Jaos, a JVM for the Aos Kernel

Patrik Reali Institut für Computersysteme
ETH Zurich
CH-8092 Zurich, Switzerland
reali@acm.org

ABSTRACT

The Java VM is the key to the Java world. Implementing a JVM is close to implementing an operating system. To simplify the JVM construction, we build it on top of the Aos kernel, which provides an active object-based high-level programming model along with services like garbage collection and dynamic module loading.

The underlying high-level model of the kernel makes this project an exercise in programming model mapping. The extensive reuse of existing services and programming conventions allows to construct the JVM with a very modest manpower. Using the symbol table of the Active Oberon compiler opens the doors to language interoperability among compilation-units.

This paper provides an overview of the problems encountered and solutions adopted in the realization of the Jaos JVM; it also presents the first preliminary results obtained.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Aos, Java; D.4.7 [**Operating Systems**]: Organization and Design; D.2.12 [**Software Engineering**]: Interoperability

Keywords

JVM, Aos, Active Objects, Active Oberon, Java, System Design, System Implementation, Language Interoperability Support

1. INTRODUCTION

The Java Virtual Machine is one of the key components to Java success. Java programs can run on any JVM, and the JVM specification [8] is relatively easy to port to different run-time environments. The implementation of a JVM is closely related to the implementation of an operating system, because more or less the same components are required. To simplify and speed up the implementation of a JVM, preexisting software components can be used instead of implementing functionalities from scratch.

This paper presents the Jaos JVM—Java on Active Object System—implemented on top of the Aos/Bluebottle kernel[1] [10], a modern kernel and run-time environment first implemented for the Active Oberon language [6, 11, 12].

The Aos Kernel. The Active Object System Kernel (Aos) is a lean multiprocessor kernel developed in the spirit of the ETH Oberon Project. It is the base of the Bluebottle system and provides a runtime environment for the Active Oberon language, supporting active objects directly. It allows the construction of efficient active object-based systems that run directly on the hardware. Above the kernel layer is a flexible collection of modules providing generic abstractions for devices and services, e.g., file systems, user interfaces and networking.

The Aos kernel is currently implemented for Intel IA32 SMP multi- and single-processor systems and severals computers based on StrongARM-processors.

The Kernel provides a runtime environment for dynamically-loaded compilation-units, support for object-oriented programs, exception-handling, memory management with garbage collection, and multi-threading (using the active object model). It is an ideal base for strong-typed, imperative and object-oriented programming languages.

The Jaos JVM. The Aos model and the Java programming model have many affinities, thus it is very tempting to implement a JVM on top of it. In particular, most of the run-time services required to support Java are already available, allowing to implement the JVM with a limited effort. Second goal of the project is to prove that the Aos Kernel is generic enough to support languages and systems other than Active Oberon.

The core of Jaos is implemented using the Active Oberon language. The Java Libraries are provided by the GNU Classpath project [4]. The native implementation is kept to the bare minimum required to bootstrap the VM and to provide the native parts of the Java Libraries. The rest of the implementation is provided by the Java Libraries.

From services to data layout, Jaos reuses as much as possible from the underlying kernel. This allows to use a proven design, and to reuse much of the kernel services. By using the same symbol tables as the Active Oberon compiler,

[1]Aos stands for Active Object System, the programming model used in the kernel; Bluebottle is the codename of the current release, which includes the whole Oberon IDE

seamless language interoperability is also achieved. Interoperability is only limited by the differencies in the programming models.

The modular design of Aos lets the designer decide which modules to deploy, depending on the configuration of the target machine. In fact, Jaos can be configured to run alone on top of the kernel, in such a way that the result is very close to a Java native environment. Although being a generic kernel, Aos is small enough allow us putting a reduced Java native system on a floppy disc.

Paper Overview. This paper is structured as follows. Section 2 presents the main design choices in Jaos; section 3 introduces the implementation details of Jaos, in particular the model mapping from Java to Aos; section 4 relates the first preliminary results obtained with Jaos; section 5 indicates our future research directions; and section 6 concludes the paper by resuming the most important results achieved.

2. JAOS DESIGN

One of Jaos goals is to minimize the development work by leveraging the existing Aos software and APIs. Reusing the data-layout and the system components, in particular the Active Oberon compiler's symbol tables, creates the preconditions for language interoperability between Oberon and Java.

