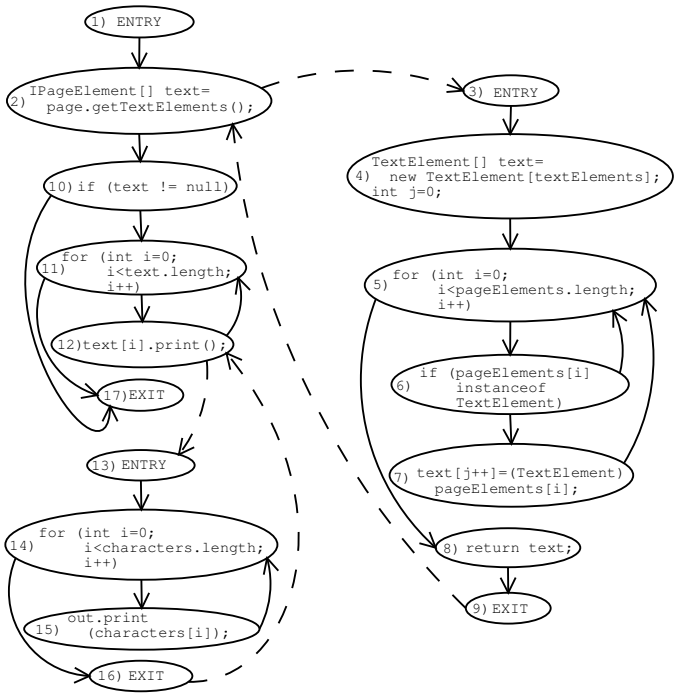


Figure 9: The control flow graph of the methods `DotMatrixPrinter.print`, `TextPage.getTextElement`, and `TextElement.print`.

for the methods `print` in `DotMatrixPrinter`, `getTextElement` in `TextPage` (invoked by `print`), and `print` in `TextElement` (invoked by `getTextElements`). The control flow graph of each method starts with an entry node and ends with an exit node: these nodes are entered upon method invocation and method termination, respectively. Hence, the overall program control flow can be built out of the method control flow graphs by moving from one control flow graph to the other according to method invocation and termination[9].

Program analysis is carried out by relying on two groups of equations. The first group focuses on a given node in the control flow graph, and defines the relation between the information entering and exiting the node. This group of equations is sufficient to analyze a single path in the given program. However, a node in the control flow graph may have multiple incoming edges that represent different control flow paths, e.g., due to branches or loops, as shown in Figure 9. The second group of equations specifies precisely how the information coming from these different sources is merged at the entry point of a given target node n, by defining the relationship between the outgoing information associated to the sources of all the edges insisting on n, and the information effectively entering n.

Given these two groups of data-flow equations, the global solution can be computed by standard techniques like a *worklist* algorithm, typically used in data flow analysis to solve equation systems. In this algorithm, a representation of the work to be done (e.g., the computation of some variable properties through the program control flow graph) is stored in a worklist. The algorithm iteratively removes a task from the worklist and processes it, potentially causing the insertion of new tasks in the worklist. The algorithm proceeds until the worklist is empty and the least solution of data flow equations is found[10]. In other words, the difficult (and original) part of the analysis is not the solution of the equation system, rather in the definition of the relationship between nodes. In the remainder, to simplify the description we define the equations only informally.

Object aliasing. Our analysis is complicated further by *object aliasing*, i.e., the ability of Java to refer to the same object through different object references. Aliasing is a fact of life in object-oriented programming, and Java is no exception. This means that, to determine whether an object must be serialized, we need to keep track of how all of its aliases are used.

Fortunately, the aliasing problem is a well-known and thoroughly studied one in program analysis, and a number of suitable techniques exist (e.g., [11, 3, 22, 19, 12, 13]). Moreover, alias analysis is orthogonal to the type-based analysis we describe here, and the two can be combined straightforwardly as follows. First, we can exploit the results of alias analysis to annotate each node of the control flow graph with the alias set associated to each parameter attribute found in the concrete graph currently under consideration. Here, we do not discuss further the details of how to accomplish this, since it can be done straightforwardly by exploiting the aforementioned alias analysis techniques found in the literature. Then, when "decorating" the edges of the concrete graph under consideration, based on the walkthrough on the control flow graph, we need to be careful about changing the state of an edge not only when an attribute is being modified by a node of the control flow graph, but also when any of its aliases is.

In the sequel, we first describe how the analysis is performed on a single path, by defining how each instruction in the control flow graph of m affects the labeling functions *defined* and *skip* defined for that control path. Then, we explain how the functions *defined* and *skip* computed on different paths are merged when different paths of the control flow graph meet in a single node.

