Universidade Federal de Itajubá – Campus Itabira

Matheus Venturyne Xavier Ferreira

Automatic Computation Offloading of Java Applications

A Virtual Machine Agnostic Approach

Itabira

2016

Matheus Venturyne Xavier Ferreira

Automatic Computation Offloading of Java Applications
A Virtual Machine Agnostic Approach

Undergraduate Thesis presented as partial requirement to obtain the title of bachelor in Computer Engineering from Universidade Federal de Itajubá – Campus Itabira.

Advisor: Juliano Monte-Mor

Itabira
2016

Matheus Venturyne Xavier Ferreira

Automatic Computation Offloading of Java Applications
A Virtual Machine Agnostic Approach

This Undergraduate Thesis was judged, as partial requirement, to obtain the title of bachelor of Computer Engineering from Universidade Federal de Itajubá – Campus Itabira.

Grade obtained: ___________

Itabira, ___ of _________ from 2016.

Prof. Dr. Juliano Monte-Mor
Prof. Advisor

Prof. Dr. Sandro Izidoro
Universidade Federal de Itajubá – Campus Itabira

Prof. Dr. Fernando Afonso Santos
Universidade Federal de Itajubá – Campus Itabira

Abstract

Each year, we see the market growth of pervasive computing devices with limited resources, including smartphones and wearables. This work describes a system that allows centralized Java applications to execute as distributed applications exploiting underutilized resources on more powerful computers (e.g. computers and laptops). We present a prototype that makes use of dynamic partitioning, does not require modifications in the Java Virtual Machine, and supports the synchronization of multiple threads. To achieve our goal, we implemented a bytecode transformation system that modifies the user and system classes. The synchronization between machines is enforced by a Distributed Shared Memory (DSM). The execution on a Software Defined Networking (SDN) allows the enforcement of strong privacy and security guarantees to the offloaded code. We present tests with a face recognition application, with acceptable overhead, demonstrating the feasibility of the solution.

Keywords: Computation Offloading. Dynamic Partitioning. Java bytecode transformation. Distributed Shared Memory. Multithreading.

List of Figures

List of Tables

List of Abbreviations

API Application Program Interface.

CLR Common Language Runtime.

DSM Distributed Shared Memory.

HSL Hue Saturation Brightness.

ILP Integer Linear Programming.

IoT Internet of Things.

JIT Just-In-Time.

JNI Java Native Interface.

JVM Java Virtual Machine.

OpenCV Open Source Computer Vision Library.

QoE Quality of Experience.

RGB Red Green Blue.

RMI Remote Method Invocation.

RPC Remote Procedure Call.

SDN Software Defined Network.

TCP Transmission Control Protocol.

VM Virtual Machine.

Contents

1 Introduction

The smart-phone market is growing each year, and is expected that in 2016 there will be 2 billion users in the world, more than one fourth of the world population (EMARKETER, 2014). In addition, we live in a world of pervasive devices connected to the Internet developing multiple tasks such as processes control, and vital signals monitoring through health care devices. This Internet of Things (IoT), as we call, has been getting strength in the past years, expected to achieve a value of $7.1 trillions in 2020 (WORTMANN; FLÜCHTER, 2015). In addition, the applications run by these devices are each day more complex, requiring a huge amount of computational power and energy.

As possible solution, computational offloading allows computer programs to be partitioned and executed on more powerful devices. Offloading leverage the computational power of idle computers or powerful servers to execute code that otherwise would consume too much energy or would degrade performance. As example, consider a smartphone application executing a real-time face recognition algorithm, which requires high Quality of Experience (QoE) and low power consumption. The code responsible for processing the frames and generate the output could be executed on a laptop that is having its processor underutilized saving the phone's energy and providing a better experience.

With that in mind, our objective is to propose a system for computational offloading of Java applications that does not require modifications to the Java Virtual Machine (JVM), improving application performance and reducing power consumption. Application developers can run their application over our Java runtime and have computational power from other computers utilized to execute their application.

To achieve this goal, the system instruments classes by computing bytecode transformations when the virtual machine is loading user classes. In addition, we generate a patch of instrumented system classes. The resulting application do its computation in a Distributed Shared Memory (DSM) between the client application and surrogates collaborating with computational power.

The framework requires minimum work from developers, requiring only to annotate which methods to offload. System administrators can run the server runtime and allow users to offload their code to idle machines. If no server is available in the network, the application will execute gracefully as a normal local application. In addition, we developed the prototype with a Software Defined Network (SDN) in mind; what allows network administrators to enforce strong offloading policies.

We opted by using Java because it is a popular language and offers advanced elements present in modern programming languages, such as, automatic garbage collection and a reflection API. In addition, Java is hybrid language that combines compilation and interpretation. In other words, Java programs are compiled to bytecode and at the moment of execution interpreted by the JVM Just-in-Time (JIT) compiler. Technology advancements have made JIT compilers very efficient, doing runtime optimizations that are not possible for compiled languages. In addition, the compilation to bytecode offers machine portability allowing any architecture to receive and execute the transformed code.

In Section 2, we present the background and previous work in computation offloading. We describe important concepts, elements of the Java Virtual Machine, and previous work with their contributions and limitations. In Section 3, we describe the philosophy behind our prototype design and the challenges faced during the implementation. In Section 4, we present and discuss applications executed with our prototype. Finally, we conclude in Section 5, discussing the results, describing our contributions, and suggesting future directions.

2 Background

The goal of the work is to convert centralized applications in distributed applications preserving the original execution flow of the software. Distributed systems are used in multiple applications due to the increasing need of computational power and storage. We can define such systems as a set of nodes interconnected by a network. From the view of a specific node, other nodes and their resources are remote, whereas its own resources are local (SILBERSCHATZ; GALVIN; GAGNE, 2008).

In other words, they a combination of systems collaborating through messages with the goal of increasing the computational resources, to do parallel computing or enjoy resources that otherwise would not be available – as example, we can cite the access of printers over networks or a distributed file system. In addition, the communication environment can happen through the Internet or ad-hoc networks as Bluetooth.

It is important to ensure robustness in distributed systems. The system can suffer multiple kinds of hardware and software failures. The failure of a link, the failure of a site, and the loss of a message – due to software bugs or hardware fault – are the most common types. To provide robust, we should detect those problems, reconfigure the system to continue the computation, and finally, recover after the fault element is repaired (SILBERSCHATZ; GALVIN; GAGNE, 2008).

In other words, such failures can generate errors in the computation and make the system untrustworthy. In respect to the loss of messages, communication protocols – such as TCP/IP suite – allow a trustworthy communication channel, with automatic retransmission of lost messages and ordered message delivery. Link and site failures should be dealt by routing protocols such as distance vector routing and link-state routing utilized in the Internet protocol when routing over a small size network (PETERSON; DAVIE, 2011). For our system, in face of such failures, is settled a deadline for servers to return the result of the computation. Over timeout, the runtime executes the method and all future requests locally until the network advertises a new server.

In the next sections, we are going to discuss important concepts to the development of this work such as the use of Java as reflective programming language; Java class loaders; Software-Defined-Networks (SDNs); and previous research in computation offloading.

2.1 Java

Nowadays, Java is a popular programming language both for teaching, and in the industry due to its easy of use, good documentation, security, portability, and performance improvements of Just-in-Time compilation (JIT). There are many implementations, one the most popular being the OpenJDK.

A common result of the compilation of Java source code is bytecode for the Java Virtual Machine JVM, an application Virtual Machine (VM). The JVM is an environment where the bytecode is interpreted and executed in hardware. This layer of abstraction allows the same bytecode to execute in any computer architecture with an implementation of the Java Virtual Machine. Other implementations such as the Android Dalvik VM translates the Java bytecode in Dalvik bytecode (EHRINGER, 2010).

After the compilation, the Java compiler (`javac`) stores classes' bytecode in `class` files (LINDHOLM et al., 2014). During the execution of a Java application, the virtual machine searches and loads class files when the class is referenced. In the Java runtime, class loaders, section 2.1.5, are responsible by loading these new classes. Some tools, notably ASM (KULESHOV, 2007), allows the manipulation of Java bytecode, allowing the instructions to be modified to satisfy any requirement.

Using ASM, the modified bytecode can be stored on `class` files for future use or passed to the class loader when defining a new class. The present work focused on the transformation of Java bytecode, not Dalvik's, because the resulting Java bytecode can be translated to Dalvik bytecode using the `dx` tool. In addition, transformations directly over Java bytecode are simpler because Java instructions do computations in an operand stack allowing localized transformations while Dalvik is a register-based architecture with more complex instructions.

Reflection is another important property offered by Java, and other object-oriented languages, explored in this work. Programming language reflection consists in the ability of a computer program examine and modify itself. As example, with reflection, a Java application can: convert private methods to public; invoke methods without previously knowing its name; and access any information of a class, method or field available at runtime.

The bytecode transformations performed require knowledge of the Java VM instruction set listed at (LINDHOLM et al., 2014). Those instructions operate on values in the operand stack that can be of category 1, uses one field in the stack, or category 2, uses two fields in the stack. The categories for each Java type is resumed in table 1. Only doubles and longs are in category 2 because they consume 8 bytes in memory.

Table 1: Java VM Computational Categories.

Actual type	**Computational type**	**Category**
boolean	int	1
byte	int	1
char	int	1
short	int	1
int	int	1
float	float	1
reference	reference	1
returnAddress	returnAddress	1
long	long	2
double	double	2

Source: (LINDHOLM et al., 2014, p.29)

2.1.1 Classes

All Java types are represented by classes – even primitives and arrays. Each class has a type descriptor we will use throughout this work, table 2. Note the descriptor of array classes have one [token for each dimension and reference's descriptors starts with the L token and ends with the ; token.

Table 2: Type descriptor of some Java types

Java type	Type descriptor
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Object;
int[]	[I
Object[][]	[[Ljava/lang/Object;

Source: (KULESHOV, 2007, p.11)

All primitive and array classes are automatically generated by the virtual machine. For primitives and arrays, the Java class can be retrieved by appending `.class` to the end of the type name (e.g. the class of a boolean is `boolean.class` and the class of a double array is `double[].class`. When instrumenting classes, it is helpful to understanding the format of class files, table 3, because not all elements are mandatory and they follow the reproduced structure.

Table 3: Overall structure of a compiled class (*means zero or more)

Modifiers, name, super class, interfaces	
Constant pool: numeric, string and type constants	
Source file name (optional)	
Enclosing class reference	
Annotation*	
Attribute*	
Inner class*	Name
Field*	Modifiers, name, type
	Annotation*
	Attribute
Method*	Modifiers, name, return and parameter types
	Annotation*
	Attribute*
	Compiled code

Source: (KULESHOV, 2007, p.10)

2.1.2 Arrays

Arrays are a special kind of references to class objects generated by the virtual machine. For example, the class `[I` represents and array of int primitives that have `java.lang.Object` as base class.

Each array class has a component class, the class of the elements stored in the array. For the `[I` class, `int.class` is the component class. Arrays only have the public field `length`, and their elements can be accessed by the operator `[]`. In addition, the reflection library allows internal array manipulation through the `java.lang.reflect.Array` class.

2.1.3 Methods

In Java, each object has a reference to its class in memory. Through the class reference, methods can be invoked through references stored in a vtable. Similarly to classes, methods have their own descriptors used to distinguish methods with the same name, table 4. Methods marked with the modifier `final` cannot be overridden and the JIT compiler will attempt to inline them, improving performance.

Table 4: Sample method descriptors

Method declaration in source file	Method descriptor
void m(int i, float f)	(IF)V
int m(Object o)	(Ljava/lang/Object;)I
int[] m(int i, String s)	(ILjava/lang/String;)[I
Object m(int[] i)	([I)Ljava/lang/Object;

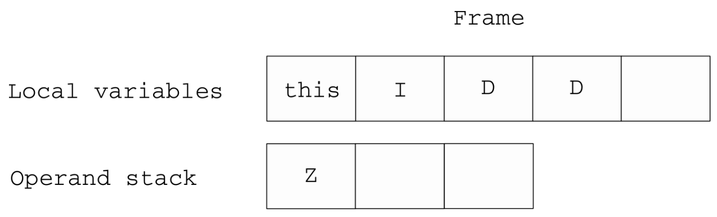
Source: (KULESHOV, 2007, p.12)

2.1.4 Frames

In the JVM, instructions execution happens in threads. Each thread has a frame stack where the computation takes place. Each time a method is invoked, a frame is pushed in the frame stack; when the method returns, the frame is popped out. In figure 1, we see an example of an execution frame. Each frame is composed by two sections of constant size, one for local variables and another for the operand stack. The frame size is computed at compile time. In addition, each local variable is addressed by a frame index. At runtime, the execution of instructions consume operands in the stack and can return values that are pushed in the stack.

It is worth to mention, as we see in the figure, doubles (of descriptor `D`), a data type of category 2, consume two fields in the frame. In addition, note the reference `this` occupies the first field in a non-static method's frame. A complexity in executing bytecode transformations in methods is related in remapping the local variables indexes and recomputing the number of local variables and the size to be allocated to the operand stack.

Figure 1: Thread frame



Source: Own making

2.1.5 Class Loaders

An important element in the Java virtual machine is the presence of class loaders – elements responsible by dynamically loading classes on demand in the virtual machine. With that in mind, all applications are initialized with a bootstrap class loader responsible by loading starter classes and protected packages[1]. In addition, the application developer can utilizes a custom class loader to load classes out of the class path if needed (e.g. from a web server).

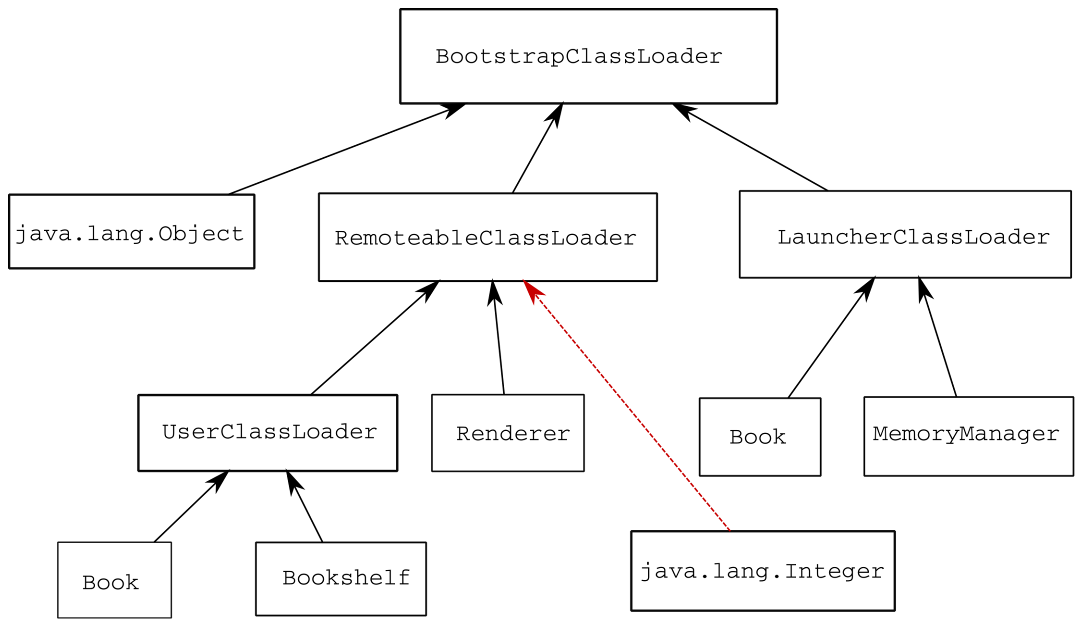
For our prototype, we implemented a `RemoteableClassLoader` responsible by loading the application classes and executing the instrumentation module. During this phase, the classes' bytecode is modified before the class definition. However, some precautions should be taken. All classes in Java are solely identified by their name and class loader – the class loader responsible by defining the class. In other words, during runtime, two classes with the same name can be present containing different bytecode implementations if they were loaded by different class loaders. To avoid a harmful security threat, the Java runtime ensures that in all operations two objects with the same name can only satisfy a type constraint if they were loaded by the same class loader; otherwise, a security exception is raised (LIANG; BRACHA, 1998).

To clarify, in figure 2, we see a possible state of class loaders in the system. In the figure, each box is a class with arrows pointing to the the parent class loader. Each Java class was loaded in some point in time by its parent class loader (note that even class loaders are themselves classes). The Bootstrap class[2] loader is a exception to the rule and is responsible by loading the initial classes into memory. In addition, note the class `Book` was loaded twice, which is perfectly fine and each one could have completely different implementations. Finally, as we pointed out before, classes in the `java` package can only be loaded by the bootstrap class loader; therefore, in the figure, the dashed arrow indicates an invalid state because `java.lang.Integer` could not be loaded by the `RemoteableClassLoader` and would throw a `SecurityException`. As consequence, we can only have one instance of each class in the `java` package.

[1]The Java runtime enforces classes on the `java` package to be loaded by the bootstrap class loader; otherwise, the runtime will trigger a security exception and stop the execution.

[2]The Bootstrap class loader is implemented internally in the JVM and is not a Java class; therefore, it cannot be accessed from Java code.

Figure 2: Class Loader



Source: Own making

As classes of same name loaded by different class loaders are independent, class loaders can enforce static fields isolation. For that, note that each static field is a class field. In other words, if we have the same class loaded by different class loaders, each one of the class static fields represent different regions in memory and will not conflict with each other.

