Actor Virtual Machine

Abstract

This paper documents the results of a class research project examining a Virtual Machine designed specifically to support the Actor Model of programming. The Actor VM uses a unique register-based design, a small set of primitive types, and futures for asynchronous communications.

Keywords: Actor Model, Virtual Machine

1 Introduction

The Actor Model of programming [1] encapsulates state within concurrent entities called Actors. Actors interact by passing messages and never sharing memory. Message processing is effectively atomic (cannot be interrupted by any other message), thus protecting the integrity of the Actor's state. An Actor's message address can be shared but not guessed. This allows reasoning about which actors could receive messages in the future -- a vital precondition for garbage collection. More importantly, this is also a vital precondition to support an Object-Capability model of security [11].

A number of Actor programming languages exist today [2]. The Java Virtual Machine (JVM) is a common target for Actor compilers because of the prevalence of the JVM. Microsoft's Common Language Runtime [4] provides another widely used Virtual Machine (VM). Both of these VMs are stack-based and support a variety of languages, but are not specifically designed to support the Actor model. The JVM has efficient implementations on many different platforms, but since it is specifically designed for Java, it is sometimes encumbered when being used for other languages [5]. Interestingly, the Dalvik JVM is a register-based machine [17]. The Erlang VM evolved from a stack-based machine to a register-based machine [10]. The Lua VM [7] is a small and elegant register-based virtual machine. While this project does not use the Lua VM, this VM is a major inspiration to our design.

1.1 Contribution

The general contribution of this research is to provide an example of a VM specifically designed to implement the Actor model. The Actor Virtual Machine is an investigation into techniques that could inform the development of Actor languages or virtual machines. Specifically, a unique register-based design is used instead of the common stack-based VM design. Other features of the Actor VM are borrowed and adapted from existing research, but are perhaps employed in unique ways.

1.2 Context

The Actor VM was developed in tandem with an Actor language called Vole. A detailed description of Vole is beyond the scope of this paper, but a brief overview is in order.

Vole is a pure Actor language, not mixing functional or procedural paradigms. Smalltalk makes everything a class, but Vole follows the Java mold where there are a number of primitive types directly supported by the language, leaving everything else to be developed as actors (classes). In Vole, classes and Actors are synonymous, and methods are messages.

Messages are defined with a set of parameters and a set of return values (either set can be empty). These values are transported as immutable tuples so that they can use pass-by-reference within a VM and only need to be serialized for communications between VMs. Messages can be gated using a predicate attribute (simply a Boolean attribute that is evaluated when the message could be received), similar to guarded task entries in Ada [18].

1.3 Usage

The Actor VM is intended to be used in affiliated groups. A user must grant access to a material resource (like disk space, internet ports, or a database) to a particular VM when it is started. Applications running within the affiliation can then access those resources via messages to actors executing in the controlling VM. Unless that VM allows code migration [13] (CPU time is another material resource so migration will be a controllable capability), malicious code can only interact with the resources through messages to trusted code.

2 Actor VM Features

The driving class within the Actor VM is the Machine class, which manages the resources of the system, including the running instances of actors. Instances control the scope of execution using frames. Frames represent the execution of an external message or an internal subroutine call (private message), providing access to the appropriate registers. Registers can contain primitive data as is described below in the type model.

2.1 Type Model

In Smalltalk, everything is a class. Similar to Java, the Actor VM divides the world into primitive types (of which there are 12) and actors. Objects (actor instances) exist within the virtual machine, but can only be accessed using actor references (a primitive type) that “point” to an instance of an actor. Actor references can be passed in messages. Other primitive types include 64-bit signed integers, 64-bit floating-point numbers, 64-bit unsigned integers (the original versions of both Java and Ada made the mistake of excluding unsigned integers), immutable strings, immutable tuples, tuple maps (folding instructions for tuples), Booleans, Actor definitions (to define the types of actors), message definitions, futures [9], and a null-object type called "Unset".