4.3.1 Analyzing a Single Path

To simplify the presentation, we assume that the input parameter p in the method invocations `o.m(p)` has a single concrete graph and is serializable, i.e., $p \in \mathcal{S}$. Moreover, we assume that all multiple-level reference expression such as `a.b().c()` and `a.b.c` are normalized into a sequence of two-level reference expression of the form `a.b()` and `a.b`, by using additional variables. For instance, `a.b().c()` can be split in `x=a.b(); y=x.c()`.

The analysis begins with an initial state where $\mathit{skip}(e) = \mathit{true}$ and $\mathit{defined}(e) = \mathit{false}$, $\forall e \in \mathcal{E}$, where $\mathcal{E}$ is set of edges belonging to the concrete graph of p. In other words, all the attributes of p are not defined and their serialization can be avoided.

We focus the discussion on a variable y being analyzed in the context of the execution of the given method m, where y is either represented in the concrete graph by some edge

[9]Exception-handling introduces additional implicit control transfers. However, these can be analyzed by using existing techniques (e.g., [20]) in conjunction with ours.

[10]The reader interested in further details on worklist algorithms can see for instance Chapter 6 of [15].

e such that $y = attr(e)$ with $e = (n_i, n_j, v)$ and $v = y$, or is an alias of the variable v represented by e. Note that when referring to an attribute y in the concrete graph we implicitly assume that there is no ambiguity, i.e., that there is only one type definition containing the attribute label y. The ambiguity can be removed by referring to the variable together with its type, at the only expense of clarity and compactness of the notation.

Essentially, we need to specify how the traversing of a given node of the control flow graph involving y affects the concrete graph, and in particular the labeling of its edges. The variable y can be affected by definitions and uses (in the common meaning of program analysis [23, 8]). *Definitions* of y are statements which assign a new value to y. *Uses* of y are all those situations where y's value (or one of y's attribute values) is used in an expression.

With these definitions, the data-flow equations can be expressed informally as follows:

- **Definition.** If *defined*(e) is already *true* before entering the node of the control flow graph containing the definition of y, nothing needs to be done, since y was already defined and the state of the concrete graph updated accordingly. Otherwise, the value of *defined*(e) is set to *true* for edge e and for all the outgoing edges of n_j, the target of e.

- **Use.** If *defined*(e) is already *true* before entering the node of the control flow graph containing the definition of y, nothing needs to be done. Otherwise, the value of *skip*(e) must be set to *false* in the concrete graph, since the value of y is needed in the execution of the method under analysis.

The rationale behind the above rules can be grasped by stating that if all the uses of y are preceded by a definition, or there are no uses, then the serialization of y can be avoided.

Note how the value of *skip*(e) is unchanged by a definition of y. If *defined*(e) was *false* before entering the node of the control flow graph, *skip*(e) must remain *true*, since it is safe (at this point of the analysis) to skip the serialization of the variable because its value is immediately reset by a definition. Similarly, if instead *defined*(e) was *true* before entering the node, nothing is changed by a definition. If we already decided that y must be serialized and *skip*$(e) = false$, e.g., because a previous use occurred, we cannot override this decision at this point of the analysis because the value of y will still be needed before this definition.

Attribute accesses and method invocations are an important kind of use. *Attribute accesses* are in the form[11] `y.x`, where x is an attribute defined in the class of y. An attribute access is a use of y, but it requires to consider, from this point on, not only the definitions and uses of y but also of x, to determine whether it is in turn serializable. *Method invocations* where y is involved can be either of the form `y.g(...)`, where y is the invocation target, or `g(y,...)`, where y is one of the actual parameters. Again, a method invocation is treated as a use, but it also requires the analysis of the method g whose execution involves y, that is performed by insisting on the concrete graph that has been labelled up to the invocation point.

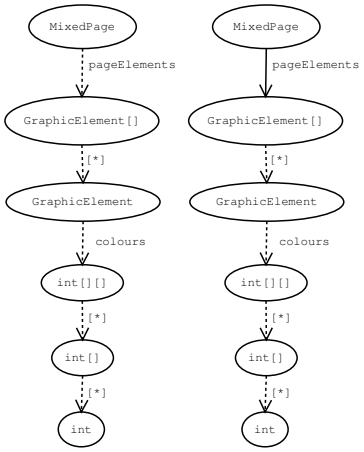
[11] As mentioned earlier, if y is an array a reference to one of its elements is treated like a reference to an attribute.

Example. Let us consider the printing application we described in Section 3, and let us focus on the remote invocation of the method `print` on an object of type `DotMatrixPrinter`, with a parameter `aPage` of type `MixedPage`, as we previously showed in Figure 2. For this case, we already built the concrete graphs in Figure 8, and the control flow graphs in Figure 9. For now, let us consider only the first concrete graph, related to `TextElement`, and let us walk through the control flow graph along the path defined by entering the body of all `if` and `for` statements.

Figure 10 shows the result of this analysis as a series of snapshots of the concrete graph as the control flow graph is analyzed. Note how the nodes of the control flow graphs in Figure 9 are numbered to reflect the order in which their analyzed in this specific walkthrough.