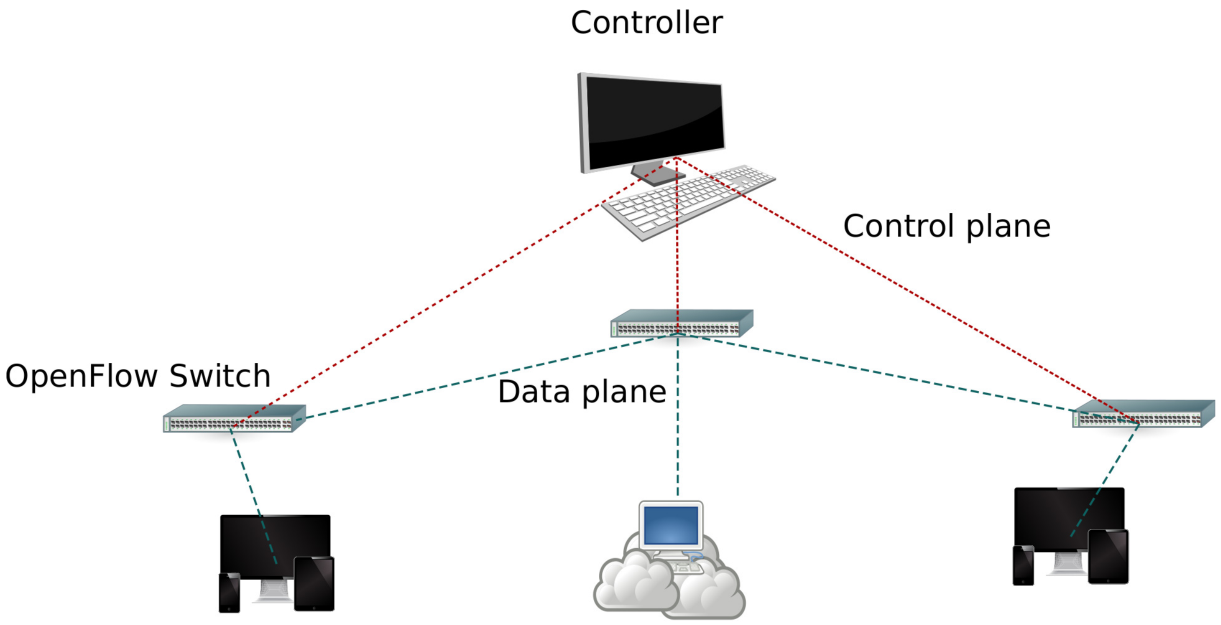
2.2 Software Defined Networking

Software Defined Networking (SDN) is a new network architecture proposed as solution for the current innovation stagnation in computer networking research mainly due to lack of abstractions and because of the complexity involved in network management (MCKEOWN et al., 2008). Currently, OpenFlow is the most successful specification for SDNs.

The key idea is the separation of the control plane from the data plane, figure 3, having the network controlled through a logically centralized controller, dispatching control messages to programmable switches. Switches keep a flow table where rules are installed. When a switch does not know what to do with a packet, the switch forwards the packet to the controller that decides what to do. In sequence, the controller install rules in switches, so future packets can be processed without asking the controller. The rules can have a timeout to indicates for how long they are valid.

Having a centralized view point of the network allows easier configuration and implementation of complex policies over the network. SDNs are already popular in corporation networks, such as B4, a backbone network for Google's data centers across the planet(JAIN et al., 2013), and has attracted the interest of others major network compa-

Figure 3: Software Defined Network



Source: Own making

nies.

In the past, ECOS (GEMBER; DRAGGA; AKELLA, 2012) was developed to support strong security and privacy for offloading applications leveraging SDNs. As example, an organization might offload applications only for local idle computer without disclosing to outside servers. In addition, a user might want only his laptop receiving offloaded code (e.g to do processing over his photos). Using a SDN controller, it is easy to self contain the network and enforces the interaction with trustworthy servers.

In real world applications, SDNs require OpenFlow enabled switches; however, for testing the prototype over a SDN, we used Mininet (LANTZ; HELLER; MCKEOWN, 2010), a system to easily prototype software defined networks. The application controller was built in Python using POX (MCCAULEY, 2011), a network software platform.

2.3 Computation Offloading

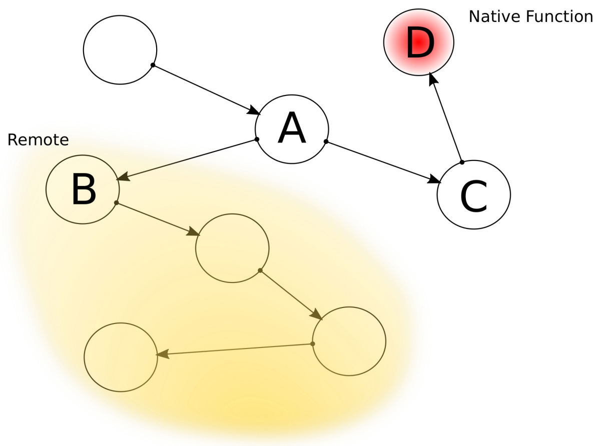
Computation offloading consist in requesting another computer (also called surrogate) to execute code on behalf of a client application. Previous research has focused on the feasibility, infrastructure and performance of offloading (KUMAR et al., 2013; WANG; CHEN; WANG, 2015). When comparing the different techniques already proposed, there are some points we should consider such as support to multithreading, nested migration and easiness of integration to current applications.

Partitioning, the process of deciding what code will execute locally or remotely, is the first step for computation offloading. The partition can be static or dynamic, it changes depending on parameters, such as, network availability, latency and throughput. It can use the help of application developers or use static or dynamic code analyses to decide possible partitioning candidates. Previous works, such as the MAUI runtime (CUERVO et al., 2010), have focused on modeling the partitioning problem as a linear programming (FERGUSON, 2000) (a technique for the optimization of linear objective functions) to minimize power consumption.

Sometimes, the application needs to execute native functions – i.e., functions often implemented in C/C++ with bindings allowing execution from Java. For example, the OpenGL API to Java actually are bindings to the OpenGL API implemented in C. API calls change native state or even the state of the graphic card – what cannot be read from the Java VM. Those functions should not be offloaded, unless we use I/O virtualization (SANI et al., 2014), because native state cannot be offloaded. When an offloaded function invokes a native function on its function invocation tree, it can only be offloaded if the system support nested migration – the surrogate requests the client to execute a function before he returns from the initial migration. However, even some native methods can be offloaded, such as math libraries, because they do not store native state; those are included on a whitelist and treated as non-native methods.

To illustrate, consider figure 4 representing a call graph. In the figure, each node represents a function and arrows represent function calls. The shaded region represents a partition with functions being executed remotely; shaded nodes are native function. If an offloading architecture does not support nested migration, the function C cannot be executed remotely because before it returns control to A, it calls D, a native function that cannot be executed remotely.

Figure 4: Call graph



Source: Own making

As example of previous offloading solutions, the Java Remote Method Invocation (RMI) (PITT; MCNIFF, 2001) is a Java API that allows the development of applications with the traditional client server architecture; however, adapt an application to the

RMI API requires rewriting the whole system because the developer should define what methods to offload. In addition, the partitioning is static and done in development time; a method defined to execute remotely can only execute remotely even if there is no server available.

For our goal of offloading code to improve performance, it is necessary to consider the latency and bandwidth of the network. Moreover, sometimes is more favorable to execute the code locally than offload because the cost would be too high. In other occasions, a server might not even be available.

Chroma (BALAN et al., 2003) is a coarse-grained solution, for remote executions in the order of seconds, that exposes an API for application developers where they can specify Remote Procedure Calls (RPCs). For that, they develop the concept of tactics, possible partitioning of the application the developer specifies. When running over the Chroma runtime, resource monitors help the system to adapt and choose which tactics to use for the application partitioning.

The solution proposed by Xiaohui (GU et al., 2004) rewrote the Java virtual machine to include transparent Remote Procedure Calls (RPCs) when the execution flow finds a partition. The solution goes further by providing a dynamic profiling of the application and deciding which partition to use during the execution what gives adaptability to memory limit in the Java heap, network bandwidth, and response time. The system also allows nested RPCs, and access to static data is always done in the master VM.

More recently, the MAUI runtime (CUERVO et al., 2010), was built with the goal of minimize power consumption in smartphones. Requiring only the application developer to annotate methods that can be offloaded, MAUI was implemented for *C#*, a high-level language that offers important properties shared with Java; notably reflection, and compilation to the Common Language Runtime (CLR) – similar to bytecodes, gives an indirection allowing code to be executed on any hardwares. MAUI offloads static fields and has a profiler that, if necessary, monitors the program and the network, changing the program partition through the solution of an Integer Linear Programming (ILP). As a limitation, MAUI does not support synchronization between threads.

The coarse-grained technique used by CloneCloud (CHUN et al., 2011) consist in the migration of a VM state. Using a static analyzer, they addressed the limitation of MAUI of depending in the developer to annotate functions to offload. CloneCloud works on thread granularity by migrating the execution state of the virtual machine application. As a limitation, CloneCloud does not support nested migration; therefore, given a offloaded function, all the functions in its call-graph should not relies on local features or share native state with functions being executed in another machine.

COMET uses a fine-grained approach by using a Distributed Shared Memory (DSM) instead Remote Procedure Calls (RPCs) (GORDON et al., 2012). By modifying the Dalvik VM, the virtual machines are synchronized supporting thread migration without changes of the bytecode.

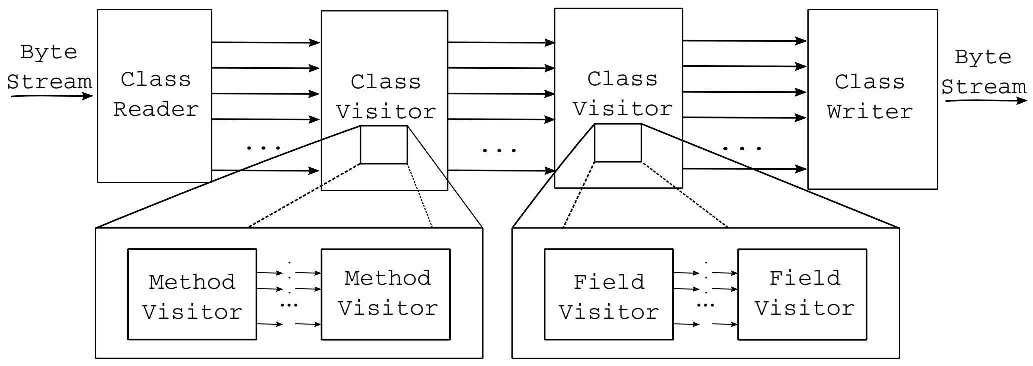
When offloading code, it is important to consider the threat model involved. The isolation offered by application virtual machines gives protection to servers receiving malicious offloaded code; however, the client application willing to have code offloaded might want to ensure privacy over sensitive data. By leveraging software defined networks, ECOS (GEMBER; DRAGGA; AKELLA, 2012) proposed a security scheme to ensure that offload only happens to trustworthy servers.

2.4 Bytecode Transformation

The work was grounded on previous research towards tools to manipulate Java bytecode. We opted by ASM (KULESHOV, 2007) because it offers a low-level API to manipulate bytecode, giving high flexibility in what transformations to perform. In addition, ASM can easily be used to implement static analyzers. In fact, others high-level bytecode manipulation tools use ASM in their implementation.

The figure 5 illustrates the interface provided by ASM which uses an event-based API. When we want to perform a transformation or analyses over the bytecode we pass the class byte array to a Class Reader. While the class reader reads the class file, it generate events that are fed to class visitors chained in the output. The events follow the sequence of elements present in the class files, table 3. These events include the discovery of the class signature, fields, methods, and many other possible events. Moreover, when a method is discovered, a class visitor can redirect the events inside the method to a chain of method visitors that can receive events corresponding to Java instructions. In Appendix C, we present a creation of a method visitor inside a class visitor. In the example, we generate and feed events to the method visitor with the goal of generating the body of a new method.

Figure 5: Chained bytecode transformations



Source: Own making

Finally, we can use a class writer, a special class visitor, to write the results of the transformations on a new byte array. Over performance penalty, we can specify class writers to automatically recompute the stack and local variables section of each method's frame.

The event based API gives better performance than a tree API, where all the bytecode is loaded to memory before starting the computation. Being possible to compute transformations efficiently allow transformations and static analysis to be performed when the application is loaded. As alternative, we could perform transformations previously and save the modified bytecode back to a `class` file, removing the overhead. As we discuss at section 3.4, because only the Bootstrap class loader can load system classes, transformations to classes in the `java` package should be precomputed and saved in class files.

3 Implementation

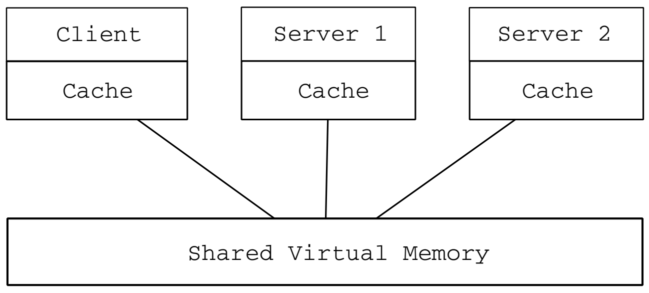
The prototype consists of a runtime system built on top of the Java VM. A computer systems willing to execute offloaded code runs the runtime server and an application that is going to be partitioned executes the runtime client. Once the server is executing, it advertises the service to the SDN application. The SDN controller is responsible for configuring the network and connect clients with the servers. With this solution, the offloading architecture does not have to care about security and privacy policies that should be enforced over offloaded code once the SDN application can enforce those policies over the network (GEMBER; DRAGGA; AKELLA, 2012).

On the client side, the runtime receives the fully qualified name (e.g. the package and the class name) of the Java class – the class where is the entry point –, the entry point (often `public void main(String args[])`) and the class path of the runtime environment. All classes in the runtime class path are loaded by our `RuntimeClassLoader` which has its bytecode modified and analyzed by ASM (KULESHOV, 2007), a Java bytecode manipulation framework. During the bytecode manipulation phase, methods annotated as `Remoteable` has the code replaced by a Remote Procedure Call (RPC).

To synchronize memory between virtual machines, we built a Distributed Shared Memory (DSM) where the local memory of each node is a cached version of the (DSM), figure 6. During synchronization points – methods invocation and return, monitor enter and exit – any dirty memory in the local cache is advertised to the other nodes.

As a current limitation, in Java, volatile is a special modifier for object and class fields that prohibit the field from being cached. Volatile variables ensure that modifications in one thread will automatically be visible in all other threads. For non-volatile variables, the compiler can cache the value making writes not visible to other threads. To support volatile variables in our system, we require that any write to a volatile variable becomes a synchronization point where the dirty field is always advertised to other nodes.

Figure 6: Distributed Shared Memory



Source: Own making

In the DSM, objects are addressed by a global unique ID. Dirty memory is tracked on field granularity. If an object field is written, we advertise the new field value to the other participants in the computation. We track dirty fields with a bitset for each object.

Static fields are tracked with a bitset for each class. Similarly, a whole array is considered dirty if any index receives a new value.

To simplify the implementation, we modify the classes so that all non-array objects implement the interface `RemoteableObject`. This allow the runtime, to manipulate and execute common operations to all objects.

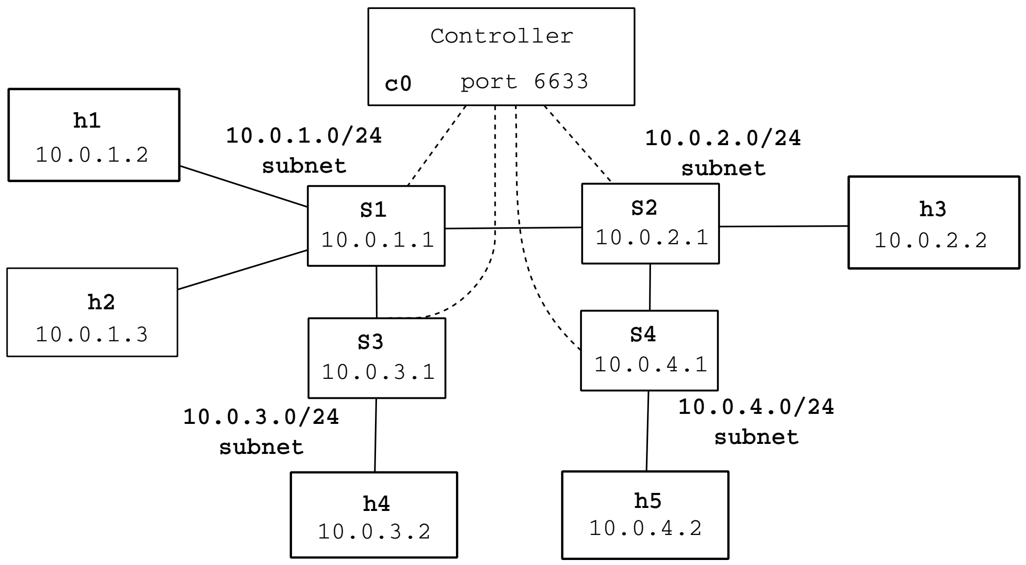
In the following sections, we will talk about the implementation of the communication protocol, section 3.2; the application partitioning, section 3.3; the bytecode transformations, section 3.4; the distributed shared memory, section 3.5; and the modifications to the garbage collection policies, section 3.6.

3.1 Network

As we discussed before, we run our tests over a Software Defined Network (SDN). The network topology we use for tests is presented at figure 7. In the figure, there is five hosts (e.g. `h1`), four switches (e.g. `s1`) and the controller `c0`. Note each switch has a direct communication channel with the controller (the control plane).

Each hosts is a terminal where a client or a server might run. The topology is created with Mininet, source code in appendix A.1. The controller was implemented in Python, appendix A.2, and runs at TCP port 6633. The controller implemented configure the switches as routers and the routing table is hard-coded in the source code.

Figure 7: Network Topology



Source: Own making

3.2 Communication

The communication between server and clients happen through a protocol we developed where the basic unity of communications happens through messages, Appendix B.1. Each message has fields to identify the sender and the destination and a message ID generated by a monotonic increasing counter. The ID is used to do the synchronization between client and servers. As example, when the client requests a server to execute code, the client thread will block until a timeout happens or the client receives a response. As we can possible have more than one thread waiting for a response, the reply should have the request ID so we can unblock the correct thread, Appendix B.2.

3.3 Application Partitioning

The current prototype uses a dynamic partitioning and requires the developer to annotate which methods can be offloaded. An annotated method will be offloaded if a connection is available and will execute all the invoked methods remotely until it returns with the result of the computation. If the invocation fails due to a timeout or other errors in the server, the method is executed locally.

3.4 Transformations

In this section, we list all the transformations performed over user and system classes. A major challenge present in implementing the offloading architecture with bytecode transformations is due to system classes. We define system classes as Java classes present in the package `java` and any of its sub-packages. As security policy, the Java specification states that system classes can only be loaded by the bootstrap class loader which our runtime does not have access. As work around, all the system class transformations are performed and saved on disk before the runtime is started.

Although we cannot load system class with our `RemoteableClassLoader`, we can specify the bootstrap class loader to load our modified system classes in place of the originals. Another challenge is the fact that most of our runtime environment was implemented in Java and we can only have one instance of each system class loaded in the JVM. It means the code responsible by managing the runtime state will itself be executing modified code. To avoid conflicts, we implemented isolation policies that will be described at section 3.4.3.

