vrije Universiteit amsterdam

Runtime serialization code generation for Ibis serialization

revision 1.1

Pavel Petřek

Thesis submitted for the Masters Degree at the
Vrije Universiteit

· 2006 ·

Abstract

Java programs consist of classes stored in class files. Using a bytecode manipulating framework, we can simply modify (rewrite) any of the class files. This rewriting approach was chosen at the Vrije Universiteit for the Ibis project as a way of implementing a highly efficient serialization mechanism. The rewriting is performed at compile time. In this thesis, we examine if we can move the rewriting process into runtime. We also provide an implementation of this approach. To achieve high performance rewriting, we attempt to use a more efficient technique than the current compile-time solution. We show that the runtime rewriting approach provides significantly faster start-up time of the application compared to the compile-time solution.

Acknowledgements

This thesis has been produced by drawing upon the effort of a number of people. I wish to thank all of them, especially Jason Maassen and Rob van Nieuwpoort for their ideas that led me to the successful completition of this project and for the time that Jason has spent advising me. I would also like to thank all my interested friends, especially Olga Sivková, Kassidy Clark, Katar Abed and Sebastian Abdallah for their strong moral support and significant help with the proper grammar and wording of this document.

Pavel Petřek
May 2006

Contents

Chapter 1

Introduction

1.1 Motivation

The Java programming language is a powerful object-oriented language that offers programs that are executable on almost any platform in the world of information technologies. It also offers a very rich API that makes the programmer's life easier, so he does not have to think about, for instance, how to implement a hashtable for storing data or how to create a zip archive. Java is very popular. Java programs can be found on enterprise servers as well as on home desktop computers and large cluster systems.

One of Java's aims is to efficiently support distributed applications, e.g., using remote method invocation (RMI). Implementing distributed application requires that Java objects can be converted to a form in which that can be transmitted over the network. This process is called serialization. Sun's standard implementation always scans all fields of an object and converts their values into an array of bytes. This array of bytes is then sent as a serialized object. Reflection is another term for the scanning of the class. Reflection of the fields to serialize is done at runtime. Even if we serialize a thousand instances of one class, the reflection is always done again. This drawback can be solved by extending the class with methods that directly write and read the values of the fields. Reflection has to be performed only once. It can be done at compile time or load time before the class is used in a Java virtual machine (JVM)[1].

1.2 The Ibis project

Due to a wide range of supported platforms, Sun[2] engineers selected a trade-off underlying network protocol for remote objects technology. The protocol

[1] Java Virtual Machine - interpreter of Java bytecode which is the code of a compiled Java program

[2] Sun Microsystems created the Java concept and introduced its first implementation

is TCP and thus the RMI model based on TCP is the default way of distributed programming in Java.

Java offers RMI/TCP as a generic solution for network programming. Unfortunately, it is not efficient for all platforms. High-performance systems offer communication interfaces aimed at speed and/or latency. These interfaces use special protocols since using TCP causes a loss of available performance. Large clusters often use high-speed network cards (e.g., Myrinet) and using these interfaces leads to more efficient communication than using traditional ethernet based TCP/IP communication. This is one of the reasons why a group from Vrije Universiteit started a project called *Ibis* which is aimed at offering more than pure RMI/TCP for efficient Java based network programming. Ibis offers an entire framework that can be implemented on top of various network protocols. It also offers models other than RMI, such as group method invocation (GMI) or the divide and conquer model called *Satin*. Additionally, due to its modular design, Ibis offers an option of adding support for other protocols or other programming models.

Generally, we can say that Ibis took known environments and combined the major benefits of them. This results in a portable, flexible and highly efficient solution [6].

One of the important targets of the Ibis project is grid computing, especially distributed supercomputing applications. Grid computing, as described by Wikipedia [7], is an emerging computing model that provides the ability to perform higher throughput computing by taking advantage of many networked computers to model a virtual computer architecture that is able to distribute process execution across a parallel infrastructure. Grids use the resources of many separate computers connected by a network (usually the Internet) to solve large-scale computation problems. The reality is that not only separate computers can be connected together, but several clusters of computers can be interconnected (for example using the Internet) to create a high performance grid.

As we may expect, on separate computers or even on clusters, shared files or a shared file system can be used, but as we start to talk about a grid, IO operations related to the sharing of the file systems become too expensive. This is an important fact that led to certain decisions being made during the development of the solution proposed as a part of this project.

The Ibis project introduced the notion of rewriting those class files, which contain serializable Java classes, and storing them at their original location. The JVM then uses the classes in this rewritten form and objects are then serialized without requiring runtime reflection. This rewriting can be very expensive and even though it is possible to rewrite core classes that are stable, it is very inconvenient to frequently rewrite user classes that are, for example, being debugged. One can say that a program can be rewritten once it is debugged, but there is still the fact that programmer is only human and can forget to rewrite the program's classes, so the performance would

be significantly lower. To solve this problem we will propose runtime class rewriting in this thesis. Moreover, using the runtime approach on a grid, we can move the rewriting to a grid node and thus the first build of the Ibis application will also be much faster.

1.3 The goal of the thesis

The basic goal of the thesis is to examine whether the current compile-time rewriting can be replaced by runtime rewriting using a customized class loader. We will attempt to develop such a class loader.

Ibis currently uses the BCEL framework (Byte Code Engineering Library) for compile-time rewriting. Unfortunately, due to a high level of abstraction, BCEL is inefficient. Currently, a parsed class is stored as a tree structure containing detailed information, e.g., fields, methods and code. This approach has the advantage that all class information is accessible and modifiable at any time, but the approach also causes memory and processing overhead. Moreover, BCEL stores all inter-class dependencies whose maintenance also reduces efficiency.

In this thesis we will examine if it is possible to replace BCEL with a more efficient tool, namely ASM (ASM bytecode framework). ASM uses a visitor approach. Information from the class file is traversed and, instead of storing, ASM passes the information directly to an appropriate method of an object called a Visitor. A Visitor is given to the parsing mechanism by the programmer. If the programmer does not need particular parts of the class, he can simply skip them by implementing an empty method.

To prove that the use of another framework for rewriting is more efficient, we will attempt to implement class loaders using both tools, BCEL and ASM. We will perform measurements of memory usage and time efficiency of both approaches. Observations based on the measured values as well as the actual values will be presented as a part of this thesis.

1.4 Structure of the thesis

Chapters 2 and 3 introduce all background knowledge required to fully understand the mechanisms covered in this work. They give a detailed description of serialization as it is implemented by Sun and Ibis. Grid computing and the *Ibis project* are briefly introduced. Chapter 2 also describes class loading in Java with a brief preview of the class file structure. Finally, the BCEL and ASM frameworks are introduced.

Chapter 4 explains our implementation, decisions that had to be made, along with their reasoning and also the limitations of the solution.

Chapter 5 compares the performance of the ASM and BCEL implementations.

Chapter 6 concludes the thesis by briefly describing the benefits of this project and covering possible extensions to the work.

Chapter 2

Background and related work

2.1 Class loading

With the term *class loading* we mean the creation of the class or interface C denoted by the name N in the internal structures of the JVM. This internal representation, as well as its construction, is implementation dependent. The class loading itself is implicitly triggered when another class refers to the class C through its constant pool. It is also possible to invoke the class loading explicitly in the program, e.g., using the *classLoad* method of a class loader.

A class or interface can be loaded from any array of bytes that satisfies the specification of the class file format[1]. Even when a class is loaded from a file stored on a drive, the entire file is first loaded into memory as an array of bytes and then class loading is called on this array.

There are two types of class loaders: bootstrap and user-defined loaders. The bootstrap class loader is used directly by the JVM core and loads the classes required for the primordial start of the program, e.g., *java.lang.Object*, *java.lang.Class*, *java.lang.String*, etc. The user-defined loader is represented by a subclass of the general *java.lang.ClassLoader* class and is responsible for loading the classes in a special way, e.g., by loading it from the network or by constructing it from the source code on the fly.

Class loaders can delegate requests for the loading of a class or interface C. If the class loader L does not delegate the loading of the class to another class loader, but defines it directly, we say that L is the defining class loader of C, or equivalent, that L defines C. If L delegates the loading of C or L defines C, we say that L is the initiating loader of C. A pair of the fully qualified names N of C and its defining class loader L is then the unique run-time identification of C in the JVM. For the class with a fully qualified name N and its initiating loader L_i we use the notation N^{L_i}.

The loading of C can be triggered by another class or interface D. This

[1]For detailed specification of the class file format see [3], Chapter 4

occurs when D refers to C through its constant pool, e.g., it has a field of the type C, its method returns the type C or the parameter of its method is of the type C. D can also trigger the loading of C through invoking certain methods from Java class libraries, such as the reflection methods. If C represents a non-array class or interface, the class loader used for D is also used for the loading of C. This can be a bootstrap loader or a user-defined loader. If C represents an array class, the class is defined directly by the JVM, but to load the class representing an element of C, the class loader of D is used and this class loader is also considered as the initiating loader of C.