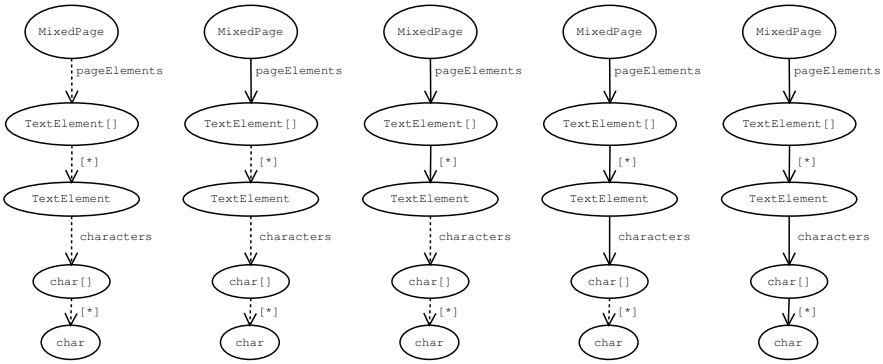
In the initial state, shown in Figure 10(a), all the attributes of the concrete graph are not defined and hence can be skipped during serialization. This graph is identical to the one in Figure 8(a). A dashed edge e means that the corresponding attribute can be safely skipped, i.e., *skip*$(e) = true$, while a solid edge means that the attribute must be serialized. In our example there are only uses and no definitions, thus it is safe to render graphically only the value of *skip*(e).

The analysis of the control flow graph begins in the entry point of `DotMatrixPrinter.print`, node 1 in Figure 9. Upon entering the first statement of `print` in node 2, the formal parameter `page` (and hence the actual parameter `aPage`) is used during the invocation of method `getTextElements`. Invocation of this method is analyzed by moving to the entry point of its control flow graph (node 3), but retaining the concrete graph labelled thus far. The traversal of node 4 leaves the concrete graph unaffected. On the other hand, node 5 contains a use of the attribute `pageElements`, through access to its attribute `length`. The edge corresponding to `pageElements` is then is marked as to be serialized, shown with a solid arrow in Figure 10(b). Since we set



(a) initial state (b) node 5

Figure 11: Decorating the concrete graph of Figure 8(b) while walking through the control flow graphs in Figure 9.



(a) initial state (b) node 5 (c) node 12 (d) node 14 (e) node 15

Figure 10: Decorating the concrete graph of Figure 8(a) while walking through the control flow graphs in Figure 9. The value of *skip* is *true* for dashed edges and *false* for solid ones.

out to enter every `for` loop, node 6 must be considered next. This node is particularly interesting since it gives us the opportunity to consider how the analysis must proceed in presence of the language construct `instanceof`. One could think that, since `pageElements[i]` appears as an argument of this instruction, this constitutes a use of the variable. Instead, it must be observed that the result of `instanceof` does not depend at all on the *value* of `pageElements[i]`, but only on its *type*. Moreover, the value of this variable is left unaffected by the execution of `instanceof`. Hence, the traversal of this node leaves the concrete graph unchanged. Note also that in this case we are *forced* to go through node 7 instead of choosing the `else` branch and return to node 5. In fact, choosing this latter path would be a violation of the previous assumption about the type of the elements of `pageElements`.

The remaining two nodes of the method `getTextElements` do not affect the concrete graph directly, but establish the object aliases that enable further changes effected by the other methods. Node 7 establishes an alias between an attribute of the concrete graph, i.e., an element of `pageElements`, and an element of the local array `text`. Node 8 propagates this alias back to the method `print`, by returning `text` as a result value. Node 9 brings the control back to the `print` method, that resumes from node 10. Nodes 10 and 11 do not affect the concrete graph, since they contain only uses of `text`. Instead, node 12 contains a use of an element of `text`, which is potentially aliased to one of `pageElements`. Hence, the corresponding edge in the concrete graph must be marked accordingly, as in Figure 10(c). The use of `text[i]` is a method invocation, which causes the analysis to move to the control flow graph of the method `print` in `TextElement`.

The first statement of this method, corresponding to node 14, contains a use of the array `characters` which, by virtue of aliasing, is an attribute of the element of `pageElement` aliased to the invocation target `text[i]`. Hence, `characters` must be serialized (Figure 10(d)). Finally, node 15 contains an invocation of the method responsible for printing an element `characters[i]`. Although here we do not show the code of this method, it intuitively relies on the input parameter, which then needs to be serialized, leading to the last and final concrete graph in Figure 10(e).

According to this analysis, the whole object graph rooted at the parameter `aPage` must be serialized—at least for concrete graph and control path we considered. This matches the intuition associated to our example, where all the information associated to a text page is effectively used by a dot-matrix printer.