3.4.1 Object State

In our distributed application, object references are useless. The JVM normally uses the memory address of an object as the reference identification. As we cannot control where in memory an object will be allocated we should use another way of identification. For that, we use a global ID that is common for a given object to all nodes participating in the computation. We use a monotonic increasing counter to assign IDs. To avoid collisions and allow each network node to allocate objects, each object global ID has the node ID appended at the most significant bits.

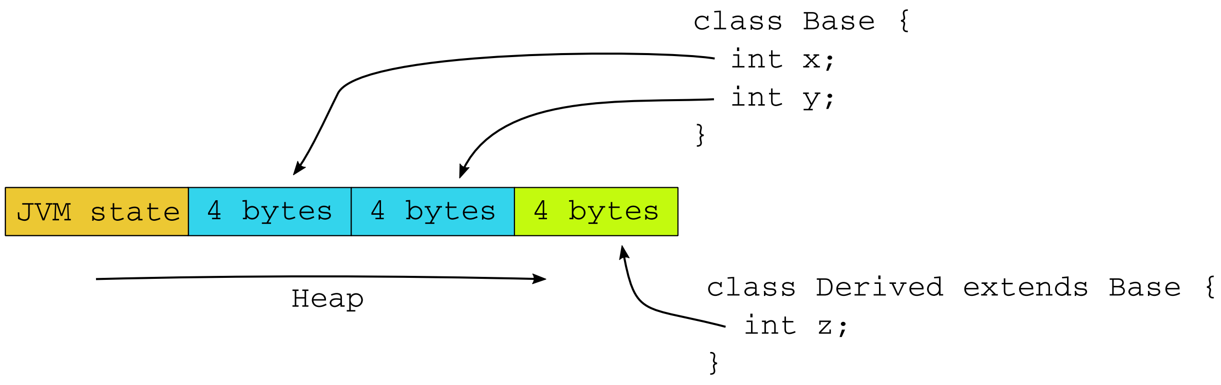
In figure 8, we see how an object is stored in memory. In addition to object fields, the JVM would store other variables to control internal object state such as the reference to the object's class. When a class extends another, the extra fields are stored to the right of the base class fields. In our solution, as we do not have control over the implementation of the JVM we should add additional state as object fields and store in the field section of the object.

A problem we faced when implemented the runtime was that some system classes rely in the field order when executing native code. As example, for the `java.lang.Integer` class, the JVM expects the first field to be an integer primitive of 4 bytes. As consequence, the less intrusive solution would be to add the extra fields to the right of the original fields. For that, each class we instrument, we make it implements the interface `RemoteableObjectBase` and copy the extra fields to the end of the class, figure 9. To avoid variable shadowing[3], we ensure that the only the first non-abstract class on a class hierarchy implements `RemoteableObjectBase` and have the extra fields.

Generating all the bytecode to implement an interface manually can be time consuming. Another alternative, is to compile a class with the fields and methods we want to add and copy the resulting bytecode to the destination class. The class implementing the interface `RemoteableObjectBase`, `RemoteableObjectBaseImplementation`, is reproduced at appendix D.4.

To avoid name collisions and consequently variable shadowing, we add a preamble to each field and method injected in the bytecode. The fields injected includes the global ID, a bitset to track the object dirty fields and a bitset to track the class dirty fields.

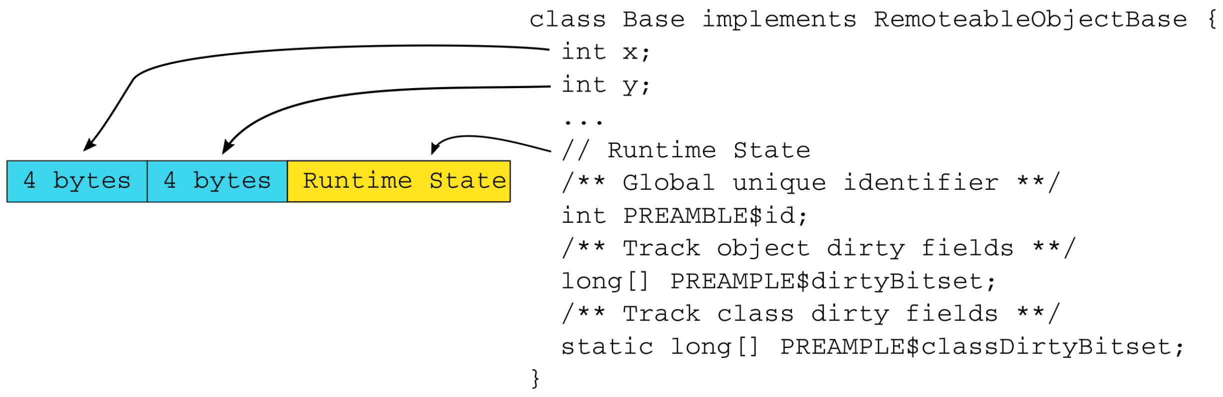
Figure 8: Objects internal memory



Source: Adapted from http://www.programcreek.com/2011/11/what-do-java-objects-look-like-in-memory

[3]Variable shadowing occurs when in a class hierarchy two or more classes have fields with the same name. As consequence, only the field in the super class will be visible.

Figure 9: Instrumented object



```
class Base implements RemoteableObjectBase {
  int x;
  int y;
  ...
  // Runtime State
  /** Global unique identifier **/
  int PREAMBLE$id;
  /** Track object dirty fields **/
  long[] PREAMPLE$dirtyBitset;
  /** Track class dirty fields **/
  static long[] PREAMPLE$classDirtyBitset;
}
```

Source: Own making

For arrays, a special type of objects, with no class we can instrument, we keep an array table that map array references to `RemoteableArrays` that stores the global ID. The interface implemented by objects and the bytecode transformations responsible by adding the additional state to classes are in appendix D. In table 5, we have an example of mapping local references to global IDs.

Table 5: Global IDs

Master		Surrogate	
Object Reference	Global ID	Object Reference	Global ID
@232204a1	1	@633204a2	1
@12166213	2	@faba3c12	2
...			
@31252351	153	@fafdsa32	153

Source: Own making

3.4.2 Configurations

The current implementation of the runtime does not support full instrumentation of all system classes. As example, the `java.lang.String` is used during the virtual machine initialization and during that phase we cannot invoke our runtime environment methods. Find the best alternative to support all Java classes should be object of future works.

At the moment, we use a XML file with the configurations of what classes should be instrumented and which shouldn't. Using a configuration file has many advantages because the developer cannot annotate[4] classes he did not write either because the source code is not available or because licenses does not allow. By default, we instrument all classes outside the `java` package. Bellow we have an example of configuration file.

[4]Annotations are syntactic metadata that can be added to source code.

```xml
<?xml version="1.0"?>
<settings>
  <classes>
    <class name="java/util/ArrayList">
      <methods>
        <method name="readObject">
          <options>
            <!-- If the method is called directly by native code without prior
    constructor initialization the method should be set for initialization -->
            <initialize>true</initialize>
          </options>
        </method>
      </methods>
    </class>
  </classes>

  <filters>
    <!-- Classes that should not be instrumented either due to VM conflicts or because
     require further analysis -->
    <filter name="Not Instrument">
      <pattern>\Aorg/bytedeco/javacpp/.*\Z</pattern>
      <pattern>\Aorg/bytedeco/javacv/.*\Z</pattern>
      <pattern>\Ajava/lang/Object\Z</pattern>
      <pattern>\Ajava/lang/ClassLoader\Z</pattern>
      <pattern>\Ajava/lang/Class\Z</pattern>
      <pattern>\Ajava/lang/SecurityManager\Z</pattern>
      <pattern>\Ajava/io/ObjectInputStream\Z</pattern>
      <pattern>\Ajava/io/SerializablePermission\Z</pattern>
      <!-- sun is a dangerous, non-portable, library and should be avoided -->
      <pattern>\Asun/.+\Z</pattern>
    </filter>
    <!-- JDK classes that should be instrumented -->
    <filter name="Instrument">
      <pattern>\Ajava/lang/(Boolean|Byte|Character|Double|Float|Integer|Long|Short)\Z<
    /pattern>
      <pattern>\Ajava/lang/Number\Z</pattern>
      <pattern>\Ajava/lang/String\Z</pattern>
      <pattern>\Ajava/lang/ArrayList\Z</pattern>
      <pattern>\Ajava/lang/Hashtable\Z</pattern>
      <pattern>\Ajava/awt/Point\Z</pattern>
      <pattern>\Ajava/awt/geom/Point2D\Z</pattern>
    </filter>
    <!-- Classes that should have the bytecode logged -->
    <filter name="Log Instrumentation">
      <pattern>.*</pattern>
    </filter>
    <filter name="Not Track State">
      <pattern>\Ajava/lang/String\Z</pattern>
    </filter>
  </filters>
</settings>
```

The runtime state for each object is initialized in their constructor, such as the assignment of a global ID and the allocation of the bitset; moreover, one of the elements of the configuration file allows the developer to specify other methods we should attempt to initialize an object – an attempt to initialize an object twice is ignored. This is necessary because in native code, objects can be instantiated without calling a constructor, not initializing the object global ID and dirty bitsets what will cause faulty behavior. As example found, an instance of `ArrayList` class can invoke its `readObject` method without previously calling a constructor. In the configuration file, we specify that the object should be initialized when invoking `readObject` so the program executes properly.

We can still specify instrumentation filters. The filters use a combination of regular expressions that are combined and compiled at runtime. The are filters to specify classes that should not be instrumented; system classes that should be instrumented; classes that should have the bytecode logged for debugging; and classes that we should instrument but not track internal state. As example, `java.lang.String` is an immutable class that does not need to track internal state, but should implement the `RemoteableObjectBase` interface so it is inserted both in the `Instrument` and `Not Track State` filters. In addition, as we said, `java.lang.String` cannot invoke our runtime environment methods during initialization, what is avoided by its insertion in the `Not Track State` filter (our runtime environment methods are invoke only when tracking state).

3.4.3 Thread

The runtime executes user code on a `RemoteableThread`. It is meanly meant to track the state of execution on a safe environment without the risk of race conditions. User classes that extends `java.lang.Thread` are modified to extend our thread implementation. Direct instantiations of `java.lang.Thread` class are replaced by the instantiations of `RemoteableThread`.

The `RemoteableThread` implementation has a array cache, discussed in section 3.4.5, used to track array writes. In addition, it has a flag to identify if the system is running in runtime mode. As we discussed before, a limitation of our solution is that instrumented system classes are running both for user code[5] and runtime code because we can have only one instance of those classes in the virtual machine. This will make the runtime code try to track state of itself causing inconsistent behavior. We created the abstraction of runtime mode to avoid this kind of conflict; therefore, if the thread is running in runtime mode, it should ignore any attempt to track state.

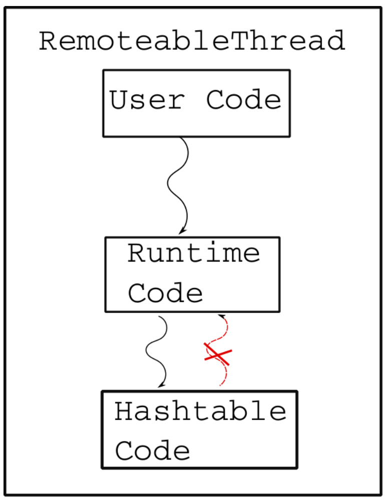
```
package org.matheus.runtime;
```

[5]We consider user code the code implemented by the application developer.

```java

public class RemoteableThread extends Thread {
  private ArrayCache arrayCache = new ArrayCache();
  private boolean runtimeMode = false;

  public RemoteableThread(String name) {
    super(name);
  }

  public RemoteableThread(Task task) {
    super(task);
  }

  public final boolean isArrayCached(Object obj) {
    return arrayCache.array == obj;
  }

  public final boolean isRuntimeMode() {
    return runtimeMode;
  }

  public final void setRuntimeMode(boolean flag) {
    runtimeMode = flag;
  }

  public final void cacheArray(Object obj) {
    arrayCache.array = obj;
  }

  public void clearCache() {
    arrayCache.array = null;
  }

  static class ArrayCache {
    Object array;
  }

}
```

As example, consider figure 10. In the figure, user code is running in a `RemoteableThread` and it invokes runtime code to notify about some state change. The implementation of our runtime uses the `Hashtable` class, which we should track state when running for user code but not for runtime code. When the runtime invokes a `Hashtable` method, it will trigger internal state changes that will invoke the runtime back; however, that is undesirable because we want to track state only of user objects. To avoid that, the thread enters in runtime mode when the runtime is first invoked and leaves only when returning control to the user code. While in runtime mode, the `Hashtable` attempts to call the runtime will be ignored.

Figure 10: Runtime Mode



Source: Own making

3.4.4 Field writes

To ensure memory consistence between virtual machines executing code, we should track when objects are modified. We do that by tracking when fields of objects and classes[6] change. For that, Java uses the instruction `PUTFIELD` to store a value in an object field and the instruction `PUTSTATIC` to store a value in a class field. We developed bytecode transformations to set the dirty bitset of an object or class when a field is written. The piece of code used for the transformation is reproduced in appendix E.

As example, consider the code bellow. We have a function where the field `a` receives a new value.

```java
class Foo extends Object {
  int a;

  public void bar(){
    this.a = 5;
  }
}
```

After the transformations, we have the resulting code bellow. The first step in the transformation was to add the local variables `object` and `value` to store the reference of the object being modified and the new value. After changing the value, we access the object's dirty bitset and check if the field was already marked as dirty – the verification minimizes runtime invocations. In the bitset, the first bit identify if the object has a dirty field. In addition, note that the position of `a` in the bitset is one[7]. If the field is not

[6]Static fields are actually class fields.

[7]The position for each field in the bitset is precomputed before execution.

marked as dirty yet, we invoke a runtime call responsible by adding the reference to a list of dirty objects.

```java
class Foo extends Object implements RemoteableObjectBase {
  int a;

  public void bar(){
      RemoteableObjectBase object;
      double value;

      object = this;
      value = 5;
      this.a = 5;

      if((object.dirtyBitset[0] & (1 << 1)) == 0){
        RemoteableRuntime.putField(object, object.dirtyBitset, value, 1);
      }
  }
}
```

The table 6 illustrates a small portion of the bytecode transformations behind the code above. Before storing the value `v1` in the field of `v2`, we duplicate the values in the stack with the instruction `DUP_X1` and store a copy in the local variables using `ISTORE` and `ASTORE`. Finally, `PUTFIELD` consumes the values of `v1` and `v2` and we reload than back to the stack with `ALOAD` and `ILOAD`.

Table 6: PUTFIELD bytecode transformation

Instruction	Operand Stack
`DUP_X1`	$v_2 v_1$
`ISTORE 1`	$v_1 v_2 v_1$
`DUP_X1`	$v_1 v_2$
`ASTORE 2`	$v_2 v_1 v_2$
`PUTFIELD` $MyObject.field_1$	$v_2 v_1$
`ALOAD 2`	
`ILOAD 1`	v_2
mark field as dirty	$v_2 v_1$

3.4.5 Arrays

To track writes to array cells, we use a course grained approach, tracking the array as a whole. If any index is written, we add the array to the list of dirty objects and add to the thread cache, section 3.4.3. The cache minimizes the overhead of writing state. If the array is present in the cache, we do not need to notify the runtime that the array is dirty again. A fine grained approach would track which individual indexes are dirty; however, the overhead of tracking the state of each index would be prohibitive. We opted to copy the whole array even if only one index was written to minimize the overhead of tracking state when writing to arrays. The source code to instrument bytecode instructions that write to array is presented in appendix F.

3.4.6 Remote calls

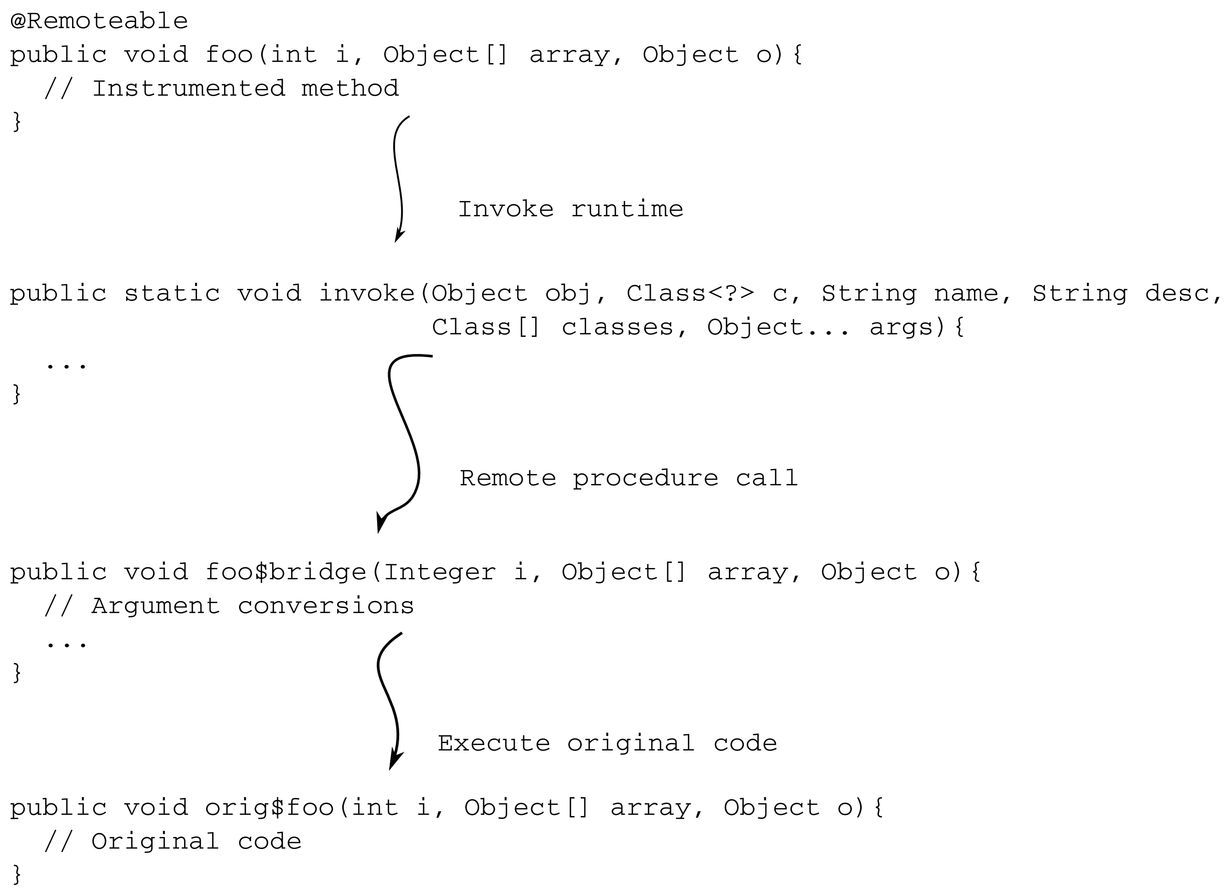
When instrumenting the methods that can be executed remotely, we compute transformations that generate the result depicted in figure 11. In the figure, the method `foo` would initially be executed locally and with the transformations it will execute remotely. To achieve this result, we generate two additional methods, `foo$bridge` and `orig$foo`[8]. We copy the original bytecode from `foo` to `orig$foo` so that the former will execute remotely when the surrogate receives the remote call.