The loading of C using the bootstrap class loader works as follows. First, the JVM determines whether the bootstrap loader was already registered as an initiating loader of C. If so, the loading is finished. Otherwise, we can expect one of the following. Either the bootstrap loader tries to load the data representing C stored in a platform-dependent manner (usually as a file) and then defines C by using the JVM's internals[2], or it delegates loading to some user-defined class loader.

The loading of C using the user-defined class loader L works as follows. First, the JVM determines whether L was registered as an initiating loader of C. If so, the already loaded class is returned and the loading is finished. Otherwise, the *loadClass* method of L is called in order to load C. The *loadClass* method can now build an array of bytes representing the class file of C and call the *defineClass* method on that. The *loadClass* method can also delegate the loading to another class loader L', typically by calling the *loadClass* method of L'. The *defineClass* method uses the JVM internals to build the internal representation of C. It is forbidden to call the *defineClass* method for any class from the *java.* package.

The loading of the array class C using loader L works as follows. The JVM determines whether L was already registered as an initiating loader of an array class with the same element name as the element of C. If so, that array class is returned and the loading is done. Otherwise, if the element is a reference, L is recursively used to load it. Then, for the element type and a dimension of the array determined from the name of C, the JVM initiates C. If the element of C is of the reference type, L is used as an initiating loader of C. For basic element types, the bootstrap loader is the initiating loader of C.

2.1.1 Loading constraints

An important issue in class loading is loading constraints. From the previous text of this section, it is clear that there can be two runtime instances of one class or interface name loaded by two different class loaders. This works

[2]See JVM Specification [3], Section 5.3.5

until these two instances affect each other. For example, if we try to assign an object of a class of the name N initiated by the class loader L to the field of a class with the same name N but initiated by another class loader L', we will fail. More specifically, the loading constraints are violated if, and only if, all of the following four conditions are met:

- There exists a loader L such that L has been recorded by the Java virtual machine as an initiating loader of a class C named N
- There exists a loader L' such that L' has been recorded by the Java virtual machine as an initiating loader of a class C' named N
- The equivalence relation defined by the (transitive closure of the) set of imposed constraints implies $N^L = N^{L'}$
- $C \neq C'$

Comprehensive details on class loading constraints can be found in [2].

2.2 Class file

For a better understanding of the following introduction of byte manipulation frameworks, it is useful to have some general knowledge of the structure of a Java class file as specified in [3].

Generally, a class file consists of:

- Header containing the version information of the file
- Constant pool containing the list of constants identified by their indexes in this list
- Access rights descriptor containing information about modifiers used for this class
- Superclass of this class or interface (if the class file realizes interface, this information must always be *java.lang.Object*)
- Implemented interfaces list specifying all directly implemented interfaces (not those implemented by transitive closure)
- Field list containing a record consisting of name, descriptor, access rights for each field and a set of attributes (for instance, the initial value of the static field)
- Method (plus constructors and optional static method) list containing records consisting of the name, descriptor, access rights for each method and set of attributes (for instance, exceptions thrown by the method or the code of the method)

- Class attribute list that contains general information about the class or interface (for instance, the source file)

All textual descriptors and names are realized by indexes of the constant pool items, which contain the actual values. This is also used for instructions that use textual descriptors. These instructions contain the index of the required constant pool item.

Internal descriptors of the fully qualified class or interface names use a slash ('/') symbol instead of ASCII periods ('.'). For example *java.lang.String* will be described as *"java/lang/String"*.

The internal descriptor of a field distinguishes three types:

- Basic type where the type is represented by the respective letter (*B*, *C*, *D*, *F*, *I*, *J*, *S*, *Z* for *byte*, *char*, *double*, *float*, *integer*, *long*, *short*, *boolean*)
- Object type in the form `"L<classname>;"` where `<classname>` is an internal description of the fully qualified class or interface name
- Array type in the form `"[<array component>"` where `<array component>` can be any basic, object or array type

The internal descriptors of methods have the following syntax:

```
"(<parameter type>*)<return type>"
```

where `<parameter type>*` denotes zero or more internal field descriptors and `<return type>` denotes either an internal field descriptor or *V* letter if the method returns *void*. For example,

```
double[][] FooMethod(String s, long l) {}
```

method is described as

```
"(Ljava/lang/String;J)[[D".
```

2.3 The BCEL project

The BCEL API is a toolkit for static analysis or dynamic creation and transformation of Java class files. It enables a programmer to implement bytecode manipulation code at a high level of abstraction without dealing with the internal details of the class file structure.

BCEL consists of three parts. The first part is a package containing classes to describe the "static" constraints of class file, to reflect the class file format without the possibility to modify it. Another package contains classes for dynamic generating or modifying of classes or methods, which

may be used to add analysis code or to strip useless code from a class file. The third part is a package containing various examples and utilities, like a tool to convert a Java class file to HTML format.

The basic component of the static part of BCEL is a *JavaClass* class that represents a class file. In this class you can find components matching all structures defined in the class file specification [3]. In the static part we can also find the *ConstantPool* class that is of major importance since it contains all the symbolic values used in bytecode. The bytecode then contains only integer values that refer to the constant pool items.

Among others, there is a *Repository* class that contains a collection of Java classes and their mutual dependencies. Using the *Repository* we can check, for example, if some class A is one of the superclasses of some other class B (in other words, if B is a subclass of A).

To create or modify a class file, we use a set of generator classes. Among these are *ClassGen* aimed at changes of the top level information in the class file. Another is *ConstantPoolGen* that handles changes to the constant pool. To operate on a field of the class, there is the *FieldGen* class and for a method and all its properties there is the *MethodGen* class.

Fields, return values and parameters have some type that is described by a signature. To put a certain level of abstraction into signatures, the *Type* class is introduced by BCEL. This generic *Type* has subclasses *BasicType*, *ObjectType*, and *ArrayType* that divides types into groups by their logical meaning.

An important issue is how to process instructions. In BCEL, each instruction is represented by a particular class based on the *Instruction* class. *Instruction* classes are grouped by their meaning into branch instructions, array instructions, arithmetic instructions, stack instructions, constant pool instructions, return instructions and local variable instructions. To collect instructions together into blocks of code, the *InstructionList* class and *InstructionHandle* are used. The *InstructionList* is a simple list of *InstructionHandles*. The *InstructionHandle* is an object that encapsulates the *Instruction* object to avoid direct referencing to the *Instruction*. Using this handle approach instead of direct references makes it easier to add, insert or delete a particular instruction. For example, if we create a certain *Instruction* object and we add it into the *InstructionList* twice, we can still distinguish between the first and the second *Instruction*, because each insertion will cause the creation of different handle. Without a handle, we could not distinguish between these two instructions, because both would be represented by the same reference. Moreover, we could not use the reference to address the target of the branch instructions, e.g., *GOTO*, because it would not be a unique identification of a position. An example is shown in Figure 2.1.

Since some instructions exist in several versions for different operand sizes, BCEL replaces them with a universal substitute. While rendering

```
// With instruction handles
InstructionList il = new InstructionList();
Instruction i = new AALOAD();
InstructionHandle ih;

ih = il.append(i);
il.append(i);
Instruction ig = new GOTO(ih);
// here we know exactly what is the target of the GOTO instruction
il.append(ig);


// The same code, but instructions are referenced directly
InstructionList il = new InstructionList();
Instruction i = new AALOAD();

il.append(i);
il.append(i);
Instruction ig = new GOTO(i);
// here we cannot say whether the program should jump to the first
// or to the second instruction
il.append(ig);
```

Figure 2.1: Importance of instruction handles encapsulating instructions

the resulting code, BCEL then dynamically decides what version will be used. An example is the *GOTO* instruction using a 16 bit operand and the *GOTO_W* instruction with a 32 bits operand.

BCEL offers a general and abstract API for bytecode manipulation in a class file. However, as we describe in the following section, a less abstract approach can lead to better performance.

2.4 The ASM project

Like BCEL, ASM is a toolkit designed to dynamically manipulate and transform Java class files. It is aimed at performance. Compared to BCEL, it is more than ten times smaller (21kB compared to 350kB) and processing overhead is about 60% of the class loading time compared to 700% in BCEL [1].

Compared to other similar projects, like BCEL or SERP, ASM brings a new technique into class files manipulation. While BCEL parses a class file and represents various structures into a tree class structure, ASM uses the visitor approach which simply traverses all items from a class file and makes a given visitor visit them. Visits proceed as the information is fetched from a file and it is not kept in memory. BCEL also supports the visitor approach, but it is implemented on top of the complete memory representation of the

class, so there is no performance benefit.

ASM consists of several packages. The basic package is the minimum needed for the parsing of existing and the creation of a new class file. Additional packages offer, for instance, functionality similar to BCEL that represents bytecode in memory as a tree structure or functionality for analyzing bytecode, i.e., type checking and/or data flow. Using the *XML* package we can also convert a class file into an XML structure. Using an XSLT filter, we can perform certain bytecode transformation and using the same package we can convert it back into the class file fashion. Packages are well separated, so the core, which is only 21kBytes, can be used without the presence of the other packages.