Only a few services required by Java are not available in kernel and had to be implemented in Jaos. The most important ones are interfaces[2] and exception handling, implemented during a 4 months student assignment [7].

The core of the JVM consists of a class loader, a bytecode compiler, and a linker. An important part of the JVM are the native methods that anchor the Java libraries to the underlying run-time environment.

2.1 Reuse-driven design

Jaos reuses the features of the Aos kernel at two distinct levels. First, it allocates the Java data structures using the same data-layout used by the kernel; the bytecode compiler has to enforce that all data manipulations are translated accordingly. This includes the hidden structures associated with every object instance, like the type descriptor. All instances with the same type share the same type descriptor, which contains informations for the type-accurate garbage collector, the method table, and the type hierarchy (for type testing). Another implicit convention is the calling convention.

Second, Jaos uses the kernel API to invoke services like loading a compilation-unit, creating new types, or allocating object instances. The kernel itself is designed to be extensible and system-independent, thus relies on a plugin pattern for all features that shall be provided by an application or system using it. This is particularly useful for all operations triggered by the kernel but implemented outside it, like loading compilations-units, loading metadata information, and handling exceptions.

2.2 Language Interoperability

Using the same data layout, data structures, and calling convention is a first step towards language interoperability. This allows to use compilation-units implemented in another language in a seamless way, completely avoiding glue code and routines.

The second important step towards automated language interoperability is the use of the same symbol tables to represent the metadata in a common way among al languages. This way, the data is available in a language-agnostic form, although using a model which may be closer to one language than another. To allow separate compilation with compilation-units from various languages, each language processor installs a metadata loader plugin into the symbol table. When the public interface information from an external module is queried, the symbol table asks each loader to provide such information.

In fact, this plugin design makes the symbol table independent of the persistency format used by the various language. Furthermore, each language is free to choose the persistency format that better suits it. This is a major advantage, because it allows us to take the Java classfiles as they are without having to first convert them into another format.

2.3 VM Core Components

The core components of the VM are the class-loader, the bytecode compiler, and the linker.

The *class-loader* internalizes a classfile, inserting the constant pool, the class and its member definitions into the shared symbol tables. During this step, it maps the Java types into Active Oberon types; allocation (i.e. assigning an offset to each field, and an entry in the method table to each method) is performed automatically by the symbol table. For each class, the loader checks if a native implementation exists, in which case it checks the definition's consistency (the native implementation must *at least* define the same members) and includes the additional member definitions present in the native implementation.

The *bytecode compiler* translates the Java bytecode into intel IA32 code. In particular, it encodes all data access instructions (`load, store, get, put`) and all `invoke` instructions using the kernel's data-layout and calling convention. To test different implementations, the compilers are designed as a plugins. We currently implemented two compilers: first, a pattern-expansion compiler, in which every bytecode is compiled to a fixed code pattern; second, an optimizing compiler using a virtual stack to minimize needless stack operations as described in [1]. Both compilers compile a whole classes on-demand. No code is compiled ahead-of-time.

The *linker* resolves dependencies between compilation-units; this includes the dependencies to the native implementations of methods. Module containing native methods are dynamically loaded, and jumps to the java method implementation are patched into them; the type descriptor and global data section[3] of the native implementation is used for by the Java implementation too.

2.4 Java Libraries

Jaos uses the GNU Classpath Java Libraries [4] out-of-the-box (version 0.5).

All the classes containing native methods must be declared in Active Oberon; the native methods are then given an implementation in Active Oberon. These methods are

[2]in fact, the newly introduced Active Oberon interfaces reuse the Jaos interface's implemententation

[3]the static fields of the class are stored there

usually gateways to low-level functionalities provided by the underlying kernel.

2.5 Kernel Plugins

To react to kernel-triggered events, Jaos installs three plugins in the kernel: a metadata loader, a compilation-unit loader, and an exception handler.

The metadata and compilation-unit loaders allow the kernel to ask an application to provide metadata, respectively compiled code, on-demand. This allows separate compilation to import the definitions of units compiled in another language, and the kernel to load the needed modules whenever the user executes a command, or loading one module requires other modules.

The exception handler plugin is installed per-thread and invoked whenever an exception happens during the code execution.