At the body of `foo`, we generate an invocation to the runtime method `invoke`. `invoke` will receive the following arguments: `obj`, a reference to the object invoking `foo` or null if it is a static method; `c`, the class of the method being invoked; the method's name and descriptor; `classes`, the arguments' class; and finally, `args`, the method's arguments. Note that it involves converting primitives to their object versions (e.g. an `int` is converted to `java.lang.Integer`). This is necessary because `invoke` should receive an arbitrary number of parameters through the use of varargs. In Java, varargs are implemented by an array and array of `Objects` are the only type that can store any data type.

In the server, the runtime invokes the method `foo$bridge` responsible by converting objects back to primitives. And finally, the bridge method invokes the original method with the parameters received. The implementation of this transformations are in appendix C.

[8]The character $ is often used by compiler generated fields and methods.

Figure 11: Remote method invocation


```
@Remoteable
public void foo(int i, Object[] array, Object o){
  // Instrumented method
}
```

```
public static void invoke(Object obj, Class<?> c, String name, String desc,
                          Class[] classes, Object... args){
  ...
}
```

```
public void foo$bridge(Integer i, Object[] array, Object o){
  // Argument conversions
  ...
}
```

```
public void orig$foo(int i, Object[] array, Object o){
  // Original code
}
```

Source: Own making

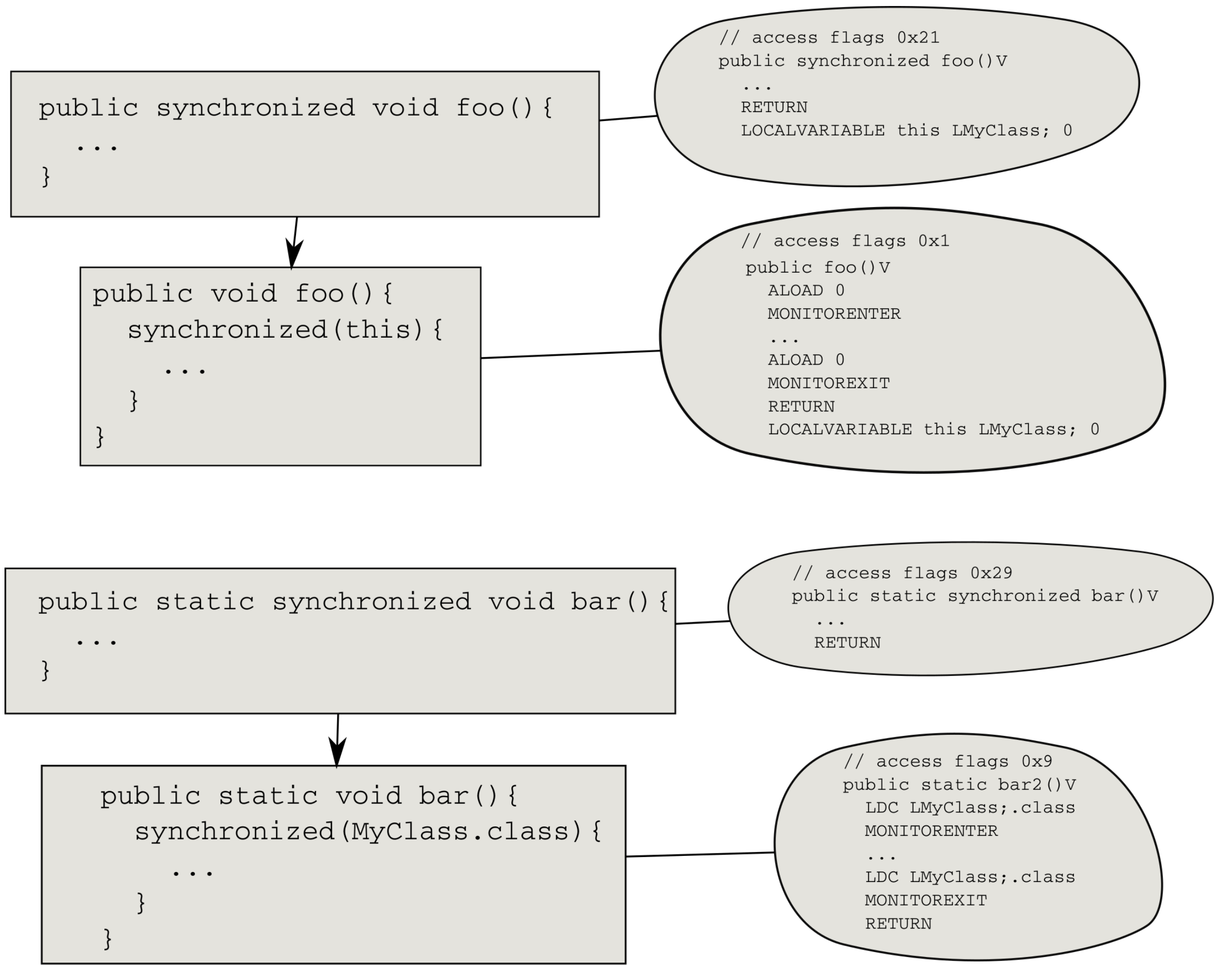
3.4.7 Multithreading

The prototype also supports offloading of multi-threaded applications. Such applications require the protection of critical sections. In the instruction set of the Java VM, synchronization is enforced by the instructions `MONITORENTER` and `MONITOREXIT`. These instructions receive an object reference as parameter which means all objects in Java have a monitor. In addition, synchronization can be enforced by synchronized methods; and by the methods `Object.wait()`, `Object.notify()`, `Object.notifyAll()` – all implemented natively and do not rely on the instruction set support.

In Java, all classes, fields and methods have an access flag that represents their properties. A synchronized method has the synchronized bit marked in its access flag, and when the method is invoked, the Virtual Machine receives this information. To enforce synchronization in our runtime we firstly replaces synchronized methods by the equivalent function with all its content surrounded by a synchronized block, figure 12. In the figure, even though we modify the bytecode directly, for clarity, the Java code is depicted in the left and the equivalent bytecode in the right. In addition, non-static methods are synchronized by the object monitor and static methods are synchronized by the class monitor

– in fact, all Java classes are itself objects of the class `java.lang.class`.

Figure 12: Synchronized methods



```
public synchronized void foo(){
  ...
}
```

```
// access flags 0x21
public synchronized foo()V
  ...
  RETURN
  LOCALVARIABLE this LMyClass; 0
```

```
public void foo(){
  synchronized(this){
    ...
  }
}
```

```
// access flags 0x1
public foo()V
  ALOAD 0
  MONITORENTER
  ...
  ALOAD 0
  MONITOREXIT
  RETURN
  LOCALVARIABLE this LMyClass; 0
```

```
public static synchronized void bar(){
  ...
}
```

```
// access flags 0x29
public static synchronized bar()V
  ...
  RETURN
```

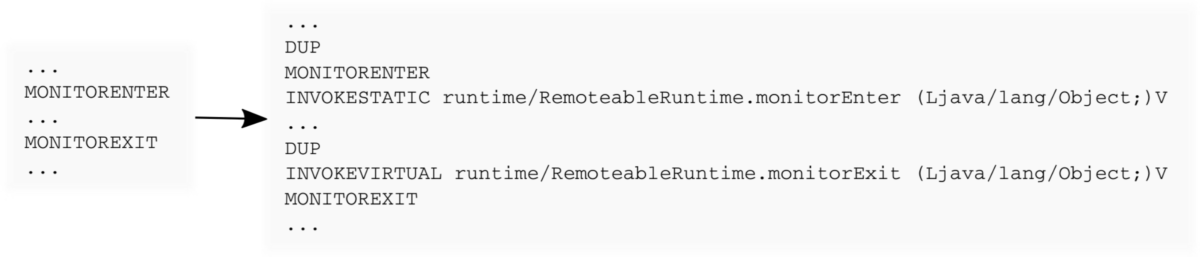
```
public static void bar(){
  synchronized(MyClass.class){
    ...
  }
}
```

```
// access flags 0x9
public static bar2()V
  LDC LMyClass;.class
  MONITORENTER
  ...
  LDC LMyClass;.class
  MONITOREXIT
  RETURN
```

Source: Own making

Once the previous transformation is performed, we apply the transformation in figure 13. The transformation wraps all instructions `MONITORENTER` and `MONITOREXIT` by our runtime implementation of monitor enter and exit to ensure that the monitor is acquired from the Distributed Shared Memory (DSM). The `RemoteableRuntime` will delegate any attempt to enter a monitor to the client application and secure proper synchronization.

Figure 13: Bytecode transformation to Distributed Monitor



```
...
MONITORENTER
...
MONITOREXIT
...
```

```
...
DUP
MONITORENTER
INVOKESTATIC runtime/RemoteableRuntime.monitorEnter (Ljava/lang/Object;)V
...
DUP
INVOKEVIRTUAL runtime/RemoteableRuntime.monitorExit (Ljava/lang/Object;)V
MONITOREXIT
...
```

Source: Own making

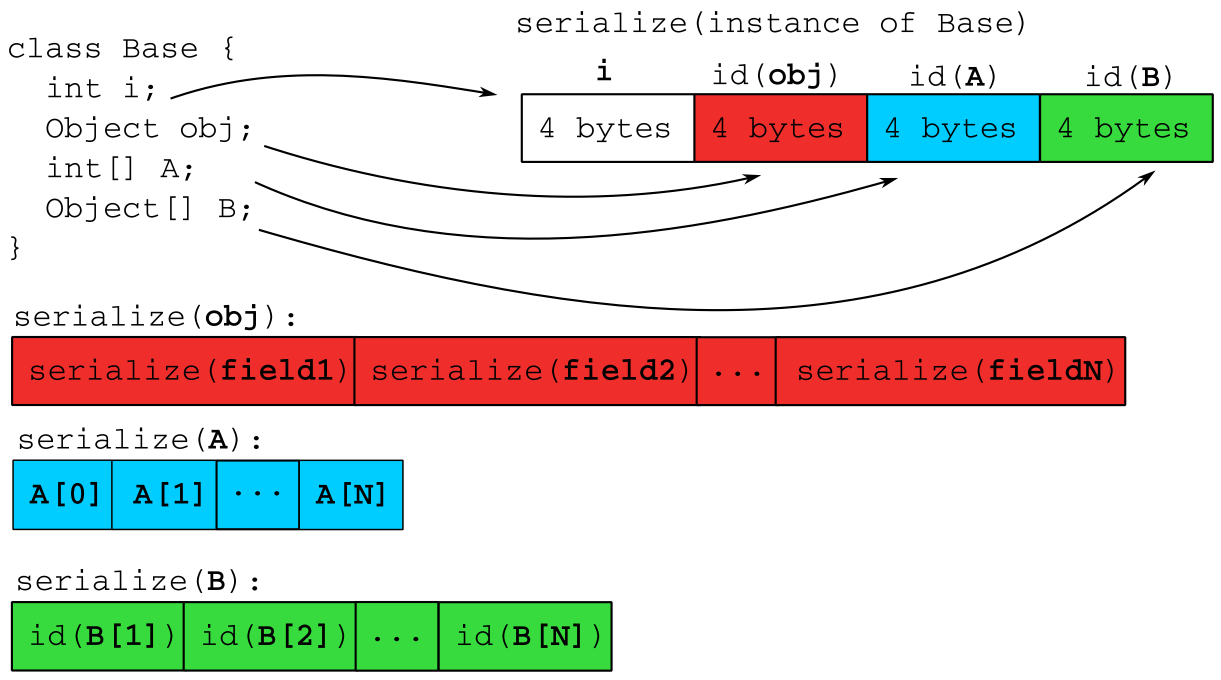
Finally, the last group of synchronization mechanism – `Object.wait()`, `Object.notify()` and `Object.notifyAll()` – is not currently supported. In future works, we plan to implement wrappers to allow waiting and receiving notifications from any object in the DSM.

3.5 Distributed Shared Memory

As we mentioned before, for each object we assign a global ID that is common for a given object through all virtual machines participating in the computation. Here we present how this objects are synchronized between machines. The implementation uses our own serialization class to serialize objects and classes, figure 14.

In the figure, when serializing the non-array object we get an array of 16 bytes. The first position stores the primitive value; the second stores the global ID of the field `object`; and finally, the third and fourth position stores the global ID of the arrays `integers` and `objects`. In sequence when the array `integers` is serialized, we actually send the array as it is because it is an array of primitives. For arrays of references, such as `objects`, we send an array of integers representing the global ID of each reference. For the reference `object`, we apply the same algorithm.

Figure 14: Object Serialization



Source: Own making

The first step is ensure when the fields of an object are dirty, we advertise the new value. A field value can be primitive (e.g. int, boolean, char) or another object. If the new value is a primitive, we send the primitive value; otherwise, we send the object's global ID because references only have meaning inside a given virtual machine. When the other runtime receives a field update, it writes the value in the object field using the reflection library.

For primitive arrays, we send the whole array and copy to the destination using `System.arraycopy()`, a native method that copies the array's block of memory. Arrays of objects are converted to arrays of global IDs before transmission and are recovered index by index at the destination.

An important element in the implementation consists in the capacity of allocating objects without constructor invocation which is possible through the Java Native Interface (JNI), appendix I. JNI is a C/C++ API that allows manipulation of the virtual machine state from native code. The `sun.reflect.ReflectionFactory` class offers an implementation of the native call for us.

When we send a global ID, we have no insurance the object is allocated on the destination machine. As we will discuss at section 3.6, about garbage collection, even if we send all the objects when they are allocated, we cannot guarantee they were not garbage collected when they are needed. We could define a police where we store strong references of objects to avoid garbage collection; however, that would leak memory. To overturn this situation, we send the whole reference tree of all objects referenced by a memory write.

This solution increase the amount of memory we have to send, but unfortunately by only using bytecode transformations we have little control over the garbage collector behavior.

3.6 Garbage Collection

For the management of objects, the runtime system keeps a table of all instantiated objects. The objects are inserted on their creation and removed when the runtime detects the garbage collector reclaimed the object's memory. The garbage collector is a system responsible by freeing the memory of objects that are not referenced anymore. Reference to objects are often stored in strong references. In addition, Java allows the storage of object references in soft (`SoftReference<?>`), weak (`WeakReference<?>`) or phantom (`PhantomReference<?>` references to give the developer some control over the garbage collector.

All those additional references can be registered with a `ReferenceQueue<?>` where reference objects are appended by the garbage collector when the reachability change is detected. This allow the system to detect when an object is collected and execute clean ups. This is a better alternative than the `finalize` method, declared in `java.lang.Object`, invoked when an object is about to be garbage collected. `finalize` is not recommended because it creates a delay when an object is marked for collection and when it is collected. In addition, it allows an object to resuscitate by storing its reference in a strong reference avoiding the garbage collection. This trick saves the object but the `finalize` method is never invoked again by the garbage collector. In addition, `finalize` is executed on a separated thread. If there is too many objects implementing their `finalize` method, the overhead of calling `finalize` becomes very high.

At each iteration of the garbage collector, it executes an algorithm to test the reachability of each Java object. Most garbage collector implementations cause the effect `stop-the-world` where all the execution stops while the garbage collector executes. For a given object, the execution of the garbage collector might result in three possible outcomes: strong referenced objects cannot have memory reclaimed; soft referenced objects can be reclaimed when the Virtual Machine is running out of memory and they will be freed before a `OutOfMemoryError` is raised; and finally, weak referenced, phantom referenced or unreferenced objects can be garbage collected any time.

Our runtime keep references to all objects to track their state. If we use strong references to store the references, we would leek memory. To allow memory to be collected, we make use of weak references instead of strong references. Each reference is registered to a reference queue that notifies when an object is garbage collected. However, even though an object is garbage collected in the local cache, it can still be strong referenced on another virtual machine; therefore, we ask to all virtual machines if the object is strong referenced. If not, we can clean up the global ID and allow the use by another object.

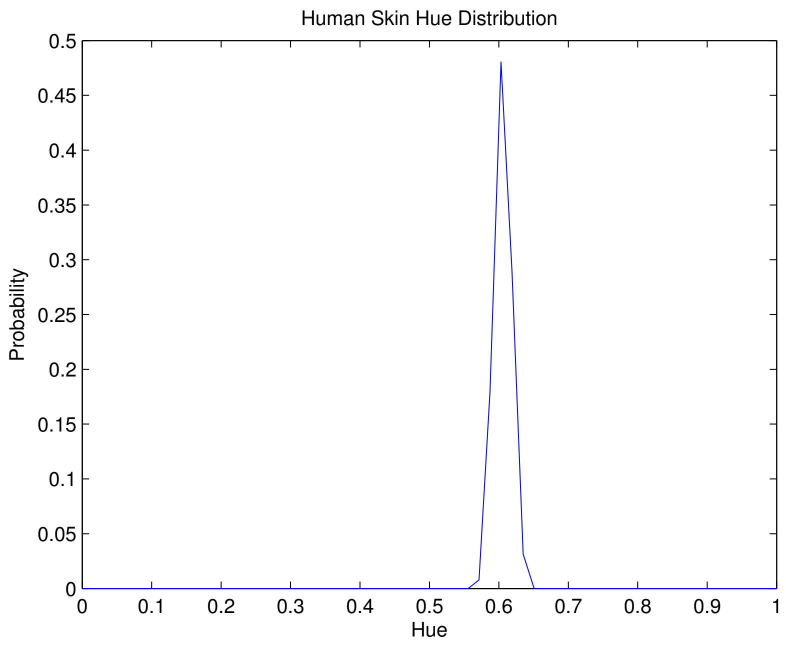
4 Evaluation

To test the feasibility of the offloading architecture for real world applications, we implemented the CAMSHIFT algorithm (BRADSKI, 1998), a computer vision algorithm for face tracking. This example, in special, represent an important class of applications, when offloading, due to the copy of video frames, requires high network bandwidth, and considerable processing over the frame data. In addition, the application requires access to the Open Source Computer Vision Library (OpenCV) API whose access is made through native functions to acquire webcam images.