For static analysis, we use the *ClassReader* class and the *Visitor* interface. The ClassReader parses the requested class file and makes a given visitor visit each of the parsed items. For visiting more complex structures, like a field, method or annotation, the *ClassReaded* asks the *Visitor* for *FieldVisitor*, *MethodVisitor* or *AnnotationVisitor* objects.

Due to simplicity, the basic package does not contain a repository like the one introduced in BCEL. If the programmer needs to keep track of class dependencies, he must do so himself. On the other hand, it can lead to a performance benefit, since unused information can be ignored and does not need to be stored.

As we need to create or modify a class dynamically, the *ClassWriter* class enters the stage. It implements the *Visitor* interface, so for the simplest parse/rebuild use of this framework, we can just pass an *ClassWriter* instance directly to the *ClassReader*. If we want to modify a certain class, usually we use another custom visitor that is forwarding visits from the *ClassReader* to the *ClassWriter* and, if necessary, performs changes to the visited information. There are also *FieldWriter*, *MethodWriter* and *AnnotationWriter* classes matching the corresponding visitor. To handle the constant pool items we use the *Item* class.

Field descriptors, as well as method descriptors or class/interface name descriptors, are always treated in its original fashion as a string without any conversion. All conversion or encapsulation into another object would cause unwanted overhead.

Probably the most important reduction of overhead lies in avoiding any kind of checking or verification and mainly representing instructions using integer constants instead of individual objects like in BCEL. A block of code is stored in *MethodWriter* in an annotated vector of bytes that almost corresponds to the resulting bytecode.

Besides the visitor approach and avoiding the usage of additional objects and unnecessary checks, there is also a strict set of performance and size optimizations that helps ASM to achieve its goal. Performance is mainly addressed by avoiding string manipulations and array copying. Another performance optimization is fetching frequently used fields to local variables

to avoid field accesses. Also, efficient *integer* driven data structures are used. For size reduction, ASM keeps the number of the classes in the framework as low as possible. It uses short field names, so neither storing nor accessing of the field takes too much space. ASM also reduces the number of methods by inlining those that are called only once. Finally, it uses string constants directly in the code instead of declaring them as static fields.

The results presented in [1] show that the use of ASM leads to significant performance benefits compared to BCEL. On the other hand, the absence of the storing of visited parts can be insufficient to solve more complex tasks. But even if we build the structure manually, we can profit from the skipping of unwanted information and thereby save time and memory.

Chapter 3

Serialization

3.1 Serialization in Java

To build parallel/distributed applications, it is crucial to have the ability to transfer objects. To do this, the information carried by the object must be converted into a form, e.g., an array of bytes, which is easily storable and transferable. A serialized object must contain sufficient information to reconstruct it again. Therefore, the type identification must be included as well as all other serializable objects referenced from fields in the current object.

Sun specifies [5] how to achieve serialization by a simple yet extensible mechanism. The basic idea of serialization consists of marking classes as being serializable. This is to be done by implementing the *java.io.Serializable* or *java.io.Externalizable* interfaces.

3.1.1 Serializable

The *java.io.Serializable* interface is used if a programmer wants automatic conversion of the contained information as well as the information from serializable superclasses. Since complex datastructures may contain cycles, cycle detection is also performed automatically. The *java.io.Serializable* interface is an empty public interface:

```
package java.io;

public interface Serializable {};
```

There is one important rule and a few additional options. A class implementing this interface (directly or indirectly) must have access to the non-argument constructor of the first superclass that is not serializable to be able to initialize its content while being deserialized. The class can also

identify which fields are stored and which are not. This can be done explicitly by a list stored in the *serialPersistentField* field or by denoting a field with the *transient* keyword saying that the field is omitted from conversion. Besides that, the following methods can optionally be implemented:

- A *writeObject* method to control what information is saved (the superclass does not need to be considered) or to append additional information to the stream
- A *readObject* method either to read the information written by the corresponding *writeObject* method or to update the state of the object after it has been restored (e.g., evaluation of some data dependent on deserialized data as well as on THE current computer)
- A *writeReplace* method to allow a class to nominate a replacement object to be written to the stream[1][2]
- A *readResolve* method to allow a class to designate a replacement object for the object just read from the stream[2][3]

3.1.2 Externalizable

The *java.io.Externalizable* interface is used if the programmer wishes to have complete control over how conversion is handled. The conversion of the class fields and fields of its superclasses is entirely controlled by the programmer, including whether or not a field will be stored and which output format will be used. The *java.io.Externalizable* interface is declared as follows:

```
package java.io;

public interface Externalizable extends Serializable {
  public void writeExternal(ObjectOutput out)
    throws IOException;

  public void readExternal(ObjectInput in)
    throws IOException, java.lang.ClassNotFoundException;
}
```

If a class implementing this interface is being serialized, only the type identification is written, after which the *writeExternal* method is invoked. The type identification is necessary for deserialization to recognize that the

[1] See Section 2.5 of the Sun Serialization Specification [5], "The *writeReplace* Method" for additional information.

[2] Neither of the *writeReplace* and *readResolve* method calling mechanisms is implemented in the current version of Ibis

[3] See Section 3.7 of the Sun Serialization Specification [5], "The *readResolve* Method" for additional information.

readExternal method of the particular class must be called. The *readExternal* and *writeExternal* methods are entirely responsible for the format of the serialized output as well as for the processing of the fields of the superclass. The only requirement for the class is to have a public no-argument constructor, because the object must be created before the *readExternal* method can be invoked. Optionally, the class can also implement the following methods:

- A *writeReplace* method to allow a class to nominate a replacement object to be written to the resulting byte array[12]
- A *readResolve* method to allow a class to designate a replacement object for the object just read from the resulting byte array[23]

3.1.3 Conversion mechanism

We will now introduce the classes that implement serialization. The basic set of the methods used for the serialization is defined by the *java.io.ObjectOutput* and *java.io.ObjectInput* interfaces. The *java.io.ObjectOutputStream* and *java.io.ObjectInputStream* classes implement these interfaces. The *writeObject* and *readObject* methods of these classes implement serialization using runtime type inspection. The values of fields are written into the output stream in accordance with the Sun specification[4]. A special form is chosen just for *String* and *Class* values. The runtime type inspection is one of the major performance drawbacks of the generic Java serialization provided by Sun and led to the compile-time rewriting approach presented by the *Ibis project* [6].

3.2 Serialization in the Ibis project

As explained in the previous section, serialization is a mechanism of converting objects (graphs of objects) in Java into a form which is easy to store and transfer. Serialization following Sun's specification [5] introduced in Section 3.1 is called *Sun serialization* in Ibis. This serialization is implemented using *java.io.ObjectOutputStream* and *java.io.ObjectInputStream*. Ibis offers its own serialization mechanism which is fully compatible with *Sun serialization* (with regards to its interfaces), but is more efficient.

Since object serialization is not always needed, Ibis introduces alternatives for *java.io.DataOutputStream* and *java.io.DataInputStream* that are aimed at serialization of basic types and its arrays. Simple versions of the output/input streams that can only process bytes and arrays of bytes are also present in Ibis.

[4]See Sun Serialization Specification [5], Chapter 6, "Object Serialization Stream Protocol" for more details about its form.

```
class Foo implements java.io.Serializable {
  int i1, i2;
  double d;
  int[] a;
  Object o;
  String s;
  Foo f;
}
```

Figure 3.1: Serializable class Foo

Although Ibis serialization is designed to take maximum advantage of the communication interface, it is independent of the communication layer, which can contain pieces of native code. The communication layer, however, must have some knowledge of how the serialized data is stored in order to minimize copying. This knowledge does not affect the serialization itself. Therefore, the serialization is implemented in pure Java and is highly portable.

In a work done prior to the *Ibis project* [4], most of the important performance issues of *Sun serialization* were addressed. The most significant performance overhead resides in the runtime type inspection, object creation, multiple copying of data and using an inefficient protocol.

To prevent the overhead of runtime type inspection, Ibis uses compile-time reflection of a class and extends the class with code for writing and reading of its fields. Figure 3.1 shows an example of a serializable class *Foo*. Figure 3.2 shows the rewritten version of the class.[5] The class is tagged with the interface *ibis.io.Serializable* to show that it supports the Ibis serialization mechanism. It is also extended with the *ibisWrite* method for the writing mechanism and the *Foo(ibis.io.IbisInput- Stream in)* constructor for reading. The cycle check for the writing part is performed in the *ibis.io.IbisOutputStream*, because the current object reference already exists before the *ibisWrite* method call. Since the reading part is implemented by the constructor, the code of the constructor is the first place after allocation where a new object reference exists. Because the object can refer to itself through one of its fields, the new reference is immediately added to the cycle check mechanism.