It is instructive at this point to see what happens if the concrete graph of Figure 8(b) is considered instead, when walking through the same control flow graphs in Figure 9. The corresponding snapshots of the analysis are shown in Figure 11. Again, the initial state is represented by the first graph, where all the edges are dashed. Up to node 6, the analysis proceeds like in the previous case, by requiring the attribute `pageElements` to be serialized. The test in node 6, however, forces us to choose a different path, returning to node 5. In fact, as we mentioned earlier, proceeding to node 7 would violate the assumption we are making when considering this concrete graph, i.e., that the elements of `pageElements` are of type `GraphicElement`. The rest of the analysis proceeds through nodes 5, and 8 to 17. However, since no alias has been established between `text` and some attribute of the concrete graph graph, the latter remains unchanged. Hence, the graph in Figure 11(b) shows the outcome of the analysis, which again confirms the intuition about our example: the serialization of an element of `pageElements` whose type is `GraphicElement` can be safely skipped, while the array `pageElements` must be serialized anyway.

4.3.2 Merging Information from Multiple Paths

What we described thus far is sufficient to analyze methods whose code does not contain branches in the control flow. Otherwise, we need to specify how the information collected through separate control paths is reconciled when the control paths are rejoined.

In our case, the information that we need to reconcile is the labelling of edges of the concrete graph, i.e., the value returned by the functions *defined* and *skip*. The problem is that an attribute y in the concrete graph may have been recorded as defined ($\mathit{defined}(e) = \mathit{true}, y = \mathit{attr}(e)$) through

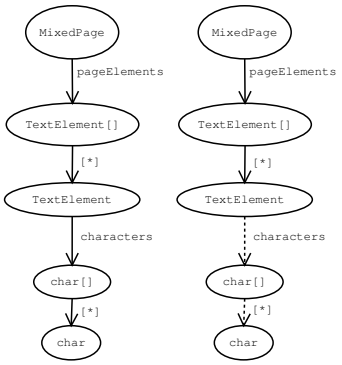


Figure 12: The result of the analysis in Figure 10 (left) and the result of the analysis performed by following a different path on the control flow graph (right).

one control path, and not defined in another. Even more important to the outcome of the analysis, the same attribute may have been deemed necessary to the enclosing method, and hence marked as to be serialized ($skip(e) = false$) along one path, and marked as to be skipped along another.

Clearly, to preserve a correct program behavior we need to take the most conservative stand. In the aforementioned case we need to preserve, in the node where the control flow rejoins, the values $defined(e) = false$ and $skip(e) = false$. In other words, an attribute is defined in the joining node if it was defined through all of the joining paths, and similarly it can be safely skipped during serialization if it can be skipped through all the joining paths.

To express the rule in a formal way, we simply need to set the value of the labelling functions *defined* and *skip* in the join point to be the logic conjunction of the values of all the functions $defined_i$ and $skip_i$ computed along an incoming path i, that is, $\forall e \in \mathcal{E}$:

$$defined(e) = \bigwedge_{i=1}^{n} defined_i(e),$$

$$skip(e) = \bigwedge_{i=1}^{n} skip_i(e)$$

Once the data-flow equations are given, the analysis is completely defined and the least solution can be computed by a working list algorithm, as we mentioned earlier. Obviously, the analysis must be performed for each method that can be invoked remotely, for each serializable object parameter, and for each of the possible concrete graphs of such parameter.

Example. Figure 12 compares the result of the analysis we performed previously (Figure 10(e)) with the result of the analysis performed using the same concrete graph but choosing a different control flow path in node 10. Since the most conservative annotation of the concrete graph prevails, the final result of the analysis is the one shown on the left. It confirms that the whole object graph must be serialized in this case. In the case where a `GraphicElement` is considered instead, all the alternative control flows leave the concrete graph unchanged, thus confirming that serialization of the graphic page elements is always unnecessary when a dot-matrix printer is exploited, independent of the control flow followed on the server.

4.3.3 From Types to Objects

The analysis of the control flow graph terminates by returning the definition of *skip* for all the concrete graphs under consideration. This allows one to determine, for each concrete graph, whether a given attribute should be serialized. Nevertheless, to actually implement our optimization and redefine serialization we need to link the type information stored in the annotated concrete graphs with the particular object graph rooted at the parameter involved in a remote method invocation.

From an abstract point of view, the required steps can be summarized easily. The nodes of the object graph associated with the parameter contain the actual field objects, while edges represent the references established among them. One can navigate through the object graph, starting at the root, and mark each object as serializable or transient by looking at the information stored in the concrete graph that matches the parameter.

In practice, however, things are slightly more complicated by the details of Java serialization. For instance, the object graph can be constructed only at run-time, and the Java API does not provide any direct way to obtain it.