The algorithm relies on samples of human skin to extract the human skin hue probabilistic distribution. Computer images are represented in the RGB (Red, Green, and Blue) space composed by 1 byte for each color and 1 byte for the alpha channel. Converting each pixel to the HSL (Hue, Saturation, and Brightness) space (SMITH, 1978) and extracting the hue channel, $H \in [0, 1]$, we built the one dimension histogram of the hue distribution from skin samples, as shown in Figure 15. The normalized histogram represents the hue probabilistic distribution of the human skin. As example, from the distribution, a pixel with a hue of 0.6 has high probability of belonging to a human skin.

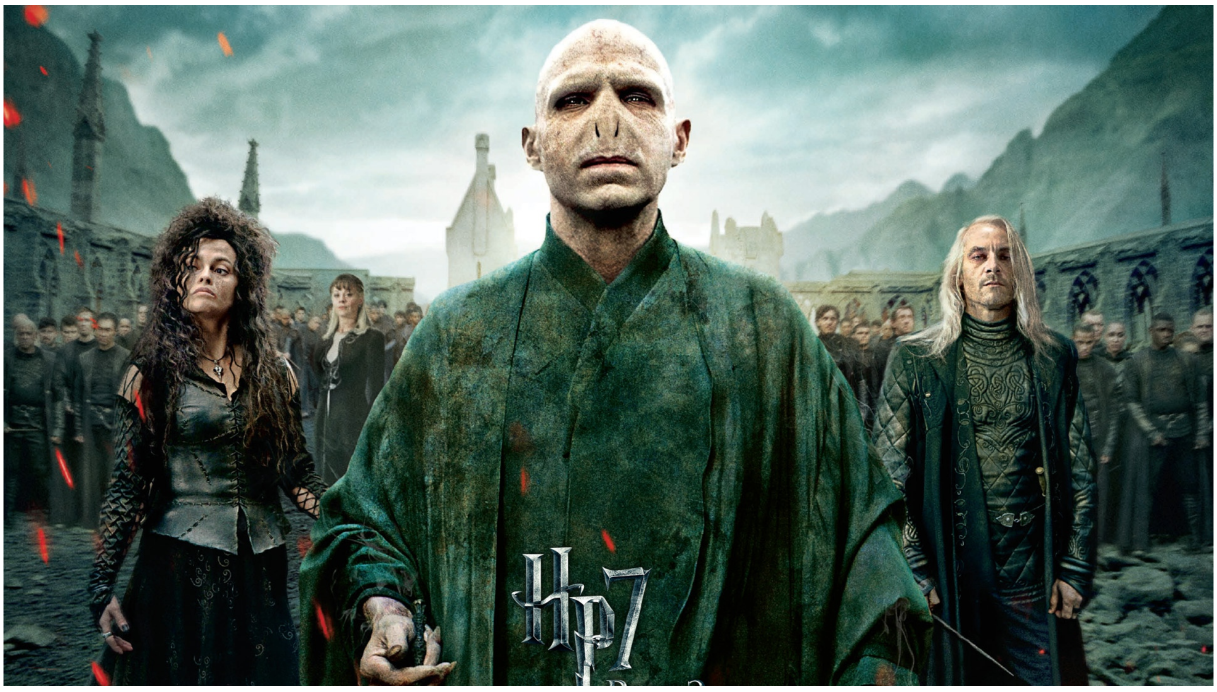
With the histogram, each image or video frame, Figure 16, is converted to the HSL space, the hue channel is extracted, and we lookup at the distribution the probability of each hue sample belonging to a human skin. The result is depicted at Figure 17 – where whiter pixels mean higher probability the pixel was from a human skin. To find the human face, we climb the gradient of the probability distribution to find the peak (BRADSKI, 1998). In another example, at Figure 18, the runtime is used to face track in a webcam streaming outputting an acceptable performance.

Figure 15: Probabilistic distribution of human skin hue



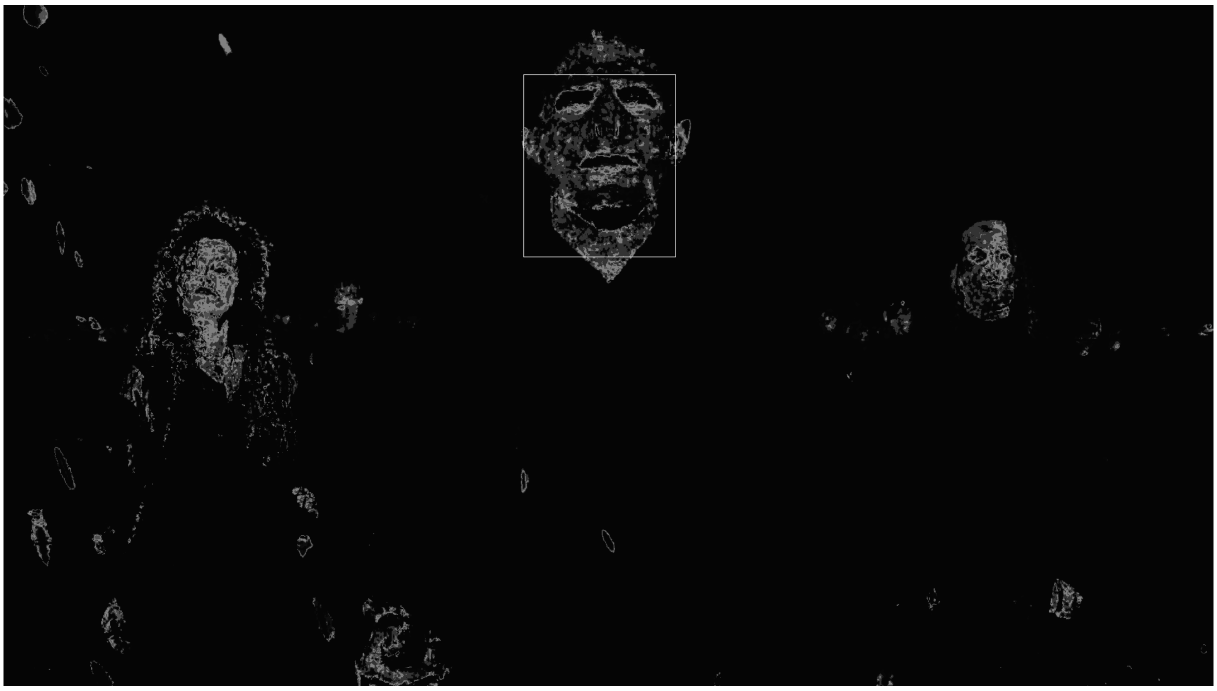
Source: Own making

Figure 16: Image to detect human face



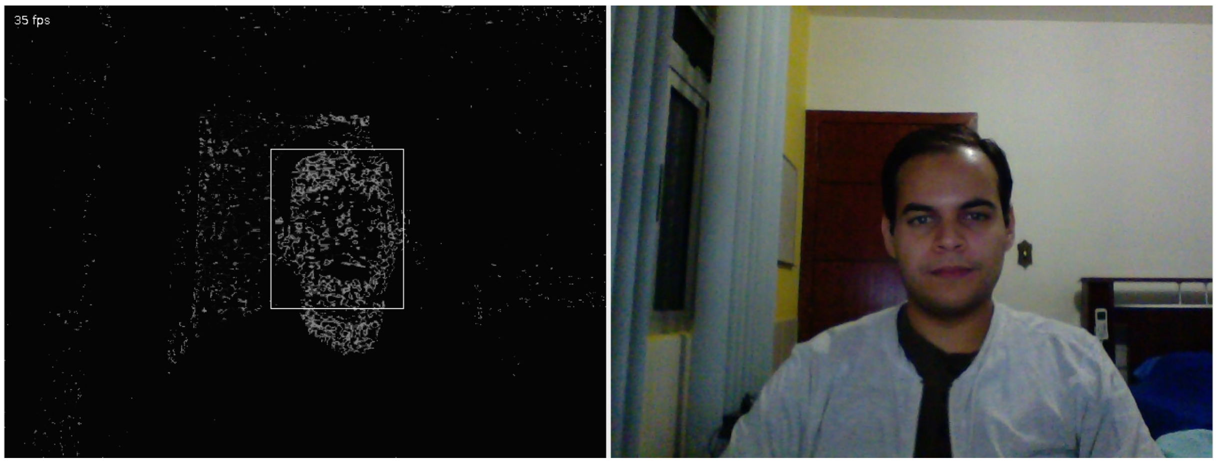
Source: Available at <http://www.flashcoo.com/movie/2011_07_Harry_Potter_7_Part2/index.html>

Figure 17: Image human skin hue probabilistic distribution



Source: Own making

Figure 18: Webcam face tracking



Source: Own making

5 Conclusion

In this work, we proposed a runtime system for computation offloading of Java applications with the goal of improving performance and reduction of power consumption. To validate the proposed system a prototype was implemented in Java which we used to execute a computer vision application for face recognition. Our system extended previous work by supporting multi-threaded applications without the need of changes of the Java Virtual Machine, modifying only the bytecode loaded by our runtime class loader. In addition, we prototyped a SDN controller which allowed the offloading over a network offering strong privacy and security guarantees.

In future works, we could explore better code partitioning algorithms, considering the impact of offloading parameters of a given size; and considering the frequency a method is executed. We could also explore how to remove the burden over the developer of annotating methods to offload. Another important point is how to easily leverage computational resources that are accessed by native functions. In addition, we could study how better explore the instruction set and the use of natives functions to improve our runtime system performance with more efficient bytecode transformations and data structures. The present work also does not support all elements of the Java language specification such as volatile variables and synchronization with `wait` and `notify`. Another promising line of work is to explore the consequences of network or virtual machine failures. As example, we should have mechanisms to recover resources, such as, objects monitors from machines not responsive. Finally, an important line of research is how to address the current limitations and challenges we faced by doing bytecode transformations over system libraries.

References

BALAN, R. K. et al. Tactics-based remote execution for mobile computing. In: ACM. *Proceedings of the 1st international conference on Mobile systems, applications and services*. [S.l.], 2003. p. 273–286.

BRADSKI, G. R. *Computer Vision Face Tracking For Use in a Perceptual User Interface*. 1998.

CHUN, B.-G. et al. Clonecloud: elastic execution between mobile device and cloud. In: ACM. *Proceedings of the sixth conference on Computer systems*. [S.l.], 2011. p. 301–314.

CUERVO, E. et al. Maui: making smartphones last longer with code offload. In: ACM. *Proceedings of the 8th international conference on Mobile systems, applications, and services*. [S.l.], 2010. p. 49–62.

EHRINGER, D. The dalvik virtual machine architecture. *Techn. report (March 2010)*, v. 4, 2010.

EMARKETER. Smartphone users worldwide will total 1.75 billion in 2014. *eMarketer*, April 2014. Disponível em: <http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536>.

FERGUSON, T. S. Linear programming: A concise introduction. *UCLA [online] http://www. math. ucla. edu/˜ tom/LP. pdf*, 2000.

GEMBER, A.; DRAGGA, C.; AKELLA, A. Ecos: leveraging software-defined networks to support mobile application offloading. In: *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. New York, NY, USA: ACM, 2012. (ANCS '12), p. 199–210. ISBN 978-1-4503-1685-9.

GORDON, M. S. et al. Comet: Code offload by migrating execution transparently. In: *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. [S.l.: s.n.], 2012. p. 93–106.

GU, X. et al. Adaptive offloading for pervasive computing. *Pervasive Computing, IEEE*, IEEE, v. 3, n. 3, p. 66–73, 2004.

JAIN, S. et al. B4: Experience with a globally-deployed software defined wan. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 43, n. 4, p. 3–14, ago. 2013. ISSN 0146-4833. Disponível em: <http://doi.acm.org/10.1145/2534169.2486019>.

KULESHOV, E. Using the asm framework to implement common java bytecode transformation patterns. *Aspect-Oriented Software Development*, 2007.

KUMAR, K. et al. A survey of computation offloading for mobile systems. *Mob. Netw. Appl.*, Springer-Verlag New York, Inc., Secaucus, NJ, USA, v. 18, n. 1, p. 129–140, fev. 2013. ISSN 1383-469X. Disponível em: <http://dx.doi.org/10.1007/s11036-012-0368-0>.

LANTZ, B.; HELLER, B.; MCKEOWN, N. A network in a laptop: rapid prototyping for software-defined networks. In: ACM. *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. [S.l.], 2010. p. 19.

LIANG, S.; BRACHA, G. Dynamic class loading in the java virtual machine. In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM, 1998. (OOPSLA '98), p. 36–44. ISBN 1-58113-005-8. Disponível em: <http://doi.acm.org/10.1145/286936.286945>.

LINDHOLM, T. et al. *The Java virtual machine specification*. [S.l.]: Pearson Education, 2014.

MCCAULEY, J. *POX Controller*. [S.l.], 2011. Disponível em: <http://www.noxrepo.org/pox/about-pox/>.

MCKEOWN, N. et al. Openflow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review*, ACM, v. 38, n. 2, p. 69–74, 2008.

PETERSON, L. L.; DAVIE, B. S. *Computer Networks, Fifth Edition: A Systems Approach*. 5th. ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN 0123850592, 9780123850591.

PITT, E.; MCNIFF, K. *Java. rmi: The Remote Method Invocation Guide*. [S.l.]: Addison-Wesley Longman Publishing Co., Inc., 2001.

SANI, A. A. et al. Rio: a system solution for sharing i/o between mobile systems. In: ACM. *Proceedings of the 12th annual international conference on Mobile systems, applications, and services*. [S.l.], 2014. p. 259–272.

SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. *Operating System Concepts*. 8th. ed. [S.l.]: Wiley Publishing, 2008. ISBN 0470128720.

SMITH, A. R. Color gamut transform pairs. *SIGGRAPH Comput. Graph.*, ACM, New York, NY, USA, v. 12, n. 3, p. 12–19, ago. 1978. ISSN 0097-8930. Disponível em: <http://doi.acm.org/10.1145/965139.807361>.

WANG, Y.; CHEN, I.-R.; WANG, D.-C. A survey of mobile cloud computing applications: Perspectives and challenges. *Wirel. Pers. Commun.*, Kluwer Academic Publishers, Hingham, MA, USA, v. 80, n. 4, p. 1607–1623, fev. 2015. ISSN 0929-6212. Disponível em: <http://dx.doi.org/10.1007/s11277-014-2102-7>.

WORTMANN, F.; FLÜCHTER, K. Internet of things. *Business & Information Systems Engineering*, v. 57, n. 3, p. 221–224, 2015. ISSN 1867-0202. Disponível em: <http://dx.doi.org/10.1007/s12599-015-0383-3>.

Appendix A Software Defined Network

A.1 Network Topology

Mininet implementation of the network topology in figure 7.

```python
"""Network Topology

Adding the 'topos' dict with a key/value pair to generate our newly defined
topology enables one to pass in '--topo=mytopo' from the command line.
"""

from mininet.topo import Topo

class RouterTopo(Topo):
  "Simple topology example."

  def __init__(self):
    "Create custom topo."

    # Initialize topology
    Topo.__init__(self)

    # Add hosts and switches
    h1 = self.addHost('h1', ip='10.0.1.2/24', defaultRoute='via 10.0.1.1')
    h2 = self.addHost('h2', ip='10.0.1.3/24', defaultRoute='via 10.0.1.1')
    h3 = self.addHost('h3', ip='10.0.2.2/24', defaultRoute='via 10.0.2.1')
    h4 = self.addHost('h4', ip='10.0.3.2/24', defaultRoute='via 10.0.3.1')
    h5 = self.addHost('h5', ip='10.0.4.2/24', defaultRoute='via 10.0.4.1')

    s1 = self.addSwitch('s1')
    s2 = self.addSwitch('s2')
    s3 = self.addSwitch('s3')
    s4 = self.addSwitch('s4')

    # Add links
    self.addLink(h1, s1)
    self.addLink(h2, s1)
    self.addLink(s1, s2)
    self.addLink(s1, s3)
    self.addLink(s2, s4)
    self.addLink(s2, h3)
    self.addLink(s3, h4)
    self.addLink(s4, h5)

topos = { 'mytopo': (lambda: RouterTopo()) }
```