Ibis also tries to avoid expensive creation of objects during deserialization. Due to inheritance, we cannot always know the actual type of the object that will be referenced from some field in advance (see *o* field in Figure 3.1). Therefore, we write a type descriptor into the stream. During deserialization *Sun serialization* uses a private native method of the standard class library to create an object of the given type without invoking a

[5]The code in the figures is presented in its Java fashion for better readability, although the rewriting is performed on compiled bytecode.

```
public final class FooGenerator extends Generator {
  public final Object doNew(ibis.io.IbisInputStream in)
    throws ibis.ipl.IbisIOException, ClassNotFoundException {
      return new Foo(in);
    }
  }

class Foo implements java.io.Serializable, ibis.io.Serializable {
  int i1, i2;
  double d;
  int[] a;
  String s;
  Object o;
  Foo f;

  public void ibisWrite(ibis.io.IbisOutputStream out)
    throws ibis.ipl.IbisIOException {
      out.writeInt(i1);
      out.writeInt(i2);
      out.writeDouble(d);
      out.writeObject(a);
      out.writeUTF(s);
      out.writeObject(o);
      out.writeObject(f);
    }

  public Foo(ibis.io.IbisInputStream in)
    throws ibis.ipl.IbisIOException, ClassNotFoundException {
      in.addObjectToCycleCheck(this);
      i1 = in.readInt();
      i2 = in.readInt();
      d = in.readDouble();
      a = (int[])in.readObject();
      s = in.readUTF();
      o = (Object)in.readObject();
      f = (Foo)in.readObject();
    }
  }
```

Figure 3.2: Code of the rewritten class Foo

```
public final void ibisWrite(ibis.io.IbisOutputStream out)
  throws ibis.ipl.IbisIOException {
    // Code to write fields i1 ... o is unchanged.
    boolean nonNull = out.writeKnownObjectHeader(f);
    if(nonNull) f.ibisWrite(out);
  }

public Foo(ibis.io.IbisInputStream in)
  throws ibis.ipl.IbisIOException, ClassNotFoundException {
    // Code to read fields i1 ... o is unchanged.
    int i = in.readKnownTypeHeader();
    if(i == NORMAL_OBJECT) f = new Foo(in);
    else if(i == CYCLE) f = (Foo)in.getFromCycleCheck(i);
    else f = null;
  }
```

Figure 3.3: Optimization for Foo class marked final

constructor (since this can have unwanted side-effects). It is more efficient to create new object using ibis constructor, which directly reads the content without any side-effects. One solution is to use the *Class.newInstance* method for the creation. As [6] explains, the *Class.newInstance* call is up to six times more expensive than the *new* instruction call. Therefore, it is more efficient to call the *new* instruction followed by the ibis constructor call. However, the name of the class to create using *new* must be known at compile-time. Therefore, the class name read from the input stream cannot be used. Thus, it is more efficient to generate a piece of code that calls the *new* instruction for the desired class. To implement this, Ibis introduces a generator class. This class is a product of the rewriting process. It contains a method *doNew* (see *FooGenerator* in Figure 3.2) that simply calls the *new* instruction for the desired class followed by the ibis constructor call. The serialization mechanism calls the expensive *Class.newInstance* method only once to obtain the generator and then the cheap *doNew* method is called for every appearance of the class during deserialization.

For final types, Ibis serialization performs another optimization. If the type of a field is marked as final, then the type of the referenced object is always known at compile-time. Therefore, the type information does not have to be written into the stream. In addition, the reading part can directly call *new* for the field value. Of course, cycles must also be checked. For the writing part this is done by the *writeKnownObjectHeader* method and for the reading part by the *readKnownObjectHeader* method, as shown in Figure 3.3.

An important benefit of Ibis is the zero-copy approach that reduces memory copying overhead. This overhead can be relatively expensive for fast networks. By extending the buffering mechanism with a new layer that treats

arrays separately, the number of copies per array is reduced to one.

In [6] measurements of *Sun serialization* and *Ibis serialization* are presented and compared. For serialization of an object containing only an array of bytes (without zero-copy optimization) we see only little performance benefit. As soon as we serialize complex structures of objects, the benefit peaks to 500%. From this we can conclude that the rewriting approach can significantly improve serialization performance.

Chapter 4

Proposed solution

This chapter describes the runtime class rewriting mechanism proposed earlier. In section 4.1, the current, compile-time rewriting approach is summarized. It is also explained when and how the rewriting proceeds and how the current rewriting mechanism works.

Section 4.2 explains how the loading of classes is customized by the new class loader. Trade-offs given by loading constraints, as briefly explained in Section 2.1, are also described.

Section 4.3 explains the actual rewriting of the class file structure. The set of classes of the solution is introduced, then implementation using BCEL is explained, followed by an explanation of the ASM solution. The SysGenerator program is described at the end of the section. Finally, changes to the current Ibis are listed.

In the following text we will use the *java serializable*, *externalizable* and *ibis serializable* terms for classes. The *java serializable* class is a class implementing the *java.io.Serializable* interface (either directly or indirectly), but not *ibis.io.Serializable*. The *externalizable* class is a class implementing the *java.io.Externalizable* interface, but not *ibis.io.Serializable*. Finally, *ibis serializable* is a class directly implementing the *ibis.io.Serializable* interface. A class that indirectly implements *ibis.io.Serializable* should not exist. *Serializable* is the term we use for a class which is either *java serializable*, *externalizable* or *ibis serializable*. A constructor created by the rewriting mechanism with its method descriptor

```
"(Libis/io/IbisSerializationInputStream;)V"
```

we call the *ibis constructor*.

4.1 Current Compile-time rewriting approach

The current compile-time rewriter (see Figure 4.1) consists the of *ibis.frontend.io.IOGenerator* program that is responsible for the rewriting of existing

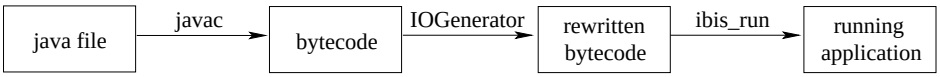

Figure 4.1: Compile-time rewriting approach

class files. The behavior of the IOGenerator is controlled by command-line parameters. To modify classes, the IOGenerator uses the BCEL framework.

The IOGenerator processes classes in three phases. First, based on parameters, it prepares a list of the classes to be rewritten. Then it takes the list and extends all classes with the *ibis.io.Serializable* interface and with empty new write/read methods. Finally, it adds bytecode to these methods. The creation of the code must be separated from the actual creation of methods, because code creation depends on the presence of the Ibis constructor in its superclass or field classes.

The classes to rewrite can be listed based on the directory structure, a list of files or a list of classes. All of the items on the list are translated into class names and then the list is processed further using following algorithm:

- for each class in the original list:
  - load it
  - scan to see if its parent is java serializable
  - if so, add the parent to the rewrite list
  - scan to see if the class itself is serializable
  - if so, add it to the rewrite list
- for each of the classes in the rewrite list:
  - check if it contains references to the final serializable objects
  - if so, add these objects to the rewrite list
  - check if it is already ibis serializable
  - if not, add the *ibis.io.serializable* interface and add the necessary methods (without bytecode)
  - check if the class is present in the original list to prevent a non-requested class file from being rewritten
  - if so, check if the class is already in the target list
  - if not, add it to the target list
- for each of the classes in the target list:
  - generate the code for the empty methods
  - save the class file

Note that before any list is processed, it is always ordered (superclass first), because a class with an ibis serializable superclass is treated in a different way than a class with a sun serializable superclass.

Extension of a class with the new interface *ibis.io.Serializable* proceeds by using the *addInterface* method of the particular *ClassGenerator* class. Methods are created using the particular *MethodGenerator* classes.

As we explained, extension of the class with code must proceed after the actual creation of the methods and the addition of the new interface. The creation of bytecode for a class can be influenced by the presence of the ibis constructor in classes of its final fields. If the code would be created immediately, the resulting code would work, but it would not fully benefit from the Ibis extensions. Details of the generated bytecode are described later in Section 4.3.

Now we will describe how rewriting is performed at compile-time. One approach is, that the programmer runs the IOGenerally manually for a class. However, he should be aware that all related classes should also be rewritten to achieve maximal benefit of the Ibis approach. Also, the situation when a certain class is ibis serializable, but its subclass does not implement *ibis.io.Serializable* directly, should be prevented. A much easier way of rewriting is to develop an application using the Ibis build structure and to use the ant target *iogen* after the compiling target *compile*. This is an inefficient approach for the programmer, but it was already described as one of the reasons for the runtime solution.

BCEL uses a substantial amount of memory for its class representation. Since the IOGenerator is rewriting many classes at once, it causes a very high memory demand. If we rewrite too many classes at once, the IOGenerator throws an insufficient memory error. Therefore, it is necessary to proceed with rewriting piecewise. For this purpose, the *-force* parameter is introduced. This parameter specifies that the sun serializable or externalizable class is treated as ibis serializable. For example, a certain sun serializable class will be rewritten properly, although its sun serializable superclass will be rewritten in a later turn.

4.2 Load-time approach

The mechanism of the user class loader is implemented by subclassing the *java.lang.ClassLoader* class in Java. By overriding the

```
public class loadClass(String);
```

method, we can replace any other loader with our own.

4.2.1 Calling mechanism

It is obvious that we need a certain set of classes to be present for achieving the basic functionality of the JVM environment. There must also be a certain set of classes which are used by the system class loader, and thus by our class loader. Logically, we cannot use a class loader to load itself and the classes that are used in its internal mechanisms.