3. JAOS IMPLEMENTATION

The implementation of Jaos in an exercise in mapping programming models and information from Java to Aos and back. Jaos consists of three groups of modules: the VM core, the Aos plugins, and the libraries' native implementations. This section presentes the implementation details of Jaos, focussing on the programming model mapping, and the changes made to the kernel to make it system independent.

In this section we present the generated in code using Active Oberon. This is possible, because the kernel uses the same types as the language. However the code is converted directly to the native machine language, only the native methods are actually in Active Oberon.

3.1 Jaos Core

Jaos Bytecode Compiler. Jaos has currently two compilers: a pattern expansion compiler, and an optimizing compiler.

The optimizing compiler issues all load operations in a virtual stack [1]. The stack elements can be registers, absolute memory addresses, register-indirect addresses, and constants. Code is emitted only when the operation cannot be fitted in the virtual stack. This allows to avoid pushing of elements onto the real stack and use register and Intel's complex addressing modes instead as much as possible. This technique also bears some similitudes to the Oberon's compiler items [9]

Both compilers ensure that the Java bytecode is compiled according to the kernel's data-layout. In particular, the Java calling-convention must be converted into the Aos calling-convention, whose main difference lies in the position of the self parameter in a virtual invocation[4]. In the pattern-expansion compiler, this is achieved by pushing a copy of this parameter before the invocation, and by removing the unused original parameter after the call. The optimizing compiler can avoid this copy operation: if no method parameter has been pushed on the physical stack, the parameters in the virtual stack can be reordered according to the calling convention, and only then flushed to the actual stack.

[4]in Java it is the first parameter, in Aos the last one

Some bytecodes require information which is not available to the compiler, or whose computation would require to generate a too much of code. In this cases, a system-call to a native routine is emitted instead. Table 1 shows the system-calls used.

System Call	*Description*
`Lock`	lock an object
`Unlock`	Unlock an object
`New`	Instantiate an object
`NewArray`	Instantiate an array of primitive types
`NewArrayA`	Instantiate an array of objects
`NewMultiArray`	Instantiate a multidimensional array
`Throw`	Throw an exception
`InstanceOf`	Check the dynamic object type
`CheckInitialized`	Perform static class initialization, if needed
`ILookup`	Lookup interface method table
`Trace`	Trace a method invocation

Table 1: Jaos System-Calls

Java Libraries. The Java Libraries belong to the platform specification and are by far the biggest part of the software required by a JVM. These libraries are provided as Java classfiles, and are loaded and executed by the JVM like every other Java class. Classes containing native methods must be declared in Active Oberon, and an implementation for each native method must be provided. The Java Libraries define the whole JVM; the JVM itself relies on the libraries to perform many operations. For a few operations we had to provide a native implementation, because the JVM is not completly bootstrapped when they are needed: such operations are in the classes `Object, String, System, Runtime, Throwable`.

Package	*Classes*
`java.lang`	`Double, Float, Class, Object, Math, String, System, Runtime, Thread, Throwable`
`java.lang.reflect`	`Constructor, Method, Field`
`java.io`	`File, FileDescriptor, FileInputStream, FileOutputStream`

`Table 2: Classes with an Active Oberon implementation`

The classes with native methods currently implemented in Active Oberon are shown in table 2.

When available, the Active Oberon implementation overrides the Java one even if the Java method is not native. This is particularly useful when debugging and bootstrapping the VM.

Loading, Compilation, Linking, and Initialization time. Each Java class is in a state. The states are *non-loaded*, *loaded*, *compiled*, *linked*, and *initialized*. Because classes are related among each other, one class' state transition may trigger transitions in other classes too. Table 3 shows the static state dependencies among classes. These dependencies are stronger than those required by the JVM Specification.

State(c)	*Precondition*
loaded	superclass of c is loaded
compiled	superclass of c is compiled interfaces of c are loaded all classes referenced in c are loaded
initialized	superclass of c is initialized

Table 3: Static State Dependencies among Classes

The static dependencies are designed to simplify the JVM. Having the superclass already loaded allows to perform field and method allocation while loading one class; requiring all referenced classes to be loaded simplifies the code generation because external member access can be encoded using the allocation information, thus avoid patching the addresses at a later stage, and reducing the administrative information kept.

On the down-side, these dependencies can cause classes to be loaded needlessly. However, only classes—and their superclasses—actually executed are compiled.