A.2 SDN Controller

Implementation of the network controller. This implementation makes OpenFlow switches operate as routers. For our tests we use a hard-coded static routing table.

```python
# Copyright 2012 James McCauley
# Copyright 2016 Matheus Venturyne Xavier Ferreira
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at:
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
This component is for use with the OpenFlow tutorial.

It acts as a simple hub, but can be modified to act like an L2
learning switch.

It's roughly similar to the one Brandon Heller did for NOX.
"""

from pox.core import core
import pox.openflow.libopenflow_01 as of
from pox.lib.packet import *
from pox.lib.packet.arp import *
from pox.lib.packet.icmp import *

from router_table import RoutingTable
from arp_cache import ARPCache, ARPQueueEntry

if core is not None:
    log = core.getLogger()

title = ['subnet', 'host', 'iface', 'iface_ip', 'port']
table_s1 = [title,
            ['10.0.1.2', '10.0.1.2', 's1-eth1', '10.0.1.1', 1],
            ['10.0.1.3', '10.0.1.3', 's1-eth2', '10.0.1.1', 2],
            ['10.0.2.1/24', '10.0.2.1', 's1-eth3', '10.0.1.1', 3],
            ['10.0.3.1/24', '10.0.3.1', 's1-eth4', '10.0.1.1', 4]]

table_s2 = [title,
            ['10.0.1.1/24', '10.0.1.1', 's2-eth1', '10.0.2.1', 1],
            ['10.0.4.1/24', '10.0.4.1', 's2-eth2', '10.0.2.1', 2],
            ['10.0.2.2', '10.0.2.2', 's2-eth3', '10.0.2.1', 3]]

table_s3 = [title,
            ['10.0.1.1/24', '10.0.1.1', 's3-eth1', '10.0.3.1', 1],
            ['10.0.3.2', '10.0.3.2', 's3-eth2', '10.0.3.1', 2]]

table_s4 = [title,
            ['10.0.2.1/24', '10.0.2.1', 's4-eth1', '10.0.4.1', 1],
            ['10.0.4.2', '10.0.4.2', 's4-eth2', '10.0.4.1', 2]]

switch_to_table = {}

switch_to_table['s1'] = RoutingTable('s1', table_s1)
switch_to_table['s2'] = RoutingTable('s2', table_s2)
switch_to_table['s3'] = RoutingTable('s3', table_s3)
switch_to_table['s4'] = RoutingTable('s4', table_s4)


class Router (object):
    """
    A Router object is created for each switch that connects.
    A Connection object for that switch is passed to the __init__ function.
    """

    def __init__(self, connection, routing_table):
        # Keep track of the connection to the switch so that we can
        # send it messages!
        self.connection = connection

        # This binds our PacketIn event listener
        connection.addListeners(self)

        # Use this table to keep track of which ethernet address is on
        # which switch port (keys are MACs, values are ports).
        self.mac_to_port = {}

        # Routing table
        self.routing_table = routing_table

        self.routing_table.dump()

        # Interfaces
        self.interfaces = self.routing_table.port_to_iface.values()

        # ARP cache
        self.arp_cache = ARPCache()

        # Message queue
        self.msg_queue = {}

    def resend_packet(self, packet_in, out_port):
        """
        Instructs the switch to resend a packet that it had sent to us.
        "packet_in" is the ofp_packet_in object the switch had sent to the
        controller due to a table-miss.
        """
        msg = of.ofp_packet_out()
        msg.data = packet_in

        # Add an action to send to the specified port
        action = of.ofp_action_output(port=out_port)
        msg.actions.append(action)

        # Send message to switch
        self.connection.send(msg)

    def find_interface_by_ip(self, ip):
        for iface in self.interfaces:
            if iface.ip == ip:
                return iface
        return None

    def find_interface_by_mac(self, mac):
        for iface in self.interfaces:
            if iface.mac == mac:
                return iface
        return None

    def enqueue_msg_for_arp(self, ip, packet, **kwargs):
        if ip not in self.msg_queue:
            self.msg_queue[ip] = list()

        self.msg_queue[ip].append(ARPQueueEntry(packet, kwargs))

    def act_like_router(self, packet, packet_in):
        """
        Implement router-like behavior.
        """
        log.debug('Received a packet src(%s) dst(%s)' %
                  (packet.src, packet.dst))

        # Learn the port for the source MAC
        self.mac_to_port[packet.src] = packet_in.in_port

        if packet.dst in self.mac_to_port:
            self.send_flow_mod(packet=packet, packet_in=packet_in,
                               port=self.mac_to_port[packet.dst])
        elif packet.dst == ETHER_BROADCAST:
            if packet.type == ethernet.ARP_TYPE:
                self.handle_arp_packet(packet=packet, packet_in=packet_in,
                                       protocol=packet.payload)
            else:
                self.send_flow_mod(packet=packet, packet_in=packet_in, port=of.
    OFPP_ALL)
        else:
            protocol = packet.payload

            if packet.type == ethernet.ARP_TYPE:
                self.handle_arp_packet(packet, packet_in, protocol)
            elif packet.type == ethernet.IP_TYPE:
                self.handle_ip_packet(packet, packet_in, protocol)

    def handle_arp_packet(self, packet, packet_in, protocol):
        log.debug('handle_arp_packet %d' % protocol.opcode)

        if protocol.opcode == arp.REQUEST:
            self.handle_arp_request(packet, packet_in, protocol)
        elif protocol.opcode == arp.REPLY:
            self.handle_arp_reply(packet, packet_in, protocol)

    def handle_arp_request(self, packet, packet_in, protocol):
        log.debug('handle_arp_request')
        interface = self.find_interface_by_ip(protocol.protodst)

        # Not for us
        if interface is None:
            self.send_flow_mod(packet=packet, packet_in=packet_in, port=of.OFPP_ALL)

            return

        log.debug('handle_arp_request')

        msg = of.ofp_packet_out()
        action = of.ofp_action_output(port=packet_in.in_port)
        msg.actions.append(action)

        packet.dst = packet.src
        packet.src = interface.mac
        protocol.hwsrc = packet.src
        protocol.hwdst = packet.dst
        protocol.protosrc, protocol.protodst = protocol.protodst, protocol.protosrc
        protocol.opcode = arp.REPLY

        msg.data = packet

        self.connection.send(msg)

    def handle_arp_reply(self, packet, packet_in, protocol):
        log.debug('handle_arp_reply %s' % str(protocol.hwsrc))

        self.arp_cache.insert(protocol.protosrc, protocol.hwsrc)
        msg_list = self.msg_queue.pop(protocol.protosrc)

        for entry in msg_list:
            msg = entry.msg
            packet = entry.packet
            packet.dst = protocol.hwsrc

            self.send_flow_mod(packet=packet, packet_in=msg['packet_in'], port=
    packet_in.in_port,
                               match=msg['match'])
#             self.send_packet_and_install_flow(packet, packet_in.in_port,
#                                               msg['match'])

    def handle_ip_packet(self, packet, packet_in, protocol):
        log.debug('handle_ip_packet srcip(%s) dstip(%s)' % (protocol.srcip, protocol.
    dstip))
        interface = self.find_interface_by_ip(protocol.dstip)

        if interface is None:
            self.forward_ip_packet(packet, packet_in, protocol)
        else:
            self.handle_ip_packet_to_router(packet, packet_in, protocol)

    def forward_ip_packet(self, packet, packet_in, protocol):
        log.debug('forward_ip_packet')

        entry = self.routing_table.lookup(protocol.dstip)

        if entry is None:
            self.handle_unreachable(packet, packet_in, protocol)
            return

        match = of.ofp_match.from_packet(packet)
        packet.src = entry.interface.mac
        packet.dst = self.arp_cache.findMACByIP(entry.interface.host)

        if packet.dst is None:
            self.send_arp_request(hwsrc=packet.src, protosrc=entry.interface.ip,
                                  protodst=entry.interface.host,
                                  port=entry.interface.port)
            self.enqueue_msg_for_arp(entry.interface.host, packet, packet_in=packet_in
,
                                     match=match)

            return

        self.send_flow_mod(packet=packet, packet_in=packet_in, port=entry.interface.
port,
                           match=match)

    def handle_ip_packet_to_router(self, packet, packet_in, protocol):
        log.debug('handle_ip_packet_to_router')

        if protocol.protocol == ipv4.ICMP_PROTOCOL:
            self.handle_icmp_to_router(packet, packet_in, protocol.payload)

    def handle_unreachable(self, packet, packet_in, protocol):
        log.debug('handle_unreachable srcip (%s) dstip (%s)' % (protocol.srcip,
                                                                protocol.dstip))
        srcip = self.routing_table.port_to_iface[packet_in.in_port].ip

        self.send_icmp_unreach(packet.dst, packet.src, srcip, protocol.srcip,
                               protocol, packet_in.in_port)

    def handle_icmp_to_router(self, packet, packet_in, protocol):
        if protocol.type == TYPE_ECHO_REQUEST:
            protocol.type = TYPE_ECHO_REPLY
            ip = packet.payload
            packet.src, packet.dst = packet.dst, packet.src
            ip.srcip, ip.dstip = ip.dstip, ip.srcip

            self.send_packet_out(packet, packet_in.in_port)

    def send_icmp_unreach(self, hwsrc, hwdst, srcip, dstip, payload, port):
        protocol = unreach()
        protocol.payload = payload

        self.send_icmp(hwsrc, hwdst, srcip, dstip, TYPE_DEST_UNREACH, port,
                       payload=protocol)

    def send_icmp(self, hwsrc, hwdst, srcip, dstip, opcode, port, payload=None):
        log.debug('send_icmp opcode(%d)' % opcode)

        protocol = icmp()
        protocol.type = opcode
        protocol.payload = payload

```

```python
        self.send_ip(hwsrc, hwdst, srcip, dstip, ipv4.ICMP_PROTOCOL, port,
                     payload=protocol)

    def send_ip(self, hwsrc, hwdst, srcip, dstip, opcode, port, payload=None):
        log.debug('send_ip - srcip(%s) dstip(%s)' % (str(srcip), str(dstip)))

        protocol = ipv4()
        protocol.srcip = srcip
        protocol.dstip = dstip
        protocol.protocol = opcode
        protocol.payload = payload

        self.send_ethernet(hwsrc, hwdst, ethernet.IP_TYPE,
                           port, payload=protocol)

    def send_arp_request(self, hwsrc, protosrc, protodst, port):
        self.send_arp(hwsrc, ETHER_BROADCAST, protosrc, protodst, arp.REQUEST,
                      port)

    def send_arp(self, hwsrc, hwdst, protosrc, protodst, opcode, port):
        log.debug('send_arp hwsrc(%s) hwdst(%s) src(%s) dst(%s) opcode(%d)' %
                  (str(hwsrc), str(hwdst), str(protosrc), str(protodst), opcode))

        protocol = arp()

        protocol.hwsrc = hwsrc
        protocol.hwdst = hwdst
        protocol.protosrc = protosrc
        protocol.protodst = protodst
        protocol.opcode = opcode

        self.send_ethernet(src=hwsrc, dst=hwdst, opcode=ethernet.ARP_TYPE,
                           port=port, payload=protocol)

    def send_ethernet(self, src, dst, opcode, port, payload=None):
        log.debug('send_ethernet src(%s) dst(%s)' % (str(src), str(dst)))

        packet = ethernet()

        packet.src = src
        packet.dst = dst
        packet.type = opcode
        packet.payload = payload

        self.send_packet_out(packet, port)

    def send_packet_out(self, packet, port):
        msg = of.ofp_packet_out()

        msg.actions.append(of.ofp_action_output(port=port))
        msg.data = packet

        self.connection.send(msg)

    def send_flow_mod(self, packet=None, port=None, packet_in=None, match=None):
        """
        """
        log.debug("Installing flow " + str(packet.src) + " --> " +
                  str(packet.dst) + " port " + str(port))
        msg = of.ofp_flow_mod()

        if match is None:
            # Set fields to match outgoing packet
            msg.match = of.ofp_match.from_packet(packet)
        else:
            msg.match = match

        # Set other fields of flow_mod: timeouts, buffer_id
        msg.idle_timeout = 0
        msg.hard_timeout = 0

        if packet_in is not None:
            msg.data = packet_in

        msg.actions.append(of.ofp_action_dl_addr.set_src(packet.src))
        msg.actions.append(of.ofp_action_dl_addr.set_dst(packet.dst))
        msg.actions.append(of.ofp_action_output(port=port))

        self.connection.send(msg)

    def send_packet_and_install_flow(self, packet, port, match=None):
        self.send_flow_mod(packet, port, match)
        self.send_packet_out(packet, port)

    def _handle_PacketIn(self, event):
        """
        Handles packet in messages from the switch.
        """

        packet = event.parsed  # This is the parsed packet data.
        if not packet.parsed:
            log.warning("Ignoring incomplete packet")
            return

        packet_in = event.ofp  # The actual ofp_packet_in message.

        # self.act_like_hub(packet, packet_in)
        # self.act_like_switch(packet, packet_in)
        self.act_like_router(packet, packet_in)


def launch():
    """
    Starts the component
    """
    def start_switch(event):
        log.debug("Controlling %s" % (event.connection,))

        packet_in = event.ofp
        ports = packet_in.ports
        switch = None

        for port in ports:

            print port
            if port.name.find('-') >= 0:
                switch = port.name.split('-')[0]

                switch_to_table[switch].port_to_iface[port.port_no].mac = port.hw_addr

        if switch is not None:
            Router(event.connection, switch_to_table[switch])

    core.openflow.addListenerByName("ConnectionUp", start_switch)
```

Appendix B Communication

B.1 Messages

Base class for messages exchanged between client and server.

```java
package org.matheus.runtime.message;

import java.net.InetAddress;

import org.matheus.infrastructure.Identifier;

import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.annotations.XStreamAlias;
import com.thoughtworks.xstream.annotations.XStreamOmitField;

@XStreamAlias("Message")
public abstract class Message implements java.io.Serializable {
```

```
/**
 *
 */
private static final long serialVersionUID = -6624233850795123800L;

public final static Integer BROADCAST_IP_ADDRESS = 0xffffffff;
@XStreamOmitField
/** Connection identifier **/
private transient Identifier connectionId;
@XStreamOmitField
/** IP source **/
private InetAddress source;
@XStreamOmitField
/** Source TCP/IP port **/
private int srcport;
@XStreamOmitField
/** IP destination **/
private InetAddress destination;
@XStreamOmitField
/** Destination TCP/IP port **/
private int dstport;
@XStreamOmitField
/**
 * Message ID. If the message is a reply, it should have the same ID of the
 * request
 **/
private int id;
@XStreamOmitField
/** Message modifiers **/
private int flag;
@XStreamOmitField
/** Message type **/
private MessageType msgType;

public enum MessageType {
  ACTION_MESSAGE, HELLO, INVOKE_RESULT, REGISTER_OBJECT_RESPONSE, ACTION_RESPONSE,
  ALLOC_HEAP_REQUEST, ALLOC_HEAP_REPLY, OBJECTS_REQUEST, OBJECTS_REPLY, NOTIFICATION
  , ERROR, CLASSES_FIELDS_REQUEST, CLASSES_FIELDS_REPLY, REACHABILITY_REQUEST,
  REACHABILITY_REPLY, REMOVE_REFERENCE, MONITOR_ENTER, MONITOR_EXIT
}

public Message(MessageType type) {
  this.msgType = type;
}

public void setConnectionId(Identifier connectionId) {
  this.connectionId = connectionId;
}
```

```java

public Identifier getConnectionId() {
  return connectionId;
}

public void setSrcip(InetAddress source) {
  this.source = source;
}

public InetAddress getSrcip() {
  return source;
}

public void setDstip(InetAddress destination) {
  this.destination = destination;
}

public InetAddress getDstip() {
  return destination;
}

public int getDstport() {
  return dstport;
}

public void setDstport(int dstport) {
  this.dstport = dstport;
}

public int getSrcport() {
  return srcport;
}

public void setSrcport(int srcport) {
  this.srcport = srcport;
}

public void setId(int id) {
  this.id = id;
}

public int getId() {
  return id;
}

public void set(int flag) {
  this.flag |= flag;
}
```

```

  public void unset(int flag) {
    this.flag &= ~flag;
  }

  public boolean isSet(int flag) {
    return (this.flag & flag) != 0;
  }

  public MessageType getMsgType() {
    return msgType;
  }

  public class Modifier {
    public final static int REPLY = (1 << 0);
  }

  public void handle() {

  }

  @Override
  public String toString() {
    MyMessageDumper xstream = new MyMessageDumper();

    xstream.autodetectAnnotations(true);
    return xstream.toXML(this);
  }

  class MyMessageDumper extends XStream {

    @Override
    public String toXML(Object obj) {
      alias(obj.getClass().getSimpleName(), obj.getClass());

      return super.toXML(obj);
    }

    public void omitField(Class<?> clazz, String... names) {
      for (String name : names) {
        omitField(clazz, name);
      }
    }
  }

}
```

B.2 Connection Handler

Connection handlers implementation responsible by sending and processing incoming messages.

```java
package org.matheus.infrastructure;

import java.io.IOException;
import java.io.ObjectOutputStream;
import java.net.InetAddress;
import java.net.Socket;
import java.util.HashMap;

import org.matheus.infrastructure.stream.ConnectionPipe;
import org.matheus.infrastructure.stream.MessagePipe;
import org.matheus.infrastructure.stream.MessagePipeBase;
import org.matheus.infrastructure.stream.MessageSink;
import org.matheus.runtime.message.Message;
import org.matheus.runtime.util.Logger;
import org.matheus.runtime.util.Util;

public abstract class ConnectionHandler implements MessageSink, Runnable {
  private static final String TAG = ConnectionHandler.class.getSimpleName();
  /**
   * Monotonic increasing counter. The initial value should be random for
   * security reasons.
   **/
  private static int msgId = (int) (Math.random() * (double) Integer.MAX_VALUE);

  /** Connection ID **/
  protected Identifier id;
  /**
   * How long to wait for a blocking call. Default value is 0 which waits
   * forever
   **/
  protected int timeout = 0;
  protected Socket socket;
  protected ObjectOutputStream outStream;
  /** Stores messages waiting for replies **/
  protected HashMap<Integer, PacketMailbox> mailboxes = new HashMap<>();
  /** Pipe for incoming messages **/
  protected MessagePipe msgPipeIn = new MessagePipeBase();
  /** Pipe for connection events (e.g. fail) **/
  protected ConnectionPipe connectionPipe;

  public ConnectionHandler(Identifier id, Socket socket) throws IOException {
    this.id = id;
    this.socket = socket;
    this.outStream = new ObjectOutputStream(socket.getOutputStream());
```

```java
}

public InetAddress getIP() {
  return socket.getInetAddress();
}

public int getPort() {
  return socket.getPort();
}

public MessagePipe getMsgPipeIn() {
  return msgPipeIn;
}

public void setTimeout(int timeout) {
  this.timeout = timeout;
}

@Override
public void run() {
  Logger.d(TAG, "run");

  readPackets();
  close();
  Logger.d(TAG, "lost connection with " + " (" + socket.getInetAddress().
  getHostAddress() + ")");

  if (connectionPipe != null)
    connectionPipe.removeConnection(id);
}

protected abstract void readPackets();

public void setConnectionPipe(ConnectionPipe connectionPipe) {
  this.connectionPipe = connectionPipe;
}

protected synchronized void handlePacketIn(Message msg) {
  if (Util.DEBUG) {
    Logger.f(TAG, "received:\n" + msg + "\n");
    Logger.d(TAG, "received: " + msg.getClass().getSimpleName());
  }

  msg.setConnectionId(id);

  if (msg.isSet(Message.Modifier.REPLY)) {
    PacketMailbox mailbox = mailboxes.remove(msg.getId());

```

```java
      synchronized (mailbox) {
        mailbox.reply = msg;
        mailbox.notify();
      }
    } else {
      msgPipeIn.sendMessage(id, msg);
    }
  }

  /**
   * Asynchronous message.
   *
   * @param msg
   */
  public synchronized void sendAsync(Message msg) {
    if (Util.DEBUG) {
      Logger.f(TAG, "sent:\n" + msg + "\n");
      Logger.d(TAG, "sent: " + msg.getClass().getSimpleName());
    }

    try {
      outStream.writeObject(msg);
      outStream.reset();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  /**
   * Blocking message.
   *
   * @param msg
   * @return
   */
  public Object send(Message msg) {
    PacketMailbox mailbox = new PacketMailbox(msg);

    synchronized (this) {
      msgId = (msgId + 1) % Integer.MAX_VALUE;
      mailboxes.put(msgId, mailbox);
      msg.setId(msgId);
    }

    synchronized (mailbox) {
      sendAsync(msg);

      try {
        mailbox.wait(timeout);
        return mailbox.reply;
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
    }

    return null;
  }

  public synchronized void close() {
    try {
      socket.close();
    } catch (IOException e) {
      e.printStackTrace();
    }
  }

  @Override
  public void messageSent(Identifier id, Message message) {
    sendAsync(message);
  }

  class PacketMailbox {
    Message request;
    Message reply = null;

    PacketMailbox(Message request) {
      this.request = request;
    }
  }
}
```

Appendix C Remote procedure call

This section presents the transformations illustrated in figure 11. The class at C.1 copies the original method bytecode to another method of same name with the prefix orig$ in the beginning. In addition, it creates a bridge method for type conversion when the call is received in the server. The transformation at C.2 replaces the method body by instructions that generate the runtime invocation asking for a remote execution. Finally, in C.3, the method visitor fills the bridge method's bytecode.