How the JVM should start a program is not exactly specified, but current JVMs, such as the JVM from Sun JDK 1.5, works as follows. After the JVM starts, the bootstrap loader loads the system classes that are closely related to the JVM core. These classes are sufficient for running a temporary class loader that is responsible for loading the system class loader. If there is no custom system class loader specified, this temporary class loader is used as a system class loader. Based on the triggering mechanism introduced in Section 2.1, all of the system class loader's internally used classes will also be loaded by the temporary class loader. The system class loader is then asked to load the application. Finally, the *main* method of the program is invoked. The other loading is then based on a referenced triggering mechanism.

After defining our class loader, we need to set it up into the JVM mechanism. We also need to ensure that as many classes as possible have passed through this class loader and therefore have been rewritten, if required. One way to do this is to configure the JVM to use our class loader to load the application. To achieve this, we use the system property *java.system.class.loader* that contains the name of the class loader used as a system class loader. The system class loader then loads the application.

The system class loader is returned at runtime by the static method *ClassLoader.getSystemClassLoader* or implicitly also by the *Thread.currentThread().getContextClassLoader()* method. All implicit ways of class loading are thus intercepted by setting up the system class loader, but there is also an explicit way of class loading using the static method *Class.forName*. This invocation asks a class loader of the class, that invoked the *Class.forName* method, to perform the actual loading. When the calling class has been loaded by our class loader everything works, but when it has not we must replace the *Class.forName* invocation by the *loadClass* method of our current system class loader.

As we already mentioned, we are not able to process runtime rewriting on classes from the *java.* package, because these classes are considered as system classes. The *defineClass* method will not accept their creation, since this should be handled only by the bootstrap class loader. This restriction means that either we completely skip the rewriting mechanism for the *java.* package tree, or we rewrite it at compile-time. To achieve the maximal performance benefit, we choose the compile-time rewriting approach. This partial compile-time rewriting is performed by the *SysGenerator* program described later in this chapter.

4.2.2 Class loading constraints

We have decided to use partial compile-time rewriting. Therefore, particular sun serializable classes from the *java.* package will contain reference to Ibis serialization classes. Thus, these classes will be loaded by bootstrap loader. Therefore, they must be also rewritten at compile time.

Considering the triggering mechanism introduced in Section 2.1, if system classes are extended with the Ibis approach, the bootstrap class loader L_{boot} will be also used to load some class C used for the Ibis serialization mechanism. The C class is denoted by N. Later, some non-system class, which is sun serializable, will be loaded by our class loader L_{our} and therefore will be rewritten into Ibis fashion. Because the class will use the same Ibis mechanism, it will reference a class C' denoted also by N. The triggering mechanism will find out that there is no pair $N^{L_{our}}$ registered yet, so our class loader will be asked for a class denoted by N, as well. If our class loader proceeds with the loading without delegating it, then there will be C and C'. Both C and C' are denoted by N, but such that $C \neq C'$. If there will first occur an operation when $N^{L_{our}}$ will be expected to be the same as $N^{L_{boot}}$, the operation will fail, due to loading constraints. An example of such a situation could be a user program that passes a certain *ibis.io.IbisSerializationOutputStream* object to a newly added method of some system class. It is expected that the parameter and the given value are of the same type, since it is denoted by the same name. Because the object is linked from the user class and the parameter type is referenced from the system class, there are actually two different classes and the invocation will therefore fail. We can conclude that if we want to load the Ibis serializability mechanism class, we must delegate the loading from our class loader to the bootstrap loader. Following these rules, every class will be loaded only once.

Another issue is the use of the *Class.forName* method call in the Ibis implementation. As we explained, potential class loading in this method is performed by a caller's class loader. This causes a problem for the classes used for the Ibis serialization mechanism, because we load them using the bootstrap loader. However, the Ibis classes source code can be changed, so each occurrence of *Class.forName* can be replaced with an invocation of the *loadClass* method of our class loader.

As a result, we must find a set of classes that are actually referenced from code added by rewriting to classes. Then, we must make transitive closure of this set using all field types, all methods' return types and all methods' parameter types. Finally, all *Class.forName* calls must be replaced with *Thread.currentThread().getContextClassLoader().loadClass* calls in each class of the set. Referenced system classes can be skipped from the closure, because they do not reference any class from Ibis. We try to keep the set as small as possible. If we would add another class from Ibis that is not

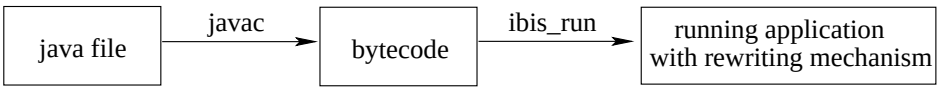

Figure 4.2: Run-time rewriting approach

in the transitive closure, it would cause another transitive growing of the set. Finally, the group could contain the whole Ibis framework and therefore runtime class loading would become useless, at least for the Ibis framework classes.

A dynamic version of the transitive closure evaluation was also considered. Dynamic scanning of detailed class dependencies would cause a significant performance drawback, so it is more efficient to have a fixed list of the classes whose loading must be delegated. However, the fixed list of delegation means that all future changes of dependencies in the Ibis framework classes must be revised, unless it does not change the delegated set.

As we explained in this section, loading all classes from the described set must be delegated. To achieve that, the *delegatedClass* method is introduced in the *IbisClassLoader* class. This method checks whether a given class name is delegated or not. Moreover, the BCEL and ASM solutions can extend the set by classes of the BCEL framework or the ASM framework that also cannot be loaded by these class loaders.

We have mentioned that the *java.* package classes are too sensitive to be loaded using the *defineClass* method. In fact, several more classes cannot be processed by *defineClass*, so the delegated list is extended with certain *sun.* classes. It is feasible that in JVM implementations other than the ones tested[1], the list needs to be extended further.

4.3 Implementation

This section describes implementation of run-time rewriting mechanism shown in Figure 4.2.

4.3.1 Structures

We must implement a new subclass of the *java.lang.ClassLoader* class. As we mentioned in the previous section, the class loading based on the BCEL framework as well as on the ASM framework will be implemented, so we will be able to evaluate the benefit of the ASM solution. If we would cover both frameworks with one class loader, performance would be decreased by the loading of the classes of the framework that is not used at that time. Therefore, it is more efficient to implement two different class loaders.

[1]Sun JDK 1.4, Sun JDK 1.5, IBM JDK 1.4

However, there is a certain part of functionality of these two class loaders, which can be generalized to one common superclass.

We therefore introduce following classes:

- *java.lang.ClassLoader*
  - subclass *IbisClassLoader*
    - subclass *BCELClassLoader*
    - subclass *ASMClassLoader*

The given set of classes works as follows. The *IbisClassLoader* overrides the *loadClass* method that is responsible for returning the loaded class to the caller. The class loader defines an abstract method, *loadClassData*, that is responsible for fetching the class from the storage space and also for the actual rewriting if it is necessary, i.e., if it is a sun serializable class to be rewritten. Both class loaders, *BCELClassLoader* and *ASMClassLoader*, override this method and proceed with rewriting in their own manner. After loading and possibly rewriting, the *IbisClassLoader* class uses the *defineClass* method to create a class in the currently running JVM.

As we mentioned in Section 3.2, rewriting one sun serializable class can result in two classes, the class and its new generator class. Therefore, we must provide the *loadClassData* method with a mechanism that can output information for multiple classes. To contain multiple classes, we introduce the *ClassItem* wrapping class.

In addition, the *CLProps* class and the *IbisLoaderFlags* interface are used to support custom settings for debugging output and verifying the rewritten classes.

Last but not least in the group of classes is *SysGenerator*. This class is responsible for partial compile-time rewriting. As will be explained later in this section, due to loading constraints, some of the classes must always be rewritten at compile-time. For this purpose, SysGenerator is introduced.

Generally, we have introduced the following classes, which are available in the package *ibis.backend.loader*.

- *IbisClassLoader*
  - *BCELClassLoader*
  - *ASMClassLoader*
- *ClassItem*
- *CLProps*
- *IbisLoaderFlags*
- *SysGenerator*

4.3.2 Using BCEL

The BCEL class loader is based on the previous compile-time rewriter, *IO-Generator*. The basic part of this solution is the nested *CodeGenerator* class. This nested class was chosen, because originally non-lazy loading was expected. Non-lazy loading means that the rewriting of the class invokes the loading and rewriting of its superclasses and any classes used for its fields. This means that many rewriting processes could be started at the same time. However, the lazy loading approach appeared to be useless, because many of these classes do not have to be used at all. Therefore, there is only one class rewriting at a time. Classes are loaded, as the JVM needs them, thus the rewriting mechanism must pretend that unloaded classes are already rewritten, although they are not yet.