2.1.1 Tuples

Tuples are immutable. Once a tuple has been "folded", it cannot be changed. It can be replaced, but the original tuple will continue to exist as long as references to it are in existence. A reference counting data structure is used to implement tuples. Folded tuples can be copied and are the foundation of both messages and responses. The semantics of Vole require that tuples be built from non-consecutive data within the local scope. The image of origami inspired the use of the verb "to fold". Tuple maps were the easy answer to achieve this folding. Tuple maps are built by the compiler and stored as constants. There currently is no facility in the VM for manipulating tuple maps.

2.1.2 Messages

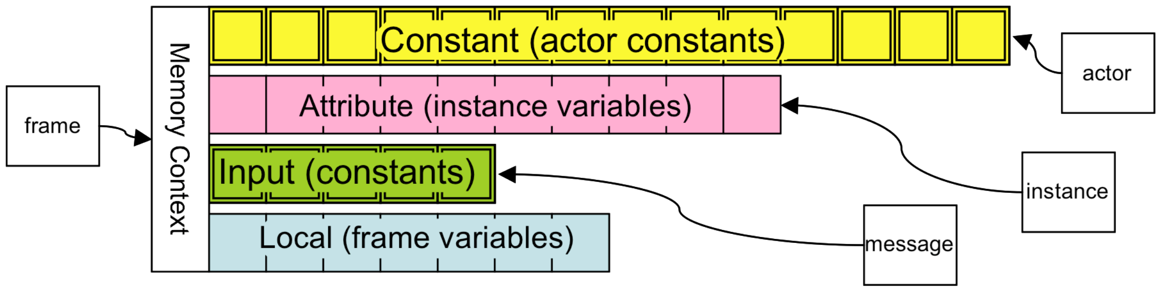
Messages are not primitive types in the Actor VM, but they play a central role in any Actor system. The payload of a message is a tuple and is therefore immutable. This allows the data to be shared while providing thread safety. Messages also carry metadata that includes the actor reference of the recipient, the sender, and the "reply to" instance. The Actor VM supports tail calls and will propagate the "reply to" reference to the next destination so that the final response can be sent directly back to the instance that is waiting for it.

Messages also carry the future used by the sender to bind the response and guard against premature use.

2.2 Registers

Code executing in the VM has a scope from which it can read and write data. Since the Actor model only allows actors to manipulate data that is local to the actor, there is no need for general access to memory. All data within scope is therefore stored in a virtual register.

Each Actor instance has its own stack of register sets. As Actor methods are invoked, a new execution frame is created with its own set of registers. Register sets have four arrays of general-purpose registers: input, local, attribute, and constant. The "input" array of registers contains the parameter array from the invoking message and is constant. The "local" array is where the method code stores its local variables. The size of this array is determined at compile time. The "attribute" array holds the actor instance's attributes and is constrained at compile time. Finally, the "constant" array contains all of the constants used by the actor instance in any of its code. A register set is implemented with four pointers to register arrays. The "constant" array exists for the lifetime of the VM. The "attribute" array exists for the lifetime of the instance. The "input" array pointer is copied from the invoking message (tuple). Finally, the "local" array is allocated for use by the frame, and is deallocated when the frame is popped.



Each register can hold any of the VM primitive data types. Once set, the type of data within a register can't be changed without causing a runtime error that will intentionally terminate the VM.

Register arrays are limited to 1024 registers, a number that should be sufficient in all but the most extreme cases.

2.3 Actors

Within the Actor VM code, the Actor class is responsible for the concept of a compiled Actor (type). Actors have an association with the Actor they are inherited from (if one exists). The Actor VM supports single inheritance. Like Java and Ada, Vole will allow a class to have multiple interfaces. Instances have associations with their Actors so that they can access the Actor constants and the compiled "word code" of the Actor messages.