C.1 Method Generators

```java
package org.matheus.runtime.instrumenter.visitor;

```

```java
import org.matheus.runtime.instrumenter.metadata.ClassMetadata;
import org.matheus.runtime.instrumenter.metadata.ClassRepository;
import org.matheus.runtime.instrumenter.metadata.MethodMetadata;
import org.matheus.runtime.util.Logger;
import org.matheus.runtime.util.Remoteable;
import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;
import org.objectweb.asm.Type;

public class RemoteableMethodReplacerVisitor extends ClassVisitor {
  private static final String TAG = RemoteableMethodReplacerVisitor.class.
    getSimpleName();
  private ClassRepository repository;
  private ClassMetadata clazz;
  private String className;

  public RemoteableMethodReplacerVisitor(ClassVisitor cv, ClassRepository repository)
    {
    super(Opcodes.ASM4, cv);

    this.repository = repository;
  }

  @Override
  public void visit(int version, int access, String name, String signature, String
    superName, String[] interfaces) {
    this.clazz = repository.getClass(name);
    this.className = name;
    super.visit(version, access, name, signature, superName, interfaces);
  }

  @Override
  public MethodVisitor visitMethod(int access, String name, String desc, String
    signature, String[] exceptions) {
    MethodMetadata method = clazz.getMethod(name, desc);

    if (method == null) {
      Logger.e(TAG, "Method not found " + clazz.getName() + "." + name);
    }

    if (method.getAnnotations().contains(Type.getDescriptor(Remoteable.class))) {
      MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
      String bridgeName = "bridge$" + name;
      String bridgeDesc = BridgeMethodVisitor.replacePrimitiveDesc(desc);
      String origName = "orig$" + name;
      String origDesc = desc;

      mv = super.visitMethod(access, bridgeName, bridgeDesc, signature, exceptions);

      if (mv != null) {
        mv = new BridgeMethodVisitor(mv, access, className, bridgeName, bridgeDesc,
    origName, desc);
        mv.visitCode();
        mv.visitMaxs(0, 0);
        mv.visitEnd();
      }

      mv = super.visitMethod(access, origName, origDesc, signature, exceptions);
      return new RemoveAnnotationMethodVisitor(mv, Type.getDescriptor(Remoteable.class
    ));
    } else {
      return super.visitMethod(access, name, desc, signature, exceptions);
    }
  }
}
```

C.2 Runtime Call

```java
package org.matheus.runtime.instrumenter.visitor;

import static org.objectweb.asm.Opcodes.*;

import org.matheus.runtime.instrumenter.metadata.ClassMetadata;
import org.matheus.runtime.instrumenter.metadata.ClassRepository;
import org.matheus.runtime.instrumenter.metadata.MethodMetadata;
import org.matheus.runtime.util.Logger;
import org.matheus.runtime.util.Remoteable;

import static org.matheus.runtime.SymbolTable.*;

import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Type;

public class RemoteableMethodWrapperVisitor extends ClassVisitor {
  private static final String TAG = RemoteableMethodWrapperVisitor.class.getSimpleName
    ();
  private String className;
  private ClassRepository repository;
  private ClassMetadata clazz;

  public RemoteableMethodWrapperVisitor(ClassVisitor cv, ClassRepository repository) {
    super(ASM5, cv);

    this.repository = repository;
```

```java
}

@Override
public void visit(int version, int access, String name, String signature, String
  superName, String[] interfaces) {
  super.visit(version, access, name, signature, superName, interfaces);

  clazz = repository.getClass(name);
  className = name;
}

@Override
public MethodVisitor visitMethod(int access, String name, String desc, String
  signature, String[] exceptions) {
  MethodMetadata method = clazz.getMethod(name, desc);

  if (method == null) {
    Logger.e(TAG, "Method not found " + className + "." + name);
  }

  if (clazz.getMethod(name, desc).getAnnotations().contains(Type.getDescriptor(
  Remoteable.class))) {
    generateWrapperCode(access, name, desc, signature, exceptions);

    return null;
  }

  return super.visitMethod(access, name, desc, signature, exceptions);
}

private MethodVisitor generateWrapperCode(int access, String name, String desc,
  String signature,
    String[] exceptions) {
  MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);
  Type types[] = Type.getArgumentTypes(desc);
  Type returnType = Type.getReturnType(desc);
  boolean isStatic = org.matheus.instrumenter.Util.hasAccess(access, ACC_STATIC);
  int paramIndex = 0;
  int maxLocals = types.length;
  Type classType = Type.getObjectType(className);

  if (mv == null)
    return null;

  mv.visitCode();

  if (isStatic) {
    mv.visitInsn(ACONST_NULL);
  } else {
    mv.visitVarInsn(ALOAD, 0); // this
    maxLocals++;
    paramIndex++;
  }

  // Method class
  mv.visitLdcInsn(classType);

  // Method name
  mv.visitLdcInsn("bridge$" + name);

  // Method descriptor
  mv.visitLdcInsn(BridgeMethodVisitor.replacePrimitiveDesc(desc));

  // Parameters class array
  mv.visitIntInsn(BIPUSH, types.length);
  mv.visitTypeInsn(ANEWARRAY, "java/lang/Class");

  for (int i = 0; i < types.length; ++i) {
    mv.visitInsn(DUP);
    mv.visitIntInsn(BIPUSH, i);
    visitTypeClass(mv, types[i]);
    mv.visitInsn(AASTORE);
  }

  // Parameters array
  mv.visitIntInsn(BIPUSH, types.length);
  mv.visitTypeInsn(ANEWARRAY, "java/lang/Object");

  for (int i = 0; i < types.length; ++i) {
    mv.visitInsn(DUP);
    mv.visitIntInsn(BIPUSH, i);
    mv.visitVarInsn(types[i].getOpcode(ILOAD), paramIndex + i);
    primitive2Object(mv, types[i].getSort());
    mv.visitInsn(AASTORE);
  }

  mv.visitMethodInsn(INVOKESTATIC, RUNTIME_NAME, INVOKE_METHOD, INVOKE_METHOD_DESC,
  false);
  object2Primitive(mv, returnType);
  mv.visitInsn(Type.getReturnType(desc).getOpcode(IRETURN));
  mv.visitMaxs(10, maxLocals);
  mv.visitEnd();

  return mv;
}

```

```java
public final static void visitTypeClass(MethodVisitor mv, Type type) {
  switch (type.getSort()) {
  case Type.BOOLEAN:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Boolean", "TYPE", "Ljava/lang/Class;");
    break;
  case Type.BYTE:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Byte", "TYPE", "Ljava/lang/Class;");
    break;
  case Type.CHAR:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Character", "TYPE", "Ljava/lang/Class;")
  ;
    break;
  case Type.DOUBLE:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Double", "TYPE", "Ljava/lang/Class;");
    break;
  case Type.FLOAT:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Float", "TYPE", "Ljava/lang/Class;");
    break;
  case Type.INT:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Integer", "TYPE", "Ljava/lang/Class;");
    break;
  case Type.LONG:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Long", "TYPE", "Ljava/lang/Class;");
    break;
  case Type.SHORT:
    mv.visitFieldInsn(GETSTATIC, "java/lang/Short", "TYPE", "Ljava/lang/Class;");
    break;
  default:
    mv.visitLdcInsn(type);
  }
}

public final static void primitive2Object(MethodVisitor mv, int type) {
  switch (type) {
  case Type.BOOLEAN:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Boolean", "valueOf", "(Z)Ljava/lang/
  Boolean;", false);
    break;
  case Type.BYTE:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Byte", "valueOf", "(B)Ljava/lang/
  Byte;", false);
    break;
  case Type.CHAR:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Character", "valueOf", "(C)Ljava/
  lang/Character;", false);
    break;
  case Type.DOUBLE:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Double", "valueOf", "(D)Ljava/lang/
  Double;", false);
    break;
  case Type.FLOAT:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Float", "valueOf", "(F)Ljava/lang/Float;", false);
    break;
  case Type.INT:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Integer", "valueOf", "(I)Ljava/lang/Integer;", false);
    break;
  case Type.LONG:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Long", "valueOf", "(J)Ljava/lang/Long;", false);
    break;
  case Type.METHOD:
    break;
  case Type.SHORT:
    mv.visitMethodInsn(INVOKESTATIC, "java/lang/Short", "valueOf", "(S)Ljava/lang/Short;", false);
    break;
  case Type.VOID:
  case Type.OBJECT:
    break;
  }
}

public final static void object2Primitive(MethodVisitor mv, Type type) {
  switch (type.getSort()) {
  case Type.BOOLEAN:
    mv.visitTypeInsn(CHECKCAST, "java/lang/Boolean");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Boolean", "booleanValue", "()Z", false);
    break;
  case Type.BYTE:
    mv.visitTypeInsn(CHECKCAST, "java/lang/Byte");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Byte", "byteValue", "()B", false);
    break;
  case Type.CHAR:
    mv.visitTypeInsn(CHECKCAST, "java/lang/Character");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Character", "characterValue", "()C", false);
    break;
  case Type.DOUBLE:
    mv.visitTypeInsn(CHECKCAST, "java/lang/Double");
    mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Double", "doubleValue", "()D", false);
    break;
  case Type.FLOAT:
```

```java
      mv.visitTypeInsn(CHECKCAST, "java/lang/Float");
      mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Float", "floatValue", "()F", false)
    ;
      break;
    case Type.INT:
      mv.visitTypeInsn(CHECKCAST, "java/lang/Integer");
      mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Integer", "intValue", "()I", false)
    ;
      break;
    case Type.LONG:
      mv.visitTypeInsn(CHECKCAST, "java/lang/Long");
      mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Long", "longValue", "()J", false);
      break;
    case Type.METHOD:
      mv.visitTypeInsn(CHECKCAST, "java/lang/reflect/Method");
      break;
    case Type.SHORT:
      mv.visitTypeInsn(CHECKCAST, "java/lang/Short");
      mv.visitMethodInsn(INVOKEVIRTUAL, "java/lang/Short", "shortValue", "()S", false)
    ;
      break;
    case Type.VOID:
      break;
    case Type.OBJECT:
      mv.visitTypeInsn(CHECKCAST, type.getInternalName());
      break;
    }
  }
}
```

C.3 Bridge Method

```java
package org.matheus.runtime.instrumenter.visitor;

import static org.objectweb.asm.Opcodes.*;

import java.lang.reflect.Modifier;

import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Opcodes;
import org.objectweb.asm.Type;

public class BridgeMethodVisitor extends MethodVisitor {
  private int acc;
  private String className;
  @SuppressWarnings("unused")
  private String name;
  private String desc;
  private String invokeName;
  private String invokeDesc;
  private int locals;

  public BridgeMethodVisitor(MethodVisitor mv, int acc, String className, String name,
     String desc, String invokeName,
      String invokeDesc) {
    super(ASM5, mv);

    this.acc = acc;
    this.className = className;
    this.name = name;
    this.desc = desc;
    this.invokeName = invokeName;
    this.invokeDesc = invokeDesc;
  }

  @Override
  public void visitCode() {
    super.visitCode();
    Type[] types = Type.getArgumentTypes(desc);
    Type[] invokeTypes = Type.getArgumentTypes(invokeDesc);
    int invokeOpcode;
    int param = 0;

    if (Modifier.isStatic(acc)) {
      invokeOpcode = INVOKESTATIC;
    } else {
      param++;
      locals++;
      visitVarInsn(ALOAD, 0);
      invokeOpcode = INVOKESPECIAL;
    }

    for (int i = 0; i < types.length; ++i) {
      visitVarInsn(ALOAD, param);
      param += types[i].getSize();
      locals += types[i].getSize();

      if (types[i].equals(invokeTypes[i]))
        continue;

      RemoteableMethodWrapperVisitor.object2Primitive(this, invokeTypes[i]);
    }

    visitMethodInsn(invokeOpcode, className, invokeName, invokeDesc, false);
    visitInsn(Type.getReturnType(invokeDesc).getOpcode(Opcodes.IRETURN));
  }
```

```

@Override
public void visitMaxs(int maxStack, int maxLocals) {
  super.visitMaxs(maxStack + locals, maxStack + locals);
}

public final static Type primitiveToObject(Type type) {
  switch (type.getSort()) {
  case Type.BOOLEAN:
    return Type.getType(Boolean.class);
  case Type.BYTE:
    return Type.getType(Byte.class);
  case Type.CHAR:
    return Type.getType(Character.class);
  case Type.DOUBLE:
    return Type.getType(Double.class);
  case Type.FLOAT:
    return Type.getType(Float.class);
  case Type.INT:
    return Type.getType(Integer.class);
  case Type.LONG:
    return Type.getType(Long.class);
  case Type.SHORT:
    return Type.getType(Short.class);
  default:
    return type;
  }
}

public static final String replacePrimitiveDesc(String desc) {
  Type[] argType = Type.getArgumentTypes(desc);
  Type[] newType = new Type[argType.length];
  Type returnType = Type.getReturnType(desc);
  int i = 0;

  for (Type type : argType) {
    newType[i++] = primitiveToObject(type);
  }

  desc = "(";

  for (Type type : newType) {
    desc += type.getDescriptor();
  }

  desc += ")" + returnType.getDescriptor();

  return desc;
  }
}
```

Appendix D Remoteable Object Interface

In this section we present the source code for major interfaces of our runtime objects as follows: the `RemoteableObject` is a common interface for all objects including arrays; `RemoteableObjectBase` is an interface exclusive for non-array objects; `RemoteableArray` is the class where arrays are stored; and finally, `RemoteableObjectBaseImplementation` has the implementation of the `RemoteableObjectBase`. The bytecode compiled from `RemoteableObjectBaseImplementation` is copied to instrumented classes.

D.1 Interface for all objects

```java
package org.matheus.runtime;

import static org.matheus.runtime.Modifier.*;

import java.io.Serializable;

public interface RemoteableObject extends Serializable {

  public default void org$matheus$runtime$init() {
    if (!RemoteableRuntime.accept() || Modifier.isInitialized(this)) {
      return;
    }
    RemoteableRuntime.currentThread().setRuntimeMode(true);

    org$matheus$runtime$init(RemoteableRuntime.runtime.memoryManager.getHeap().alloc()
    , false);

    RemoteableRuntime.currentThread().setRuntimeMode(false);
  }

  /**
   * Should be called only with runtime privilege
   */
  public default void org$matheus$runtime$initIfNeeded() {
    if (Modifier.isInitialized(this))
      return;

    org$matheus$runtime$init(RemoteableRuntime.runtime.memoryManager.getHeap().alloc()
    , false);
  }

  public default void org$matheus$runtime$init(int globalRef, boolean loading) {
    org$matheus$runtime$setId(globalRef);
    Modifier.set(this, INITIALIZED);

    if (loading) {
      RemoteableRuntime.runtime.objectManager.addLoadedObject(this);
    } else {
      RemoteableRuntime.runtime.objectManager.addObject(this);
    }
  }

  public int org$matheus$runtime$getModifier();

  public void org$matheus$runtime$setModifier(int mod);

  public int org$matheus$runtime$getId();

  public void org$matheus$runtime$setId(int id);

  public RemoteableReference org$matheus$runtime$getRef();

  public SerializedObject org$matheus$runtime$serialize();
}
```

D.2 Interface for non-array objects

```java
package org.matheus.runtime;

public interface RemoteableObjectBase extends RemoteableObject {

  @Override
  public default SerializedObject org$matheus$runtime$serialize() {
    return new SerializedObjectBaseDebug(this);
  }

  public long[] org$matheus$runtime$getDirtyBitset();

  public void org$matheus$runtime$setObjectDirty(int fieldIndex);

  public void org$matheus$runtime$setDirtyBitset(long[] bitset);

}
```

D.3 Class for array storage

```java
package org.matheus.runtime;

import static org.matheus.runtime.Modifier.*;

public class RemoteableArray implements RemoteableObject {
```

```java
/**
 *
 */
private static final long serialVersionUID = 6407989760483039164L;
protected int org$matheus$runtime$id;
protected int org$matheus$runtime$mod;
protected transient RemoteableReference org$matheus$runtime$ref;
protected Object ref;

public RemoteableArray(Object ref) {
  this.ref = ref;

  // Allocate heap identifier manually because the RemoteableObject
  // constructor will be blocked in privileged mode
  org$matheus$runtime$init(RemoteableRuntime.runtime.memoryManager.getHeap().alloc()
  , false);
}

protected RemoteableArray(Object ref, int globalRef) {
  org$matheus$runtime$init(globalRef, true);
  this.ref = ref;
}

public Object getArray() {
  return ref;
}

@Override
public SerializedObject org$matheus$runtime$serialize() {
  return new SerializedArray(this);
}

@Override
public final int org$matheus$runtime$getModifier() {
  return org$matheus$runtime$mod;
}

@Override
public final void org$matheus$runtime$setModifier(int mod) {
  org$matheus$runtime$mod = mod;
}

@Override
public final int org$matheus$runtime$getId() {
  return org$matheus$runtime$id;
}

@Override
```

```java
  public final void org$matheus$runtime$setId(int id) {
    org$matheus$runtime$id = id;
  }

  @Override
  public final RemoteableReference org$matheus$runtime$getRef() {
    if (org$matheus$runtime$ref == null) {
      org$matheus$runtime$ref = new RemoteableReference(this);
    }
    return org$matheus$runtime$ref;
  }

  public static void updateArray(SerializedArray array) {
    RemoteableArray dst = (RemoteableArray) RemoteableRuntime.runtime.objectManager.
    getObject(array.getGlobalRef());

    array.copyTo(dst);
  }

  public final void org$matheus$runtime$setObjectDirty() {
    if (Modifier.isDirty(this))
      return;

    RemoteableRuntime.runtime.memoryManager.addDirtyArray(this);
    Modifier.set(this, DIRTY);
  }

}
```