The *generateCode* method of the nested class *CodeGenerator* first adds the new interface, *ibis.io.Serializable*, to mark a class as ibis serializable. Then, the following methods are created:

- *generated_DefaultWriteObject*
- *generated_DefaultReadObject*
- *generated_WriteObject*
- optionally ibis constructor or *$readObjectWrapper$* method

The *generated_DefaultWriteObject* method was not, due to simplicity, described in a Section 3.2. The method is used by the *defaultWriteObject* method that can be invoked from the *writeObject* method of the sun serializable object. Because *defaultWriteObject* writes only fields from the particular level of the class hierarchy, we use the newly generated methods to ensure that only a certain level will be written. Due to inheritance, the level can differ from the level of the target. The reading is done in the same way using the *generated_DefaultReadObject* method. You can find the code created for these methods in Figure 4.3. For better readability, a java version of the code is shown instead of bytecode.

The *generated_writeobject* method is a default method used for the actual writing to the output stream. For a serializable parent, the method also directly uses the parent's writing mechanism. Detailed code of the method is presented in Figures 4.4 and 4.5.

Optionally, an ibis constructor can be created. The ibis constructor is not created, if and only if, the following condition holds. The class is not externalizable, the superclass is serializable and the superclass does not have an ibis constructor. This condition describes the situation if there is not a proper parent constructor to be invoked. There is no ibis constructor and the default constructor cannot be invoked due to possible side effects. This happens if the superclass is only sun serializable, for example, when Java

```
void generated_DefaultWriteObject(
  IbisSerializationOutputStream out, int level) {

  if (level == dpth) {
    // ... write fields ...
  } else if (level < dpth) {
    // This branch is just if the superclass is serializable

    // For ibis serializable superclass
    super.generated_DefaultWriteObject(out, level);

    // For sun serializable superclass
    out.defaultWriteSerializableObject(this, level);
  }
}

void generated_DefaultReadObject(
  IbisSerializationInputStream in, int level) {

  if (level == dpth) {
    // ... read fields ...
  } else if (level < dpth) {
    // This branch is only if the superclass is serializable

    // For ibis serializable superclass
    super.generated_DefaultReadObject(in, level);

    // For sun serializable superclass
    in.defaultReadSerializableObject(this, level);
  }
}
```

Figure 4.3: Template of the code of *generated_DefaultWriteObject* and *generated_DefaultWriteObject* methods

```
void generated_WriteObject(IbisSerializationOutputStream out) {

  // Initialization code for ibis serializable superclass
  super.generated_WriteObject(out);

  // Initialization code for sun serializable superclass
  out.writeSerializableObject(out);

  // Code, if writeObject is present
  out.push_current_object(this, dpth);
  this.writeObject(out.getJavaObjectOutputStream());
  out.pop_current_object();

  // Code, if writeObject is not present
  // ... write fields ...
}
```

Figure 4.4: Template of the code of *generated_WriteObject* method for serializable class

```
void generated_WriteObject(IbisSerializationOutputStream out) {

  out.push_current_object(this, dpth);
  this.writeExternal(out.getJavaObjectOutputStream());
  out.pop_current_object();
}
```

Figure 4.5: Template of the code of *generated_WriteObject* method for externalizable class

```
thisClassName(IbisSerializationInputStream in) {

  // for externalizable
  thisClassName();
  addObjectToCycle(this);

  // for non-serializable parent
  super();
  addObjectToCycle(this);

  // for ibis serializable parent
  super(in);

  // for externalizable
  in.push_current_object(this, dpth);
  this.readExternal(in.getJavaObjectInputStream());
  in.pop_current_object();

  // if readObject is present
  in.push_current_object(this, dpth);
  this.readObject(in.getJavaObjectInputStream());
  in.pop_current_object();

  // for non-externalizable and without readObject
  // ... read fields ...
}
```

Figure 4.6: Template of the code of ibis constructor

core classes are not rewritten. If the ibis constructor is not created and the *readObject* method (used to extend generic information) is present, we also create the *readObjectWrapper* method, which is used from the generator class created for this class. Detail of the code is shown in Figures 4.6 and 4.7.

As mentioned at the beginning of this section, the rewriting mechanism must treat classes as rewritten, although they may be loaded and rewritten later. This is not a problem for serialization, because if a class is sun serializable and its loading is not delegated to the bootstrap loader, then it will be rewritten and we can consider it as ibis serializable. It is more difficult

```
void $readObjectWrapper$(IbisSerializationInputStream in) {

  in.push_current_object(this, dpth);
  this.readObject(in.getJavaObjectInputStream());
  in.pop_current_object();
}
```

Figure 4.7: Template of the code of *readObjectWrapper* method

to determine the presence of the ibis constructor. To check that, we use a recursive method which browses through the superclasses and checks for the presence of an ibis constructor.

The last part of the generated code is a generator class described in Section 3.2. If there is an ibis constructor, the *doNew* method of this generator will simply invoke:

```
new thisClassName(ibisInputStream);
```

If no ibis constructor was created, a special trick is used, which uses a native call to obtain a new initialized object without any constructor call.

Finally, if the verification option is enabled, verification of the rewritten class and its generator class is invoked. The verifier is a part of the BCEL framework. This option, however, will usually report reference errors, because, for example, the superclass must not have been rewritten yet, but this class can already refer to its new methods.

4.3.3 Using ASM

Like the BCEL class loader, the ASM class loader is aimed at rewriting a class into the ibis serializable version, but to do so more efficiently. The rewritten code of the rewritten class is entirely the same as when the BCEL solution is used. Thus, we will only describe how the second solution works. The code was described in the previous section.

An important fact in ASM is that there is no repository of the classes, so the interclass dependencies are not stored. In the ASM class loader, the repository is realized by a hashtable containing the class name with its integer descriptor. The integer value consists of logical (0/1) qualities of the class (for example, whether it has an ibis constructor, whether it is sun serializable, etc.) and also the level of serializability described in Section 3.2. To obtain partial information about a class, the loader must parse a class file and optionally check other classes.

As was introduced in Section 2.4, the ASM is based on the visitor approach. The ASM class loader contains two main visitors. One for very efficient extraction of the basic class properties for the repository and another for the actual rewriting.

One of the stored properties of the class is the presence of an ibis constructor, although it has not necessarily been rewritten. Therefore, parsing a class can invoke the parsing of another class. The presence of the ibis constructor can be evaluated by the same recursive algorithm as in the BCEL class loader.

The first of the two visitors, *CodeAnalyser*, is designed for this purpose. It picks up the required information from the visited entities only. For other entities, like a field or attribute, it only implements empty visiting methods.

The second visitor, *CodeRewriter*, stores all needed visited information

and then forwards all the data to the *ClassWriter* visitor which rebuilds the original class and also adds the new methods. The visitor, while forwarding the information, also builds the *serialVersionUID* value containing the version information of the class. In the case the *serialVersionUID* field does not exist yet, the ASM class loader creates the field with the new value.

To optimize ASM rewriting performance, values of the maximum of local variables and the maximum of stack items, that are required for each method, are not evaluated automatically. The ASM class loader instead updates temporary maximum values in every branch of the code. After insertion of all bytecode, the maximum values are used for the method header.

The implementation of the efficient repository in combination with the efficient ASM visitors approach is expected to result in a very efficient rewriting mechanism. In the next chapter, performance comparisons between ASM and BCEL can be found.

4.3.4 SysGenerator

SysGenerator is a program responsible for compile-time rewriting of the System classes. Based on command-line parameters, it uses the rewriting mechanism from the BCEL class loader solution or the ASM class loader solution. It can rewrite Java core classes as well as certain non-system classes. Classes are only rewritten if they are marked as delegated.

SysGenerator is called from the *ibis-run* script which is used to start a program that uses the Ibis framework. Rewritten, non-system classes are kept in the same file, the original file is rewritten and the Java core classes are put together in one class library. The location of the library with rewritten classes can be static, which means that it is in a certain specified directory in the Ibis build structure, or the location can be in the temporary directory. A shared file system is not always available for computers running in a grid or cluster, or this space is too expensive to access. Therefore, as we run the *ibis-run* script in a distributed environment, it sets up SysGenerator to use the temporary or static location.

For the static version, we perform an optimization. The timestamp of the last change of the target library is set to the timestamp value of the last change of the Java core classes library. At the next run, if the value is not changed, we do not have to rewrite the core classes again.

As we described, besides the system classes, there is a set of Ibis framework classes that are loaded by the bootstrap loader and therefore must be rewritten at compile-time. This list is stable and also serializability of these classes is stable. Thus, if we check which classes of the list are serializable, we can set the parameters of SysGenerator to rewrite only these serializable classes. For the others, we will save the time needed to check if the file is serializable. Because only 3 classes out of about 40 classes of the current Ibis serialization implementation are serializable, we can certainly save time for

many IO operations. However, another fixed list creates a requirement for the revision of the future changes to Ibis. If another serializable, delegated class appears, the class will have to be added to the list to be rewritten by SysGenerator.