Class static initialization is performed using the technique presented in Cierniak et al. [3]: the compiler inlines a `CheckInitialized` system call before each access to an external static member. This call checks if a class is already initialized (if not, the static initializer is executed); the system call is then removed from the caller[5].

The compiler emits `CheckInitialized` only when the referenced class is in the loaded or compiled state at the time of compilation. In average, around 70 - 80% of the system calls can be avoided this way.

3.2 Kernel Plugins

A few changes to the Aos kernel were required to support Jaos. Common to all of them is to remove direct dependencies to the Oberon System by using multiple installable plugins. For each service provided outside of the kernel, the kernel has to define a service interface with the required operations, and provide methods to install and deinstall the plugins. When multiple plugins are supported, a function to determine which plugin is to be called must be supplied.

With these modifications the kernel becomes de facto system-agnostic and can provide run-time support to multiple systems at the same time. This is of course needed, as Jaos is able to interoperate with Active Oberon and both systems must be able to run on the same kernel at the same time; both systems can also execute alone.

Module Loader. The kernel module loader has been split into a shared part, and a plugin for loading the Active Oberon object-files. The common part defines the loader plugin interface, and provides two methods to add, respectively remove, the plugins.

[5]the system call is overwritten with `NOP` instructions.

```
TYPE
  (** load an object file *)
  LoaderProc* = PROCEDURE (
      name, fileName: ARRAY OF CHAR;
      VAR res: LONGINT;
      VAR msg: ARRAY OF CHAR): Module;

PROCEDURE AddLoader*(ext: ARRAY OF CHAR; proc: LoaderProc);
PROCEDURE RemoveLoader*(ext: ARRAY OF CHAR; proc: LoaderProc);
```

Jaos provides an own loader plugin for loading Java classes into the system. This plugin forward the requests to the classfile loader, the compiler, and the linker, and returns the compiled and linked compilation-unit to the kernel.

This simple modification has interesting consequences: the kernel is able to load modules persisted in different object-file formats and becomes de-facto format agnostic.

Symbol Tables and Metadata Loader. Instead of implementing a new symbol table for Jaos, we decided to reuse the existing Active Oberon compiler's symbol table.

We removed the dependencies between the symbol table and the compiler, such that the tables can be shared among more than one application.

The symbol-table persistency mechanism has been moved out of the table and wrapped in an installable plugin. This plugin loads the persisted Active Oberon symbol-files into the symbol-table. Jaos also provides such a plugin to provide metadata on request from the Java classfiles.

This change makes the symbol table independent of any persistency format, and allows to perform separate compilation in a format agnostic fashion. In practice, it doesn't matter—and the compiler won't be able to tell the difference—whether the external information comes from an Active Oberon symbol-file or from a Java classfile.

Besides the obvious code reuse aspect, this design allows language interoperability with a compilation-unit granularity. An Active Oberon module can import a Java class: the symbol table will provide the interface information from the Java class in a transparent fashion. The common model used here is the Active Oberon type system. Interoperability from Active Oberon to Java is possible but limited, because Java's type system is only a subset of Active Oberon one: only types supported by Java can be mapped.

Interface Support. The Aos Kernel provides no support for interfaces and interface calls. This required to modify the symbol table implementation to provide interfaces, and the runtime support for storing the interface method tables.

To avoid changing the type descriptor format, we decided to implement an global shared repository for all interface method tables. Class and interface type are used as index to retrieve the corresponding table from the repository. Interface method tables and (class) method tables have the same layout, thus the compiler can reuse the same code pattern to invocation.

```
PROCEDURE Insert*(class, interface: TypeDescriptor; vtable: VTable);
PROCEDURE Lookup*(class, interface: TypeDescriptor): VTable;
```

The shared repository is currently implemented with a hashtable, but the abstract interface allows us to change the implementation with a more performing one without breaking the code.

The linker creates an interface method table for each interface implemented by a class and registers it to the shared

hash-table. Interface calls (`invokeinterface`) extract the type descriptor from the self-pointer and query the repository for the method table registered with this type descriptor and the interface descriptor. If no method table is found, the class does not implement the interface.

The interface support is now part of the kernel. With the run-time and symbol table support available, adding interfaces to the Active Oberon language required mostly a parser change.