In the next section, we show a way to implement our optimization, without being concerned with efficiency. Instead, our main concern is to show that an implementation path indeed exists, and to illustrate through a clean design the steps required by our optimization. In doing this we choose to rely only on the mechanisms provided by the language API: more efficient solutions can be devised if modifications to the JVM are allowed.

5. REDEFINING SERIALIZATION

The information derived from the analysis we described in Section 4 can be summarized as a three-dimensional matrix, whose elements are identified by:

1. the type S of a remote server;
2. the method m that is being invoked on S;
3. the concrete type of a serializable parameter p being passed to m.

The value of a matrix element is the list of attributes of p whose serialization can be safely skipped during the corresponding remote invocation.

This information can be made available at run-time through a (static) object of class `InvocationData`. Part of the interface of this class is shown in Figure 13. The field `attributes` contains the aforementioned three-dimensional matrix that we assume pre-loaded with data through mechanisms whose details are irrelevant here. The (static) fields `targetClass` and `targetMethod` are instead set at run-time right before the corresponding remote invocation. This requires a straightforward instrumentation of the source code. An example is shown in Figure 14, where the original client code of Figure 2 is instrumented to determine through reflection the class and method involved in the invocation, and set them appropriately in the `InvocationData` object.

Once the invocation target and the invoked method are known, the next step is to intervene on the way serialization is performed, to skip the appropriate attributes. As we

```
public class InvocationData {
  private static Class targetClass;
  private static Method targetMethod;
  private static HashSet [][][] attributes;
  public void setInvokedMethod(Class targetClass,
                               Method targetMethod) {
    this.targetClass = targetClass;
    this.targetMethod = targetMethod;
  }
  public HashSet getNonSerializableFields(Class c) {
    int x = targetClass.hashCode();
    int y = targetMethod.hashCode();
    int z = c.hashCode();
    return attributes[x][y][z];
  }
}
```

Figure 13: The class `InvocationData` containing the results of the analysis, which are retrieved at runtime to optimize serialization.

```
...
IPage aPage = new MixedPage(20);
IPrinter printer =
  (IPrinter) Naming.lookup("//myregistry/printer");
// BEGIN INSERTED CODE
Class[] methodParameters = {aPage.getClass()};
InvocationData
  .setInvokedMethod(print.getClass(),
                    print.getClass()
                         .getDeclaredMethod("print",
                                            methodParameters));
// END INSERTED CODE
printer.print(aPage);
...
```

Figure 14: Instrumenting the remote method invocation of Figure 2.

mentioned in Section 2.1, a class can define its own serialization by defining the private methods `writeObject` and `readObject`. Hence, we need to provide an appropriate definition of these methods for all the classes whose serialization must be optimized. Again, this step can be performed trivially through a precompilation step.

The code for these two methods is shown in Figure 15. It relies on a feature introduced since Java 1.2, which allows one to declare in a class a (private, static, and final) field `serialPersistentFields`, which contains the names and types of fields that must be serialized. This can be regarded as an alternative and complementary way of declaring `transient` fields. Again, the content of this field can be set in the source code through a straightforward precompilation step, by using the result of our analysis.

From the code in Figure 15, however, it can be seen how the methods behave differently depending on whether the field being (de)serialized is an object or a primitive type. In the former case, all the object fields that are specified as serializable by the programmer are effectively inserted in the serialization stream. However, for all those fields that should not be serialized according to our analysis, i.e., for all the fields whose name is returned by `InvocationData.getNonSerializableFields`, the value `null` is inserted in the stream in place of the real object. A dual behavior is used for primitive types. In fact, all the primitive types are removed from `serialPersistentFields`, and hence set to `transient`, independent of what the programmer originally specified, and hence are not automatically inserted in the serialization stream. In this case, however, the value of fields that are *not* returned by `getNonSerializableFields` is explicitly inserted in the serialization stream together with its index, so that the corresponding deserialization step is able to restore the value found in the stream to the appropriate field.

```
private void writeObject(ObjectOutputStream out)
            throws IOException {
  try {
    ObjectOutputStream.PutField fields = out.putFields();
    HashSet skipFields =
      InvocationData.getNonSerializableFields(this.getClass());
    for (int i=0; i<serialPersistentFields.length; i++) {
      String name = serialPersistentFields[i].getName();
      if (skipFields.contains())
        fields.put(name, null);
      else
        fields.put(name,
                   getClass().getDeclaredField(name).get(this));
    }
    out.writeFields();
    Field[] allFields = this.getClass().getDeclaredFields();
    for (int i=0; i<allFields.length; i++)
      if (!notSerFields.contains(allFields[i].getName())) {
        out.writeInt(i);
        if (allFields[i].getType() == Boolean.TYPE)
          out.writeBool(allFields[i].getBoolean(this));
        else if
          ... and similarly for all the other primitive types ...
      }
  } catch(Exception e) { e.printStackTrace(); }
}
private void readObject(ObjectInputStream in)
            throws IOException {
  try {
    ObjectInputStream.GetField fields = in.readFields();
    for (int i=0; i<serialPersistentFields.length; i++) {
      String name = serialPersistentFields[i].getName();
      getClass().getDeclaredField(name).set(this,
                                      fields.get(name, null));
    }
    Field[] allFields = this.getClass().getDeclaredFields();
    int current = in.readInt();
    for (int i=0; i<allFields.length; i++)
      if (current == i) {
        if (allFields[i].getType() == Boolean.TYPE)
          allFields[i].setBoolean(this, in.readBool());
        else if
          ... and similarly for all the other primitive types ...
      }
  } catch(Exception e) { e.printStackTrace(); }
}
```