4.4 Changes to the current Ibis approaches

The proposed solution was applied to the current version of the Ibis framework, version 1.3. The changes that have been applied are:

- new package *ibis.backend.loader* containing class loading and rewriting mechanism for both (ASM, BCEL) frameworks was added
- ASM library and source files were added
- starting ibis scripts were adjusted to support a custom class loader
- support for new system options controlling verification and debugging output were added
- benchmarking programs were added
- configuration files for building the application were adjusted to cover the ASM source files and skip previous compile-time rewriting
- *Class.forName* calls were replaced in 5 classes of the *ibis.io* package - *AlternativeTypeInfo*, *Conversion*, *IbisSerializationInputStream*, *IbisSerializationOutputStream*, *SerializationBase*, and one class of the *ibis.util* package - *Timer*

The extended Ibis was tested on the following systems/JVM implementations:

- `DAS2 / SunJava1.5`
- `DAS2 / SunJava1.4`
- `DAS2 / IBMJava1.4.2`

Chapter 5

Measurements

In this chapter we will compare the performance of the ASM and BCEL load time class rewriting. The overall start-up time of both loaders is compared to that of the current compile-time solution. Also, a separate components (IO, processing) of the time required for rewriting is presented and it is described how IO operations influence the overall time.

The testbed used for the measurements was the DAS-2 system. DAS-2 is a wide-area distributed system consisting of five clusters at five Dutch universities. Every node of each cluster contains two 1GHz Pentium-III CPUs, at least 1GB of RAM, a local PATA disk of at least 20GB, a fast ethernet card and a Myrinet network interface card.

The installed operating system is RedHat Linux 7.2 with kernel version 2.4.18. There are several JVM installations available. For our testing purposes, we have used the following versions: Sun JDK 1.4, Sun JDK 1.5 and IBM JDK 1.42. For start-up timing, we have used only Sun JDK 1.5, since the other measurements proves similar results.

5.1 Stress testing

Figure 5.1 shows the results of the runtime rewriting benchmark. This benchmark measures the loading times required for an increasing amount of classes. All of these classes are based on the same template. The template class is in Figure 5.2. Every one of them is sun serializable and therefore a candidate for rewriting. Every one of them also contains fields of all Java basic types and fields of the array types of all Java basic types. There is one transient field present and the class also contains a default constructor that initiates all field values. Measurements from each JVM are shown separately.

Each plot shows three series. The first of them is for the basic loading time without any rewriting mechanism and using the generic system class loader. The second of the series is for the ASM class loader. And last of the series shows the times for the BCEL class loader.

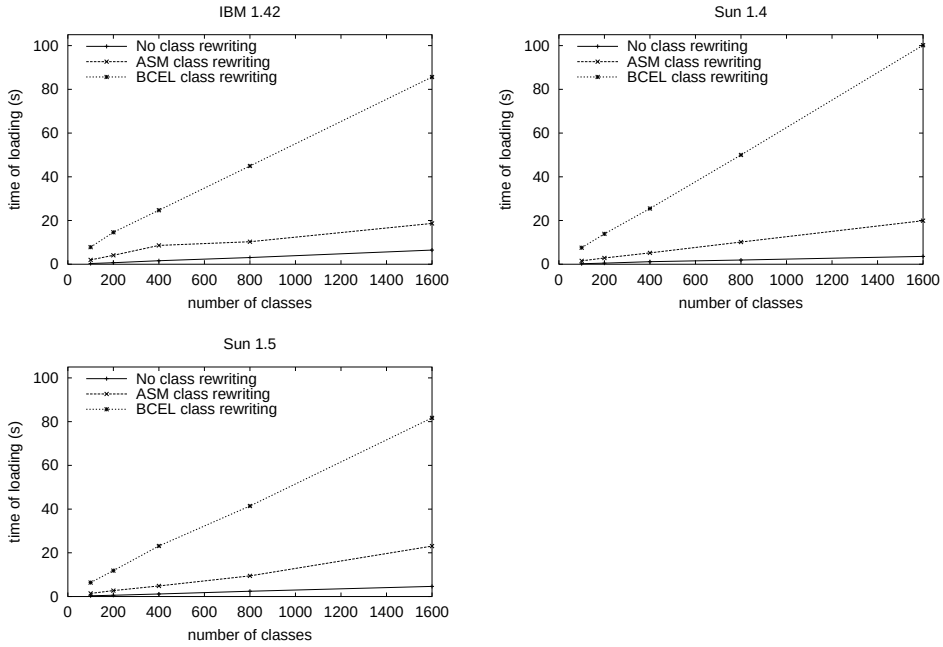

Figure 5.1: Class loading times

The generic class loading consists of the IO operation to read the classes and the internal JVM mechanisms used to create the internal structures for the class. The internal overhead is a part of each of the series.

All three experiments show the expected performance results. The BCEL solution is very inefficient compared to the ASM solution. It takes almost 5 times as long as the ASM time to rewrite all 1600 classes.

The performance of all tested JVMs is similar in the sense that the BCEL solution is significantly slower than the ASM solution. The performance of the BCEL implementation on IBM JDK 1.42 and Sun JDK 1.5 is almost the same. Sun JDK 1.4 is about 20% slower. For the ASM solution, all three JVMs show only slight deviations.

An interesting fact is the perceptible break in a linearity on the IBM JDK 1.42 after around 10 seconds of processing. This is most probably caused by a later start of the JIT[1] compiler in this JDK. Since we did not find the way of setting up the start of the JIT manually, this explanation is merely speculation.

[1] JIT: Just-In-Time compiler transforms the Java byte code to native code which is more efficient to run

```
import java.io.Serializable;

public final class temp implements Serializable {
  byte Byte;
  char Char;
  double Double;
  float Float;
  int Int;
  long Long;
  short Short;
  boolean Boolean;
  byte AByte[];
  char AChar[];
  double ADouble[];
  float AFloat[];
  int AInt[];
  long ALong[];
  short AShort[];
  boolean ABoolean[];
  transient int TransInt;

  public temp() {
    super();

    // Initialization of the values
  }
}
```

Figure 5.2: Template of the class rewritten in measurements

JVM	ASM	BCEL	BCEL (no rep.)
IBM 1.42	5.17	10.36	12.06
SUN 1.4	1.68	7.26	11.15
SUN 1.5	1.96	7.31	12.78

Table 5.1: Internal rewriting time (s) of compile-time rewriting of the core classes

5.2 SysGenerator compile-time rewriting

Tables 5.1, 5.2 and 5.3 show the results of the partial compile-time rewriting benchmark of Java core classes and some of Ibis serialization classes. Since we rewrite only 3 of the Ibis classes at compile time, it hardly influences the results, although they are at a different location than the Java core classes. The benchmark measures the time spent by the compile-time rewriting using the SysGenerator program. It also measures the approximate amount of memory used by the program. The SysGenerator rewrites only those classes that cannot be rewritten in the runtime as we have already explained in the previous chapter.

5.2.1 Internal rewriting time

Table 5.1 shows the internal rewriting time. Since the BCEL runtime class loading uses the same rewriting mechanism as the current compile-time approach, the internal rewriting time required for the current approach is the same. The table also shows the time required for the BCEL solution without bug, which does not use *Repository* (BCEL (no rep.)). Instead, it always parses the super classes and interfaces again and thus it requires more time. However, this adjusted solution uses significantly less memory as described later in this chapter.

5.2.2 IO time

We show separated IO read and write times in Figure 5.2. The write operations are easy to measure, since they are always performed by the SysGenerator. The read operations are different for each implementation. With the ASM class loader, it is easier to measure, because read operations are only performed while initializing the *ClassReader* class for the rewritten class. With the BCEL class loader, the measurement is slightly more complicated. The reading mechanism is hidden inside the class file parsing in the BCEL library. For the actual reading, it uses the *getResourceAsStream* method of the class loader that has loaded the BCEL library. Therefore, if we force a class loader with the customized *getResourceAsStream* method to load the BCEL library, we can measure the time required for read operations. This customized loader cannot be used at runtime, because the BCEL library

is used by our system class loader and therefore it must be loaded by the bootstrap loader. But we can perform such class loading in the SysGenerator, because this program uses the ASM and the BCEL class loaders as independent objects.

The results of the use of the remote location for the rewritten classes (static) are shown separated from the results of the local location for the rewritten classes (temp) to show how NFS overhead influences the resulting times. As can be expected, the difference is perceptible only in the times of write operations. While reading, classes are always local with respect to particular DAS-2 node. But as we write to the output library in the remote location (static), it takes 200% of the local write time (temp).

While the measurements were performed, the output of the BCEL solution showed significantly slower times for the internal rewriting. Therefore, extra effort was undertaken to analyze the internal behavior of the BCEL library. Upon inspection it was found that the BCEL library passes a requested file name of the class file to the *getResourceAsStream* method in an improper way. Consequently, the method unsuccessfully searches through all class file storage, after which the BCEL library searches for the class file using its own mechanism. Generally, this means that all IO reads are performed twice, but only one is measured in the *getResourceAsStream* method. To remedy this defect, we can adjust the file name in this method to its correct form. After this adjustment, we can certainly say that the only reading operations for the BCEL library are performed in the customized class loader. Therefore, we also show times for the BCEL solution with the fixing mechanism for the class file names (BCEL (no bug)) in Tables 5.2 and 5.3.