Exception Handling. The kernel catched all interrupts and exceptions. Exceptions not handled by the kernel are forwarded to a user-defined handler. Instead of having one global handler, the kernel was modified to associate one user-defined handler with each thread .

This approach implicitly assumes that the whole code executed by one thread uses the same handler, and thus is implicitly written in the same language. Although not very flexible, this nevertheless suffices our needs, because the Active Oberon language has no exception handling yet, and only Java handling is used.

We implement zero-overhead exception handling [5]. The generated code in not instrumented; the compiler substitutes the bytecode-offsets of each `Exceptions` attribute entry with the offsets within the compiled-code; the linker then registers the absolute code address to the exception handler. The exception handler keeps a balanced tree of all the registered entries to make access faster [7].

Whenever an exception happens, the kernel calls the Java exception handler. The handler searches for an entry matching the exception location. If no entry is found, or the matched entries do not handle the exception thrown, the procedure activation frame is removed, the caller's program counter taken, and the lookup repeated.

3.3 Type Mapping

This section explains how Java types are mapped to the kernel types. We use the Active Oberon type names and concepts for clarity.

Basic Types. All Java basic types can be mapped to an equivalent Active Oberon type, but `char`. For experimental purposes, the symbol table already provides character types with 8, 16, and 32 bit widths: Jaos maps `char` to the corresponding 16-bit char type.

Active Oberon uses ASCII-encoded chars, whereas Java Unicode-encoded chars. For this reason, interoperating between compilation-units requires character conversion functions. In fact, migrating the whole kernel and Active Oberon system to the Unicode charset could be also considered. Table 4 shows the mapping of the basic types.

Java Type	*Oberon Type*	*Java Type*	*Oberon Type*
`byte`	SHORTINT	`boolean`	BOOLEAN
`short`	INTEGER	`float`	REAL
`int`	LONGINT	`double`	LONGREAL
`long`	HUGEINT	`char`	CHAR16

Table 4: Basic Type Mappings

Classes. Java classes define both static and instance members, and are at the same time visibility, compilation and deployment units. No single type in Active Oberon fullfils all these requirements, thus we have to map Java classes to a composition of Active Oberon constructs: module and class.

A Java class is mapped to a module having the full class name (e.g. `java.util.Properties`) containing a object type (named `Class`[6]). The class' static members are declared in the module, whereas the instance members are declared in the object.

Public and private visibility modifiers are converted to the equivalent Oberon visibility modifiers[7]. Members with protected and package visibility are declared as public in Oberon. Jaos keeps track of the Java visibility modifiers, and enforces them according to the Java semantic; an Active Oberon module would nevertheless be able to access them.

Interfaces. Interfaces are mapped—similar to classes—to a module containing an interface. The module as code container is required because an interface may conterintuively have a static initializer, whenever the static final fields are no compile-time constants.

Without this, it would be possible to map interfaces only to symbol table entries.

Arrays. Java arrays are dynamically allocated and dynamically sized; they have a class semantic (methods defined in `Object` can be invoked on an array), but are no compilation-units, and carry no code.

To achieve the same semantic, we declare a class containing one field, a dynamic array. Because the JVM has only eight array types[8], and because no code is associated to them, we declare these eight array types once in the JVM, and map the Java declarations to them instead of redeclaring them everytime.

For Active Oberon, the single declaration of the array types avoids compatibility problems due to the name-compatibility semantic of the Active Oberon types.

3.4 Concurrency Mapping

The Java concurrency model consists of threads, reentrant locks, and signals. Active Oberon has active objects with non-reentrant locks, and arbitrary conditions. The Java features must be mapped to semantically equivalent Active Oberon constructs. This mapping is performed in the `java/lang/Object` and `java/lang/Thread` classes native implementations, because Java concurrency is realized in the class library.

Threads. In Aos, each object instance can have an associated thread of execution[9].