Figure 15: The methods `writeObject` and `readObject` implementing our optimization.

The reason for processing differently object and primitive types lies in the desire of reusing the built-in Java mechanisms. If we were to set all the object fields as `transient`, we would need to explicitly deal with object serialization ourselves, and hence redefine the serialization process completely. While this is indeed a viable approach, it places a bigger burden on the precompilation step.

As we mentioned earlier, surely the implementation we sketched here is not the most efficient. A faster solution would be to hard-wire directly in the code the necessary steps. Anyway, it is interesting to note how the solution we presented is much more flexible. If the results of the analysis change as a consequence of a code modification, in a hard-wired implementation all the classes affected must be recompiled, while here only `InvocationData` needs to be changed accordingly.

6. DISCUSSION

In this section we discuss improvements and limitations of our technique, as well as its exploitation.

Reducing the number of concrete graphs. To determine the possible types of a parameter p we relied on the inheritance hierarchy, i.e., on static type information. Nevertheless, it is possible that, in the context of a given invocation, the legal dynamic types of a given object are only a subset of those calculated conservatively by using static information. Minimizing the number of types to be considered has a strong impact on performance, since it minimizes the number of concrete graphs to be annotated during the analysis of the control flow. The necessary type information can be computed by performing a preliminary phase using a *concrete type inference* [18]. We are currently considering if and how this technique can be integrated in our approach.

Semi-static analysis. Analyses like those described in this paper are typically performed entirely statically, and indeed this is the way our approach works thus far. The reason is that the computational load of this kind of analysis is usually too high to be placed on the run-time system, especially since most of the approaches in the literature are meant to optimize execution speed.

Nevertheless, in our case we aim at reducing the bandwidth utilization. Hence, in some cases (e.g., in mobile environments with low-bandwidth connectivity) it might be reasonable to trade computation for bandwidth, and perform some if not all of our analysis at run-time.

The advantage of this approach lies in the accuracy of information about the program that becomes available at run-time. For instance, if the analysis were to be performed right upon a remote method invocation there would be no need to consider *all* the possible combinations of concrete graphs for a given parameter and control flow of the possible servers. Considering the single concrete graph matching the parameter being passed and the specific server target of the invocation would be sufficient. Hence, while on one hand there is a computational overhead to be paid at run-time, this overhead would arguably be significantly smaller than the one to be paid by an entirely static analysis. We are currently elaborating and experimenting with this idea, to determine quantitatively the tradeoffs involved.

Closed vs. open world. Thus far, we implicitly assumed that the whole code base of the distributed application is available to the analysis, and it is not going to change after the code is deployed. This *"closed world"* scenario is reasonable for a non-negligible number of distributed applications. On the other hand, RMI was designed to support an *"open world"* scenario where instead the code base of the application can change dynamically and seamlessly, by virtue of encapsulation and mobile code.

In this scenario, our analysis is no longer applicable as is. In fact, let us consider our reference scenario, and let us assume that the interface `IPrinter` exports an additional method `getPage`, which returns the page currently being printed. This method can be invoked by a client C_2, different from the client C_1 that required the page printout. No assumption can be made in general about the use C_2 makes of the page. For instance, regardless of optimizations made by our approach, C_2 might require the serialization of the entire page as originally stated by the programmer. The reason could be to implement a persistency service that stores for a given period of time all the processed pages to cope with temporary malfunctioning of the printer, thus avoiding C_1 to reissue a print request.

Now the question is whether the code of C_2 is available at the time of the analysis. If yes, then the need to serialize all the fields of a page is discovered when the analysis is run on the remote method invocation of `getPage` issued by C_2, and no modification is required to our approach. Instead, if the setting is such that the code of the clients that may invoke `getPage` is not available, our analysis must be modified accordingly, otherwise the application may experience an incorrect behavior. For instance, a monitoring client independently developed to visualize on screen the whole content of the page currently being printed on a dot-matrix printer would fail to display it correctly because, as a result of our optimization, only the textual elements of the page are present on the server.

We are currently working on an extension of our analysis that encompasses also an open world scenario. This is achieved by exploiting escape analysis [4], a static analysis originally developed to optimize memory allocation and object synchronization by determining *i)* whether the lifetime of an object can exceed the lifetime of the method where it is created, and *ii)* whether an object created by a thread can be accessed by another.