D.4 Implementation of RemoteableObjectBase

```java
package org.matheus.runtime;

@SuppressWarnings("serial")
public class RemoteableObjectBaseImplementation implements RemoteableObjectBase {

  protected int org$matheus$runtime$id;
  protected int org$matheus$runtime$mod;
  protected transient RemoteableReference org$matheus$runtime$ref;

  @Override
  public final int org$matheus$runtime$getModifier() {
    return org$matheus$runtime$mod;
  }

  @Override
  public final void org$matheus$runtime$setModifier(int mod) {
    org$matheus$runtime$mod = mod;
  }

  @Override
  public final int org$matheus$runtime$getId() {
    return org$matheus$runtime$id;
  }

  @Override
  public final void org$matheus$runtime$setId(int id) {
    org$matheus$runtime$id = id;
  }

  @Override
  public final RemoteableReference org$matheus$runtime$getRef() {
    if (org$matheus$runtime$ref == null) {
      org$matheus$runtime$ref = new RemoteableReference(this);
    }
    return org$matheus$runtime$ref;
  }

  /** RemoteableObjectBase Implementation **/

  public long[] org$matheus$runtime$dirtyBitset;

  @Override
  public final SerializedObject org$matheus$runtime$serialize() {
    return new SerializedObjectBaseDebug(this);
  }

  @Override
  public final void org$matheus$runtime$setObjectDirty(int fieldIndex) {
    org$matheus$runtime$dirtyBitset[fieldIndex / 64] |= (1 << (fieldIndex % 64));

    if ((org$matheus$runtime$dirtyBitset[0] & 0x1) == 0) {
      org$matheus$runtime$dirtyBitset[0] |= 0x1;
      RemoteableRuntime.getRuntime().getMemoryManager().addDirtyObject(this);
    }
  }

  @Override
  public final long[] org$matheus$runtime$getDirtyBitset() {
    return org$matheus$runtime$dirtyBitset;
  }

  @Override
  public final void org$matheus$runtime$setDirtyBitset(long[] bitset) {
    org$matheus$runtime$dirtyBitset = bitset;
  }
```

```

}
```

Appendix E Dirty field tracking

The code bellow generate bytecode transformations to track field writes.

```java
package org.matheus.runtime.instrumenter.visitor;

import org.matheus.runtime.instrumenter.InstrConfiguration;
import org.matheus.runtime.instrumenter.metadata.ClassMetadata;
import org.matheus.runtime.instrumenter.metadata.ClassRepository;
import org.matheus.runtime.instrumenter.metadata.FieldMetadata;
import org.matheus.runtime.util.Logger;
import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.Label;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Type;
import org.objectweb.asm.commons.AdviceAdapter;

import static org.matheus.runtime.SymbolTable.*;
import static org.objectweb.asm.Opcodes.*;

public class PutFieldVisitor extends ClassVisitor {
  private static final String TAG = PutFieldVisitor.class.getSimpleName();
  ClassRepository repository;
  ClassMetadata classData;
  String className;
  String methodName;

  public PutFieldVisitor(ClassVisitor cv, ClassRepository program) {
    super(ASM5, cv);

    this.repository = program;
  }

  @Override
  public void visit(int version, int access, String name, String signature, String
    superName, String[] interfaces) {
    classData = repository.getClass(name);
    this.className = name;
    super.visit(version, access, name, signature, superName, interfaces);
  }

  @Override
  public MethodVisitor visitMethod(int access, String name, String desc, String
    signature, String[] exceptions) {
    MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);

    if (mv == null)
      return null;

    mv = new PutfieldMethodVisitor(mv, access, name, desc);

    this.methodName = name;

    return mv;
  }

  class PutfieldMethodVisitor extends AdviceAdapter {
    boolean constructor;
    boolean initialized = false;
    boolean hasLocals = false;
    int reference;
    int value;

    public PutfieldMethodVisitor(MethodVisitor mv, int access, String name, String
    desc) {
      super(ASM5, mv, access, name, desc);

      if (name.equals("<init>"))
        constructor = true;
    }

    @Override
    public void visitCode() {
      super.visitCode();
    }

    @Override
    public void visitMethodInsn(int opcode, String owner, String name, String desc) {
      if (opcode == INVOKESPECIAL && name.equals("<init>"))
        initialized = true;
    }

    @Override
    public void visitFieldInsn(int opcode, String owner, String name, String desc) {
      ClassMetadata classData = repository.getClass(owner);
      FieldMetadata fieldData;
      int pos;

      if (constructor && !initialized) {
        super.visitFieldInsn(opcode, owner, name, desc);
        return;
      }

      if (classData == null) {
```

```
      Logger.e(TAG, "class not found - " + owner);
    }

    if (InstrConfiguration.getInstance().shouldNotInstrument(owner)) {
      super.visitFieldInsn(opcode, owner, name, desc);
      return;
    }

    switch (opcode) {
    case PUTFIELD:
      fieldData = classData.getField(name);

      if (!hasLocals) {
        hasLocals = true;
        reference = newLocal(Type.getType(Object.class));
        value = newLocal(Type.getType(double.class));
      }

      if (fieldData == null) {
        Logger.e(TAG, "visitFieldInsn " + className + "." + methodName + " field not found - " + owner + "."
            + name);
      }

      pos = fieldData.id + 1;

      Type type = Type.getType(desc);
      if (type.getSize() == 2) {
        mv.visitInsn(DUP2_X1);
        mv.visitVarInsn(type.getOpcode(ISTORE), value);
        mv.visitInsn(DUP_X2);
        mv.visitVarInsn(ASTORE, reference);
      } else {
        mv.visitInsn(DUP_X1);
        mv.visitVarInsn(type.getOpcode(ISTORE), value);
        mv.visitInsn(DUP_X1);
        mv.visitVarInsn(ASTORE, reference);
      }

      super.visitFieldInsn(opcode, owner, name, desc);

      int index = pos / 64;
      Label l0 = new Label();

      // if((obj_dirty_table[index] & (1 << pos % 64)) == 0)
      mv.visitVarInsn(ALOAD, reference);
      mv.visitFieldInsn(GETFIELD, owner, OBJECT_DIRTY_BITSET, "[J");
      mv.visitLdcInsn(index);
```

```
        mv.visitInsn(LALOAD);
        mv.visitLdcInsn((long) (1 << (pos % 64)));
        mv.visitInsn(LAND);
        mv.visitLdcInsn((long) 0);
        mv.visitInsn(LCMP);
        mv.visitJumpInsn(IFNE, l0);

        // If body
        mv.visitVarInsn(ALOAD, reference);
        mv.visitInsn(DUP);
        mv.visitFieldInsn(GETFIELD, owner, OBJECT_DIRTY_BITSET, "[J");

        if (isReference(type)) {
          mv.visitVarInsn(type.getOpcode(ILOAD), value);
        } else {
          mv.visitInsn(ACONST_NULL);
        }

        mv.visitLdcInsn(pos);
        mv.visitMethodInsn(INVOKESTATIC, RUNTIME_NAME, INVOKE_PUTFIELD,
    INVOKE_PUTFIELD_DESC, false);
        mv.visitLabel(l0);

        break;
      case PUTSTATIC:

        if (!hasLocals) {
          hasLocals = true;
          value = newLocal(Type.getType(double.class));
        }

        type = Type.getType(desc);

        if (type.getSize() == 2) {
          mv.visitInsn(DUP2);
        } else {
          mv.visitInsn(DUP);
        }

        mv.visitVarInsn(type.getOpcode(ISTORE), value);
        super.visitFieldInsn(opcode, owner, name, desc);

        fieldData = classData.getStaticField(name);

        if (fieldData == null) {
          Logger.e(TAG, "visitFieldInsn at " + className + "." + methodName + " field
    not found - " + owner
              + "." + name);
        }

        pos = fieldData.id + 1;

        index = pos / 64;

        Label l1 = new Label();
        // if((class_dirty_table[index] & (1 << pos % 64)) == 0)
        mv.visitFieldInsn(GETSTATIC, owner, CLASS_DIRTY_BITSET, "[J");
        mv.visitLdcInsn(index);
        mv.visitInsn(LALOAD);
        mv.visitLdcInsn((long) (1 << (pos % 64)));
        mv.visitInsn(LAND);
        mv.visitLdcInsn((long) 0);
        mv.visitInsn(LCMP);
        mv.visitJumpInsn(IFNE, l1);

        // If body
        mv.visitLdcInsn(Type.getType("L" + owner + ";"));
        mv.visitFieldInsn(GETSTATIC, owner, CLASS_DIRTY_BITSET, "[J");

        if (isReference(type)) {
          mv.visitVarInsn(type.getOpcode(ILOAD), value);
        } else {
          mv.visitInsn(ACONST_NULL);
        }

        mv.visitLdcInsn(pos);
        mv.visitMethodInsn(INVOKESTATIC, RUNTIME_NAME, "putStatic", "(Ljava/lang/Class
    ;[JLjava/lang/Object;I)V",
              false);
        mv.visitLabel(l1);

        break;
      default:
        super.visitFieldInsn(opcode, owner, name, desc);
      }
    }

    @Override
    public void visitMaxs(int maxStack, int maxLocals) {
      super.visitMaxs(maxStack + 10, maxLocals);
    }

    private boolean isReference(Type type) {
      return type.getSort() == Type.OBJECT || type.getSort() == Type.METHOD || type.
    getSort() == Type.ARRAY;
    }
    }
    }
```

Appendix F Dirty array tracking

This code bellow generate bytecode transformations to track array writes.

```
package org.matheus.runtime.instrumenter.visitor;

import static org.objectweb.asm.Opcodes.*;

import org.matheus.runtime.instrumenter.metadata.ClassMetadata;
import org.matheus.runtime.instrumenter.metadata.ClassRepository;
import org.matheus.runtime.instrumenter.metadata.MethodMetadata;
import org.matheus.runtime.util.Logger;

import static org.matheus.runtime.SymbolTable.*;

import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Type;
import org.objectweb.asm.commons.AdviceAdapter;

public class ArrayStoreVisitor extends ClassVisitor {
  private static final String TAG = ArrayStoreVisitor.class.getSimpleName();
  private ClassRepository repository;
  private ClassMetadata clazz;
  private String className;

  public ArrayStoreVisitor(ClassVisitor cv, ClassRepository repository) {
    super(ASM5, cv);

    this.repository = repository;
  }

  @Override
  public void visit(int version, int access, String name, String signature, String
    superName, String[] interfaces) {
    super.visit(version, access, name, signature, superName, interfaces);

    className = name;
    clazz = repository.getClass(name);
  }

  @Override
  public MethodVisitor visitMethod(int access, String name, String desc, String
    signature, String[] exceptions) {
```

```java
    MethodVisitor mv = super.visitMethod(access, name, desc, signature, exceptions);

    return new ArrayStoreMethodVisitor(mv, access, name, desc);
  }

  class ArrayStoreMethodVisitor extends AdviceAdapter {
    private boolean hasLocals = false;
    private int refIndex;
    private int indexIndex;
    private int valueIndex;
    private MethodMetadata method;

    ArrayStoreMethodVisitor(MethodVisitor mv, int access, String name, String desc) {
      super(ASM5, mv, access, name, desc);

      method = clazz.getMethod(name, desc);

      if(method == null){
        Logger.d(TAG, "Method not found " + className + "." + name);
      }
    }

    @Override
    protected void onMethodEnter() {
      super.onMethodEnter();
    }

    @Override
    public void visitInsn(int opcode) {
      Type valueType = getType(opcode);

      switch (opcode) {
      case DASTORE:
      case LASTORE:
      case AASTORE:
      case BASTORE:
      case CASTORE:
      case FASTORE:
      case IASTORE:
      case SASTORE:

        if (!hasLocals) {
          hasLocals = true;
          refIndex = newLocal(Type.getType(Object.class));
          indexIndex = newLocal(Type.getType(int.class));
          valueIndex = newLocal(Type.getType(double.class));
        }

        // Store
        mv.visitVarInsn(valueType.getOpcode(ISTORE), valueIndex);
        mv.visitVarInsn(ISTORE, indexIndex);
        mv.visitVarInsn(ASTORE, refIndex);

        // Reload
        mv.visitVarInsn(ALOAD, refIndex);
        mv.visitVarInsn(ILOAD, indexIndex);
        mv.visitVarInsn(valueType.getOpcode(ILOAD), valueIndex);

        super.visitInsn(opcode);

        mv.visitVarInsn(ALOAD, refIndex);
        mv.visitMethodInsn(INVOKESTATIC, RUNTIME_NAME, INVOKE_ARRAY_STORE,
    INVOKE_ARRAY_STORE_DESC, false);

        break;
      default:
        super.visitInsn(opcode);
      }
    }

    private Type getType(int opcode) {
      switch (opcode) {
      case DASTORE:
        return Type.getType(double.class);
      case LASTORE:
        return Type.getType(long.class);
      case AASTORE:
        return Type.getType(Object.class);
      case BASTORE:
        return Type.getType(boolean.class);
      case CASTORE:
        return Type.getType(char.class);
      case FASTORE:
        return Type.getType(float.class);
      case IASTORE:
        return Type.getType(int.class);
      case SASTORE:
        return Type.getType(short.class);
      default:
        return null;
      }
    }
  }
}
```

Appendix G Monitors

Class visitor to wrap the instructions `MONITORENTER` and `MONITOREXIT`.

```java
package org.matheus.runtime.instrumenter.visitor;

import org.objectweb.asm.MethodVisitor;

import static org.objectweb.asm.Opcodes.*;
import static org.matheus.runtime.SymbolTable.*;

import org.objectweb.asm.ClassVisitor;

public class MonitorVisitor extends ClassVisitor {
  public MonitorVisitor(ClassVisitor cv) {
    super(ASM5, cv);
  }

  @Override
  public MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions) {
    return new WrapperMonitorInsn(super.visitMethod(access, name, desc, signature, exceptions));
  }

  class WrapperMonitorInsn extends MethodVisitor {
    WrapperMonitorInsn(MethodVisitor mv) {
      super(ASM5, mv);
    }

    @Override
    public void visitInsn(int opcode) {
      switch (opcode) {
      case MONITORENTER:
        mv.visitInsn(DUP);
        super.visitInsn(opcode);
        mv.visitMethodInsn(INVOKESTATIC, RUNTIME_NAME, INVOKE_MONITOR_ENTER, INVOKE_MONITOR_ENTER_DESC, false);
        break;
      case MONITOREXIT:
        mv.visitInsn(DUP);
        mv.visitMethodInsn(INVOKESTATIC, RUNTIME_NAME, INVOKE_MONITOR_EXIT, INVOKE_MONITOR_EXIT_DESC, false);
        super.visitInsn(opcode);
        break;
      default:
        super.visitInsn(opcode);
      }
    }

    @Override
    public void visitMaxs(int maxStack, int maxLocals) {
      super.visitMaxs(maxStack + 1, maxLocals);
    }
  }
}
```

Appendix H Synchronized Methods

Class visitor to convert synchronized methods to a synchronized block.

```java
package org.matheus.runtime.instrumenter.visitor;

import org.objectweb.asm.ClassVisitor;
import org.objectweb.asm.MethodVisitor;
import org.objectweb.asm.Type;
import org.objectweb.asm.commons.AdviceAdapter;

import static org.objectweb.asm.Opcodes.*;

public class SynchronizedVisitor extends ClassVisitor {
  String className;

  public SynchronizedVisitor(ClassVisitor cv) {
    super(ASM5, cv);
  }

  @Override
  public void visit(int version, int access, String name, String signature, String
    superName, String[] interfaces) {
    className = name;

    super.visit(version, access, name, signature, superName, interfaces);
  }

  @Override
  public MethodVisitor visitMethod(int access, String name, String desc, String
    signature, String[] exceptions) {
    MethodVisitor mv;

    if (org.matheus.instrumenter.Util.hasAccess(access, ACC_SYNCHRONIZED)) {
      mv = super.visitMethod(access - ACC_SYNCHRONIZED, name, desc, signature,
    exceptions);
      if (org.matheus.instrumenter.Util.hasAccess(access, ACC_STATIC)) {
        return new StaticSynchronizedMethodVisitor(mv, access, name, desc);
      } else {
        return new SynchronizedMethodVisitor(mv, access, name, desc);
      }
    } else {
      return super.visitMethod(access, name, desc, signature, exceptions);
    }
  }

class SynchronizedMethodVisitor extends AdviceAdapter {
  SynchronizedMethodVisitor(MethodVisitor mv, int access, String name, String desc)
  {
    super(ASM5, mv, access, name, desc);
  }

  @Override
  protected void onMethodEnter() {
    super.onMethodEnter();
    mv.visitVarInsn(ALOAD, 0);
    mv.visitInsn(MONITORENTER);
  }

  @Override
  protected void onMethodExit(int opcode) {
    mv.visitVarInsn(ALOAD, 0);
    mv.visitInsn(MONITOREXIT);

    super.onMethodExit(opcode);
  }

  @Override
  public void visitMaxs(int maxStack, int maxLocals) {
    super.visitMaxs(maxStack + 1, maxLocals);
  }
}

class StaticSynchronizedMethodVisitor extends AdviceAdapter {

  StaticSynchronizedMethodVisitor(MethodVisitor mv, int access, String name, String
  desc) {
    super(ASM5, mv, access, name, desc);
  }

  @Override
  protected void onMethodEnter() {
    super.onMethodEnter();
    mv.visitLdcInsn(Type.getType("L" + className + ";"));
    mv.visitInsn(MONITORENTER);
  }

```

```
    @Override
    protected void onMethodExit(int opcode) {
      mv.visitLdcInsn(Type.getType("L" + className + ";"));
      mv.visitInsn(MONITOREXIT);
      super.onMethodExit(opcode);
    }

    @Override
    public void visitMaxs(int maxStack, int maxLocals) {
      super.visitMaxs(maxStack + 1, maxLocals);
    }
  }
}
```

Appendix I Object allocation without constructor invocation

```
package org.matheus.runtime;

import sun.reflect.ReflectionFactory;
import java.lang.reflect.Constructor;

/*
 * Source: http://www.javaspecialists.eu/archive/Issue175.html
 */
@SuppressWarnings("restriction")
public class SilentObjectCreator {
  public static <T> T create(Class<T> clazz) {
    return create(clazz, Object.class);
  }

  public static <T> T create(Class<T> clazz, Class<? super T> parent) {
    try {
      ReflectionFactory rf = ReflectionFactory.getReflectionFactory();
      Constructor<?> objDef = parent.getDeclaredConstructor();
      Constructor<?> intConstr = rf.newConstructorForSerialization(clazz, objDef);
      return clazz.cast(intConstr.newInstance());
    } catch (RuntimeException e) {
      throw e;
    } catch (Exception e) {
      throw new IllegalStateException("Cannot create object", e);
    }
  }
}
```