With the exception of `java.lang.Thread`, all Java object are instantiated without thread (i.e. as passive objects). According to the Java semantic, the execution of the thread's

[6] we first used the class name, but this caused conflicts with some classes that declared fields with the same name

[7] note that in Oberon private members have a module-wide visibility, thus private instance members can access private static members and vice-versa, like in Java

[8] one per built-in type and an array of references

[9] passive objects are special case: they have no thread. Note that the life-span of an object instance can be longer than the associated thread.

code has to be delayed until the `start()` method is invoked. The following class shows the native implementation of the `Thread` class.

```
TYPE
  Thread* = OBJECT(JavaLangObject)
    VAR
      state: LONGINT;
    ...

    PROCEDURE Start*;
    BEGIN {EXCLUSIVE}
      state := running
    END Start;

  BEGIN {ACTIVE}
    BEGIN {EXCLUSIVE}
      AWAIT(state = running);
    END;
    SELF.run;  (* overloaded method *)
    state := dead
  END Thread;
```

In our active object model, objects are truly self-controlled. To force each developer in this mindset—and to avoid the the invariant breaches caused by forcefully freeing all the locks owned by one thread—the kernel provides no means to terminate or passivate a thread (object can only terminate themselves). For this reason, we cannot implement the `Thread.suspend()`, `Thread.resume()`, `Thread.kill()` methods. This is no huge problem, because those methods are deprecated, and we never encountered classes using them.

Locks. The object locking semantic of Active Oberon and Java is almost the same. Both languages provide atomicity by making each object instance a monitor and providing exclusive locks on each instance. The difference lies in lock reentrancy: Java allows a thread to lock an object multiple times, whereas Oberon doesn't.

To implement lock reentrancy, the native implementation has to keep the count of the times a locks was taken by one thread. Only one counter is needed, as other threads a denied of access to the object while it is locked. This information is associated to the lock itself: Java locks are instance based and every instance is an subclass of `Object`, thus we keep track of the count in that class.

```
TYPE
(* Native implementation of Java.lang.Object *)
  JavaLangObject = OBJECT
    VAR
      depth: LONGINT;  (* Lock Depth *)
      ...
  END JavaLangObject;

PROCEDURE JavaLock(obj : JavaLangObject);
BEGIN (* no synchronization required *)
  IF AosActive.LockedByCurrent(obj) THEN
    INC(obj.depth)
   ELSE
    AosActive.Lock(obj);
    obj.depth := 1
  END
END JavaLock;

PROCEDURE JavaUnlock( obj : JavaLangObject );
BEGIN
  DEC(obj.depth);
  IF obj.depth = 0 THEN AosActive.Unlock(obj) END
END JavaUnlock;
```

There is one caveat to this implementation: calling `wait()` releases the lock. This is done automatically by our kernel synchronization primitive (see next section), but the count is not saved automatically, in case another thread locks the object before the passivated one. Thus we have to save the lock count whenever the lock is released (i.e. when `wait()` is called).

```
PROCEDURE wait*();  (* Java.lang.Object.wait() *)
VAR savedCount: LONGINT;
BEGIN
  savedCount := SELF.depth;
  ...
  SELF.depth := savedCount
END wait;
```

Synchronization performed in an Active Oberon objects (e.g. when reading from a file's buffer) poses no problem, because Active Oberon objects can be locked only once, thus no counter is needed.

Synchronization. Java synchronization uses signals (`wait`, `notify`, and `notifyAll`), whereas Active Oberon uses arbitrary boolean conditions (`AWAIT(condition)`).

We implement signals using conditions with a ticketing algorithm [2]: a call to `wait` takes a numbered ticket, and awaits for the number to be called; `notify` calls the next available ticket, `notifyAll` calls all the available tickets.

```
TYPE
  (* Java.lang.Object native implementation *)
  Object* = OBJECT
    VAR
      in: LONGINT;    (*next ticket to assign*)
      out: LONGINT;    (*next ticket to service*)

    PROCEDURE CheckLock;
    BEGIN
      IF ~AosActive.LockedByCurret(SELF) THEN
        Throw("Java/lang/IllegalMonitorStateException")
      END
    END CheckLock;

    PROCEDURE wait*;
    VAR ticket, savedDepth: LONGINT;
    BEGIN
      (* wait while (out <= ticket < in) *)
      CheckLock;
      savedDepth := depth;
      ticket := in; INC(in);
      AWAIT(ticket - out < 0);
      depth := savedDepth;
    END Wait;

    PROCEDURE notify*;
    BEGIN
      CheckLock;
      IF out # in THEN INC(out) END
    END Notify;

    PROCEDURE notifyAll*;
    BEGIN
      CheckLock;
      out := in
    END NotifyAll;

    ...

  END Signal;
```

Java's `wait` can also have a timeout parameter. In this case, a timer must awake the awaiting thread whenever the timeout is reached, and the ticket is returned; the implementation of `notify` must then take into account the returned ticket when chosing the next available one. This implementation is not included here.