In our context, we are currently adapting escape analysis to determine whether a given object cannot *escape* the code base that is known at analysis time. In this case, the result of our analysis is still valid as is, since an object that has been only partly serialized is never passed outside the boundaries of the analyzed code. Instead, if an object escapes such boundaries no assumption can be made about its use. All we can do is to warn the programmer about fields not being accessed by an invocation occurring in the analyzed code, and demand to her the ultimate choice about whether these fields must be serialized or not.

Moreover, we are also investigating whether the aforementioned semi-static approach can provide some benefit when applied in an open world context.

Other possible uses of the analysis. Thus far, we assumed that our analysis is exploited to optimize parameter serialization on a per invocation basis. This remains the primary motivation for our approach, and the natural exploitation path. Nevertheless, our analysis can be used in other, slightly different ways.

For instance, programmers often overlook serialization, since it is enabled by default on all the fields of a serializable object. This is true especially during the initial phases of development, when the fine tuning of serialization is often deferred to a later, more mature stage of the product. Nevertheless, in complex applications it is often difficult to gain a complete understanding of which fields can safely be left `transient`. The fact that subtyping in Java preserves serializability[12], makes matters worse. In this context, our analysis can be exploited to warn the programmer that a given field is *never* accessed by an invocation target, and

[12]In contrast, in .NET serializability of a type is not propagated downwards along the inheritance hierarchy.

hence in principle can be safely declared as `transient`.

7. RELATED WORK

The approaches to RMI optimization we were able to find in the literature are all focused on optimizing the computational overhead of serialization, rather than its bandwidth consumption. Not surprisingly, most of these approaches aim at optimizing RMI in the context of scientific applications exploiting parallel computing, where computational efficiency is a particularly strong concern. Hence, to the best of our knowledge no published research has tackled directly the problem of using program analysis to reduce the traffic overhead of serialization. As a consequence, no other approach is directly comparable to ours.

Krishanswamy et al. [10] reduce the computational overhead on the client side by exploiting object caching. For each call, a copy of the byte array storing the serialized object is cached to be reused in later calls—provided that the object has not changed in the meantime. Braux [1] exploits static analysis to reduce the computational overhead of an RMI call due to the reflective calls necessary to discover the dynamic type information. The work of Kono and Masuda [9] relies on the existence of run-time knowledge about the receiver's platform, and redefines the serialization routine to exploit this information. On the sender, the object to be serialized is converted directly into the receiver's in-memory representation, so that the receiver can access them immediately without any data copy and conversion. Breg and Polychronopoulos [2] explicitly target homogeneous cluster architectures, and provide a native implementation of a subset of the serialization protocol. Their approach leverages on knowledge about the layout of data structures on the cluster machines, so that complex data structures are encoded directly in the byte stream by using only a minimal amount of control information. Philippsen et al. [17] integrate various approaches to obtain a slightly more efficient RMI implementation. They simplify the type information encoded in the serialization stream, improve the buffering strategies for dealing with the stream, and introduce a special handling for floats and double. Nevertheless, their optimizations are again closely tied to the parallel computing domain.

Another line of research that is somewhat related to ours exploits program analysis techniques to find the best way to transform a monolithic, centralized application in a distributed one by relying on the middleware as is, rather than optimizing it. Examples of this kind of research are the Pangea [21] and J-Orchestra [25]) systems, relying on Java RMI, and the Coign system [7] that instead relies on DCOM.

8. PROTOTYPE

We are currently developing a prototype suite of tools supporting our technique. The overall architecture is shown in Figure 16.

The main component of the architecture is the *analyzer*, which receives as input the Java source code of the application and outputs the result of the analysis by recording information about each remote method invocation and the corresponding annotated concrete graphs. This information is stored in a binary format for the sake of compactness. Our analyzer is currently developed using JABA [6], an API supporting program analysis of Java bytecode.

The result of the analysis can be input to one of the three

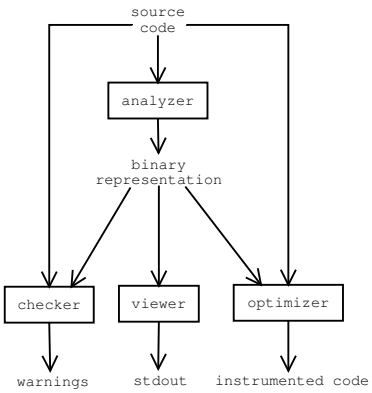


Figure 16: Exploiting our analysis through software engineering tools.

tools at the bottom of Figure 16. The *viewer* enables the visualization of the analysis result in a human readable format. Many implementation of the viewer are possible, ranging from a simple filter that outputs ASCII text, to a sophisticated GUI-based tool, possibly integrated with the debugger of an integrated development environment.