It is interesting to note that the solution without the repository (BCEL (no rep.)) needs less time for the read operations, although it is reading all of the classes again. In our detailed analysis, we have found that the reading of the super classes and interfaces is performed yet again even if it is not parsed. So the amount of reading is the same and the less time spent for the solution without repository is probably caused by the lower overhead for the memory management, since the amount of used memory it is the only difference.

5.2.3 Memory usage

The amount of memory used was measured in a method deep in the computational tree of both class loading solutions to ensure that most of the objects were in use. Even with this approach, we could not measure exact numbers, because of different garbage collector policies. But for the general comparison, the obtained numbers are sufficient. Table 5.3 shows that the amount of the memory used on the BCEL solution and the BCEL solution with the fixing mechanism is significant. This is because of the complex structures storing the java classes' representations in the class repository.

JVM	Location	ASM		BCEL (orig)		BCEL (no bug)		BCEL (no rep.)	
		read	write	read	write	read	write	read	write
IBM 1.42	static	3.86	0.72	45.92	1.64	17.36	0.70	5.34	0.90
	temp	3.75	0.48	45.80	0.48	17.22	0.48	5.11	0.49
SUN 1.4	static	2.45	0.65	31.90	0.62	10.10	0.68	5.64	0.70
	temp	2.03	0.38	35.88	0.40	10.04	0.40	5.93	0.35
SUN 1.5	static	3.02	0.79	29.91	0.90	8.83	0.85	6.30	1.05
	temp	2.44	0.40	29.85	0.44	9.09	0.44	5.78	0.40

Table 5.2: IO operations' time (s) of compile-time rewriting of the core classes

JVM	ASM	BCEL (orig)	BCEL (no bug)	BCEL (no rep.)
IBM 1.42	7.2	69.4	69.4	6.8
SUN 1.4	2.5	43.2	56.9	3.4
SUN 1.5	3.1	50.3	63.6	3.1

Table 5.3: Maximum memory used (MB) on compile-time rewriting of the core classes

Therefore, we also present a fourth set of measurements for the BCEL solution with the fixing mechanism and also with repeated emptying of the class repository (BCEL (no rep.)). This leads to significantly less memory used, but also to a higher rewriting time, since the super classes and the interfaces of the rewritten class must be parsed again.

5.2.4 Conclusion

Generally, we see that the ASM solution achieves the best results in all measurements, but the BCEL solution, with several adjustments, can also achieve reasonable values.

If we compare the results from the IBM JDK 1.42 with the results from the Sun JDKs, we can see less rewriting time and memory efficiency in the IBM JDK. The amount of the memory depends on the internal representation of the classes, so the difference in memory use is JVM dependent. The difference in rewriting times is probably caused by the later start of the JIT compiler, as we have already explained in the previous section.

5.3 Start-up time

Table 5.4 shows the results of the start-up time benchmark. This benchmark is the most important illustration of the usability of the proposed solution. It shows what the benefit for the programmer is, if he uses runtime class rewriting using the ASM class loader instead of current compile-time approach for building and subsequent launching of the application. The measurements were performed only on Sun JDK 1.5, since we have already

Area	ASM	BCEL (bug)	Current Ibis
Java core rewriting + Ibis rewriting + App. build	25.47	35.95	975.00+
Implicit compile-time rewriting + App. build	25.47	35.95	82.90
App. build only	15.13	24.05	31.70

Table 5.4: Start-up time (s) of the application using ASM loading, BCEL loading or the current compile-time approach

shown a similarity among different JVMs.

The table contains build times of a test application with 50 classes to be rewritten. The first row shows the times for the first run of the application. The first run of the application using the runtime approach includes rewriting of Java core classes. Since the current compile-time approach does not implicitly rewrite Java core classes due to the limitations of the IOGenerator program, it was explicitly performed. Due to an insufficient memory error this rewriting failed after more than 10 minutes in about 75% of rewriting. The second row shows the implicit way of the first start of the application, which means excluding explicit rewriting of the Java core classes as done in the current compile-time approach. The last row shows the times when only the application is built and launched. For each build, we show the measurements using the ASM class loader, the BCEL (bug) class loader and the current compile-time rewriting approach using the IOGenerator program. The 50 classes used in the testing application are based on the same template as the classes in the stress testing benchmark.

Start-up using the ASM class loader is more than 3 times faster than the current compile-time solution if we launch the program for the first time using implicit way and more than 2 times faster if we launch the program a second time. The explicit rewriting of the Java core classes in the current compile-time rewriting has a launching time that is more than 39 times slower than the runtime ASM solution.

Chapter 6

Conclusions and future extensions

Conclusions

The Ibis system uses the compile-time rewriting approach to implement an efficient serialization mechanism. In this thesis we have examined if replacement of the compile-time rewriting approach with a runtime version is feasible. Moreover, we have tried to replace the current framework for bytecode manipulation with a more efficient solution to achieve a performance increase.

We have used Java's class loading mechanism to implement runtime rewriting. The use of a customized class loading process has certain disadvantages. First, we cannot force all classes to pass through the class loader, since the class loader needs some classes to be loaded in advance to implement its own functionality. Second, if we load a class twice using two different class loaders without mutual delegation, we can violate loading constraints resulting in typing problems in the application. Third, there are certain classes that cannot be loaded by a user class loader, because they are closely related to the internals of the JVM. Therefore, some of the classes must be rewritten at compile-time.

Another issue is the triggering mechanism which provides class loading based on the references from a loaded class to another class. Due to this mechanism, classes used for the Ibis serialization are loaded using the bootstrap class loader, because they are referenced from rewritten system classes. Therefore, we have to include these Ibis classes in the compile-time rewriting. In addition, it is necessary to check each change to the Ibis serialization implementation, because it can influence the class dependencies. If the dependencies are changed, the list of the delegated Ibis classes should be also changed.

Rewriting using the BCEL framework and the ASM framework differs

in their usability and performance. The BCEL framework is aimed to give an abstract view of the class representing it as the tree object structure. All classes are collected in a common repository, which also contains interclass dependencies. The ASM framework only supports a visitor approach. It passes parsed data directly to the visiting object. ASM does not contain any repository. Therefore, if we need to keep track of the interclass dependencies, we have to build our own repository. This also allows us to optimize the performance, since we can manipulate and store only the required information.

As part of this thesis, we have also implemented a version of runtime class rewriting. This implementation allowed us to perform the measurements and compare the efficiency of the runtime rewriting approach to the current compile-time approach. Using our solution, the programmer can achieve 2 times faster launching time, while he is developing and debugging the application. Moreover, if the programmer explicitly requires the rewriting of Java core classes using the current compile-time approach, he can achieve 38 times faster launching time.

An important benefit of the runtime solution is also for launching a job on a real grid. Since we do not necessarily know where the job is started, our runtime approach will save the time needed to rewrite the classes of Ibis framework before the first start of the job.

Future extensions

This section describes future extensions of this work. Firstly, there is the divide and conquer programming model called Satin, which is a part of the Ibis project. Satin rewrites programs into parallel versions. It uses the compile-time rewriting approach using the BCEL framework. Therefore, another interesting extension of this work would be to determine whether we can implement a Satin runtime rewriter using the ASM framework, for example, as an extension of the proposed rewriting mechanism. Since Satin requires more complex analysis of code, the implementation using ASM would also be complex with more than two visitors. On the other hand, it would not be necessary to rewrite Java core classes.

Secondly, certain research on replicated method invocation using the rewriting approach is now in progress. It would be interesting to examine if this rewriting can be also moved to load time.

Thirdly, we have introduced a certain level of abstraction in our class loading mechanism. Therefore, it is possible to create another class loader using some other bytecode manipulation framework for the rewriting.

Lastly, the Java 6 platform is coming soon. This should bring certain changes to the class file format. To recognize these changes, a new major version of the ASM framework will be introduced. In this new version, the

visitor approach, with only slightly modified interfaces, will be preserved.

Bibliography

[1] 'Eric Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journ'ees Composants 2002 : Syst'emes 'a composants adaptables et extensibles (Adaptable and extensible component systems)*, Nov. 2002.

[2] S. Liang and G. Bracha. Dynamic class loading in the Java virtual machine. In *Conference on Object-oriented programming, systems, languages, and applications (OOPSLA'98)*, pages 36–44, 1998.

[3] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification, Second Edition.* Addison-Wesley, Apr. 1999.

[4] J. Maassen, R. van Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, and R. Hofman. Efficient Java RMI for Parallel Programming. *ACM. Trans. on Programming Languages and Systems (TOPLAS)*, 23(6):747–775, November 2001.

[5] Sun Microsystems. Java™ Object Serialization Specification. `http://java.sun.com/j2se/1.4/pdf/serial-spec.pdf`, 2001.

[6] R. van Nieuwpoort, J. Maassen, G. Wrzesinska, R. Hofman, C. Jacobs, T. Kielmann, and H. E. Bal. Ibis: a Flexible and Efficient Java-based Grid Programming Environment. *Concurrency and Computation: Practice and Experience*, 17(7–8):1079–1107, 2005.

[7] Wikipedia, the free encyclopedia. Grid computing. `http://en.wikipedia.org/wiki/Grid_computing`, 2006.