3.5 Bootstrapping

Loading a Java class can have a domino effect, and require to recursively load other classes[10]. Furthermore, the JVM itself uses the Java implementations from a few key classes (`Object`, `String`, `Thread`, `Throwable`, `System!`, and `Runtime`), which must be preloaded by the JVM. A few exception classes are also preloaded, because they cannot be loaded by the exception handler. The JVM initialization causes the loading of little less than 70 classes.

Thus, even the execution of a simple HelloWorld program requires the compilation of 75 classes. One third of these classes are the many stream-kinds and converters involved.

This is mostly due to the design of the Java language, in which classes cannot be topologically sorted. This problem is already rooted in `java.lang.Object` which uses its own subclasses (e.g. `java.lang.String`). This makes the incremental development of a JVM a daunting task, because everything must work and be supported at once (*"all or nothing"*).

To mitigate this problem, we instructed the class-loader to load only one limited set of classes[11]. We also did override some methods of the Java Libraries to avoid them from calling unimplemented features. These temporary native implementations dump the method's parameters to prove their correctness, or provide a simplified native implementation.

As work progressed, it was possible to reduce the class black-list and remove the native overrides, thus in fact removing code from Jaos and taking advantage of the existing Java code.

The method overriding trick is also used when Jaos needs to execute Java code before the VM is bootstrapped. As an example, all the strings created in the VM (e.g. the strings in the constant pool) are instances of `java.lang.String`, but when the first classes are loaded, the VM is still not bootstrapped. Thus, we have to provide a string constructor in Oberon.

4. RESULTS

The current Jaos implementation consists of a VM core, the GNU Classpath libraries, and the native method implementation for part of the libraries. The whole Java VM specification is implemented; furthermore, Java concurrency, exception handling, file support, and console output are provided. The GNU Classpath libraries are used out-of-the-box, and Jaos depends on their progress.

The main missing features are the native implementation for network and graphical user interface (AWT and Swing).

Nevertheless, the first results are very promising and Jaos is already able to execute a few non-trivial Java applications. These include the SPEC benchmark `201-compress`, `202-jess`, and `209-db`.

4.1 Jaos Size and Deployment

Jaos runs directly on the Bluebottle kernel. Additional services and applications are either kernel plugins or run on top of the kernel. This allows to have different Jaos configurations, depending on the installed components.

[10]Measured on GNU Classpath 0.5, the transitive closure (classes referenced directly and indirectly in the constpool) of `java.lang.Object` contains 394 classes.

[11]it returned `java.lang.Object` for the censured classes

Please note that these are preliminary results: in particular, the Jaos and Paco projects are still in development and contain additional debug, trace and experimental code.

Components	*Count*	*Code*	*Vars*	*Consts*
Jaos	25	156386	62396	13384
ZipFS	8	47052	2164	1024
Paco	10	94009	3960	4364
Utilies	4	17483	832	652
File system	7	62370	116	1888
Service support	5	33754	136	1624
Kernel	10	50460	18100	1308

Table 5: The components sizes

The *Service Support* modules include the Oberon module loader, the Oberon exception-handler, and the streams; the *File System* modules include the file system implementation, a RAM-disk, and an IDE driver; the *Utilites* include the keyboard, mouse, and display[12] drivers; *Paco* part of the Active Oberon compiler (the symbol tables and part of the code generation modules[13]); *ZipFS* is a file system's plugin to mount zip files as a file system (in this case `glibj.zip`).

Components	*Count*	*Code*	*Vars*	*Consts*
Jaos (on-demand)	9	3710	28	744
HelloWorld.Java	75	111247	176	300
201-Compress	117	196557	528	468

Table 6: Optional components sizes

The previous tables shows the sizes of the modules implementing native Java functions that are loaded only on demand, and the size of the compiled Java classes used by two Java applications.

For the above results, it seems to be feasible to put the whole Jaos including a reduced set of libraries on a single floppy disc.