The optimizer instruments the source code based on the results of the analysis, as explained in Section 5. The output of this component is the original program with enhanced serialization behavior.

Finally, the serialization *checker* tool on the left allows to detect fields declared as serialized but never used in the program, as mentioned in Section 6.

9. CONCLUSIONS AND FUTURE WORK

In this work we presented a novel program analysis technique that aims at optimizing parameter serialization in remote method invocations on a per-invocation basis. The analysis identifies which portion of a parameter is actually used on the receiving side, and its results can be used to redefine the serialization mechanism to reduce the runtime communication overhead. In this paper we defined the problem, motivated our contribution, presented the program analysis technique, illustrated a way to implement the runtime optimization that relies only on the Java API, and briefly reported about the ongoing development of a prototype tool suite supporting our approach.

The latter is the focus of our current research work on the topic described in this paper. In fact, the tool will give us the possibility to demonstrate the feasibility of our approach and, more importantly, evaluate quantitatively its effectiveness in optimizing real-world applications developed with RMI. Meanwhile, we are further refining our analysis technique, in particular exploring the idea of performing part of the static analysis at run-time.

10. REFERENCES

[1] M. Braux. Speeding up the Java Serialization Framework through Partial Evaluation. In *Proc. of the Workshop on Object technology for Product-line Architectures*, 1999.

[2] F. Breg and C.D. Polychronopoulos. Java Virtual Machine Support for Object Serialization.

Concurrency and Computation: Practice and Experience, 2001.

[3] M. Burke et al. Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers. In *Languages and Compilers for Parallel Computing: Proc. of the 7th Int. WS*, LNCS 892, pages 234–244, 1995.

[4] J.-D. Choi et al. Escape Analysis for Java. In *Proc. of OOPSLA'99*, pages 1–19. ACM Press, 1999.

[5] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, 2000.

[6] Aristotle Research Group. JABA: Java Architecture for Bytecode Analysis. `www.cc.gatech.edu/aristotle/Tools/jaba.html`.

[7] G. Hunt and M. Scott. The Coign Automatic Distributed Partitioning System. In *Operating Systems Design and Implementation*, pages 187–200, 1999.

[8] K. Kennedy. A Survey of Data Flow Analysis Techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 1, pages 5–54. Prentice-Hall, 1981.

[9] K. Kono and T. Masuda. Efficient RMI: Dynamic Specialization of Object Serialization. In *Proc. of the Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 308–315, 2000.

[10] V. Krishnaswamy et al. Efficient Implementations of Java Remote Method Invocation. In *Proc. of the 4th USENIX Conf. on Object-Oriented Technologies and Systems (COOTS'98)*, pages 19–36, 1998.

[11] W. Landi and B. Ryder. A Safe Approximate Algorithm for Interprocedural Aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, July 1992.

[12] D. Liang et al. Extending and Evaluating Flow-insensitive and Context-insensitive Points-to Analyses for Java. In *ACM SIGPLAN - SIGSOFT WS on Program Analysis for Software Tools and Engineering*, pages 73–79. ACM Press, 2001.

[13] V. Martena and P. San Pietro. Alias Analysis by Means of a Model Checker. In *Proc. of 10th Int. Conf. on Compiler Construction*, LNCS 2027, 2001.

[14] Sun Microsystems. Java Remote Method Invocation Specification, 2002.

[15] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[16] OMG. The Common Object Request Broker: Architecture and Specification, Revision 3.0, 2002-06-01. Technical report, OMG, July 2002.

[17] M. Philippsen et al. More Efficient Serialization and RMI for Java. *Concurrency—Practice and Experience*, 12(7):495–518, 2000.

[18] J. Plevyak and A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *OOPSLA '94 Conf. Proc.*, pages 324–340, 1994.

[19] M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *Proc. of the Symp. on Principles of Programming Languages*, pages 1–14, 1997.

[20] S. Sinha and M.J. Harrold. Analysis of Programs That Contain Exception-Handling Constructs. In *Proc. of Int. Conf. on Software Maintenance*, pages 348–357, Nov. 1998.

[21] A. Spiegel. Automatic Distribution in Pangaea. In *Proc. of the Workshop on Communications-Based Systems*, 2000.

[22] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *Proc. of the Symp. on Principles of Programming Languages*, pages 32–41, 1996.

[23] M. Suedholt and C. Steigner. On Interprocedural Data Flow Analysis for Object-Oriented Languages. In *Proc. of the Int. Conf. on Compiler Construction*, LNCS 641, 1992.

[24] T. L. Thai and H. Lam. *.NET Framework Essentials*. O'Reilly, June 2001.

[25] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java Application Partitioning. In *Proc. of ECOOP Conf.*, 2002.

[26] J. Waldo et al. A Note on Distributed Computing. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, LNCS 1222. Springer, April 1997.