4.2 Jaos speed

This sections shows the preliminary results obtained with Jaos. We did absolutely no profiling to try to understand and improve where the bottlenecks are. With little effort, it should be possible to improve these results.

Table 7 presents our results, which are the average of 3 runs on a Dell Latitude 600 machine with a Pentium-III 750 MHz processor and 256 MB of memory.

5. FUTURE WORK

Altough already functional, Jaos can be improved in many ways. The missing native implementation for the network

[12]in this case a Permidia2 driver

[13]these will be removed in a future refactoring phase

Test	*Time [s]*
201 Compress	46.3
202 Jess	50.0
209 DB	127.9

Table 7: Preliminary Benchmark Results

and the graphical user interface is the only real missing feature. Otherwise, the platform will be useful for understanding performace issues of a Java system. In particular, we still have to profile the benchmarks and find weaknesses and bottlenecks to optimize in our implementation. We think that some performance issues may be hidden in the library implementation, and would like to prove or refute this thesis; in particular, we suspect `java/lang/String`—combined with our mapping—to be sub-optimal.

Another interesting application is to retarget Jaos for the StrongARM processor, the other processor family supported by Aos. This requires to write a new JIT-Compiler, whereas the rest of the system—written in a high-level language—should not change.

We are still considering if interoperability is an advantage or a burden. The use of the shared symbol tables makes the JVM 20% bigger. Only an extensive usage of this feature can legitimate such costs.

6. CONCLUSIONS

This paper presented our experience with Jaos, a Java VM build on top of the Aos Kernel. The Aos Kernel provides an high-level and typed native runtime environment. It supports an active object programming model (thus classes and concurrency), and provides many services like garbage collection and dynamic module loading.

We managed to construct a whole JVM minimizing the implementation and maximizing the reuse of existing components and design. The man-power required by the implementation was modest: it was the authors spare-time pet project plus a student's diploma thesis[14].

Thanks to the extensive use of the plugin pattern, it was possible to build a different system on top of Aos. In fact, the JVM can run as only system on top of the kernel. This proves that the Aos Kernel is generic enough to successfully support systems other than Active Oberon.

Besides the implementation simplifications, the high-level programming model used by the kernel makes the construction of a JVM (or other similar system) an exercise in programming model mapping, but without sacrifying effeciency and performance because we emit native code.

The use of the same structures and programming conventions, associated with reuse of the Active Oberon symbol tables, provide a rudimental language interoperability platform.

7. REFERENCES

[1] A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. Parikh, and J. Stichnoth. Fast, Effective Code Generation in a Just-In-Time Java Compiler. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI-98)*, ACM Sigplan Notices, pages 280–290. ACM Press, 1998.

[2] G. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley, 1991.

[3] M. Cierniak, G.-Y. Lueh, and J. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, ACM Sigplan Notices, pages 13–26. ACM Press, May 2000.

[4] GNU Classpath. http://www.classpath.org/.

[5] S. Drew, K. Gouph, and J. Ledermann. Implementing zero overhead exception handling. Technical Report 95-12, Faculty of Information Technology, Queensland University of Technology, 1995.

[6] J. Gutknecht. Do the Fish Really Need Remote Control? A Proposal for Self-Active Objects in Oberon. In *Proc. of Joint Modular Languages Conference (JMLC). LNCS 1024*, Linz, Austria, Mar. 1997. Springer Verlag.

[7] R. Laich. A Java Virtual Machine for Aos. Master's thesis, Institut für Computersysteme, ETH Zürich, 2000.

[8] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, 2nd edition, 1999.

[9] H. Mössenböck. Compiler Construction - The Art of Niklaus Wirth. In L. Böszörményi, J. Gutknecht, and G. Pomberger, editors, *The School of Niklaus Wirth*, pages 55–68. dpunkt.verlag/Copublication with Morgan-Kaufmann, 2000.

[10] P. Muller. *The Active Object System – Design and Multiprocessor Implementation.* PhD thesis, Institut für Computersysteme, ETH Zürich, 2002.

[11] P. Reali. Active Oberon Language Report. http://www.bluebottle.ethz.ch/languagereport/, Mar. 2002. Bluebottle System Documentation.

[12] P. Reali. *Using Oberon's Active Objects for Language Interoperability and Compilation.* PhD thesis, Institut für Computersysteme, ETH Zürich, 2003.

[14] a 4 month work