2.4 Instances

As a class defines the type of an object, an Actor defines the type of an instance. Instances contain the state of a running actor, including a stack of execution frames that have been invoked by messages passed to themselves. Instances also represent concurrent threads of execution within the VM.

Instances have a unique "name" called an actor reference. These names can be shared but not guessed. There simply is no programmable facility within the VM to construct actor references.

When an active instance receives a new message, the message is queued until the instance completes its current processing. When an idle instance receives a new message, it creates a frame of execution to process that message. Synchronous messages to the same instance ("self calls" or "gosub") must be treated differently in that they suspend and push the current frame onto a stack and start a new frame.

2.5 Frames

One issue with Java thread switching is the overhead needed to save a stack-based context [5]. Kilim circumvents this using a “weaver” to modify compiled bytecode [15] and achieve a lightweight abstraction of a thread. This project took a different path and chose a register-based VM architecture. Context switching is as easy as changing the Workspace Pointer in the old TMS 9900 microprocessor [6]. The TMS 9900 microprocessor kept its general-purpose registers in RAM outside the CPU. Similarly, the Actor VM uses register sets that are outside of the virtual CPU. As subroutines are invoked, the instance pushes a new frame onto its stack. Unlike the TMS 9900, each frame has its own Program Counter to keep track of the point of execution within the frame.

When a frame completes, it is popped off the stack and execution resumes at the Program Counter of the top frame (or the instance turns idle and waits for a new message). Another aspect of completing is returning a response to the instance that invoked the frame (be it a self call or another actor). The frame maintains the “reply to” actor reference along with any associated future, and makes it available at the end. This meta-data is not available to the running instance because it is considered private to the caller.

2.6 Processor

The Actor VM uses a Processor class to interpret the 32 different VM instructions. Each instruction occupies a 64-bit word in the “word code” file and in memory. Instructions use an Ada variant record. Variant records can be thought of as C unions of structures where there is a field in every variation that defines the type of variation. The Op Code field specifies which instruction is contained in the record.

Instructions contain zero, one, two, three, or four register references. A register reference is a 12-bit structure that specifies which register set is being referenced (4 values, 2 bits), and the offset into the array (10 bits, thus the limit of 1024 registers in a register array). Except for instructions that require four register references, references are laid out on 16-bit boundaries to reduce bit manipulations. Only one instruction format requires tight packing of four register references.

Instructions can be grouped into four categories: messaging, control flow, data manipulation, and machine control. See appendix A for more details.

The responsibility of the Processor class is to interpret the instructions. Setting and getting registers is therefore delegated to the frame and resource utilization is delegated to the machine.

2.7 Machine

Within the Actor VM there is a Machine class that is responsible for all resources available to the executing actor instances. Each instance is given one of four states: active (if it is currently executing), idle (if it has completed all processing and is waiting for a message), blocked (if it is waiting for a response to a message), or deferred (if it's resource request can't be accommodated immediately).

2.7.1 Message (and Response) Passing

The machine routes messages locally or perhaps remotely to an affiliated VM. If the message can't be consumed immediately, the machine queues the message for future delivery. Messages requiring a response are given a future [?]. The future is immediately supplied to the sending instance in place of a response. When the response is available the future will be replaced by the response tuple. In the meantime, an instance that finds a future in place of an actual response will become blocked until the response is delivered.

It is important to support both synchronous and asynchronous message semantics. One way to implement (apparent) synchronous messaging is to create closures that will eventually process the return message, as is done in SALSA [14]. This limits what can be done in the post-return part of the message processing because that part could be executed after the actor instance has processed other messages. We chose to implement use futures instead.

2.7.2 Flow Control

If a message producer runs faster than its corresponding message consumer, messages can accumulate in the machine's message queue. It is the responsibility of the machine to manage the message queue resource. This is achieved by deferring execution of the producer to some later time to allow the consumer time to catch up.

2.7.3 Garbage Collection

The Actor VM deallocates idle instances if they can't be reached (directly or indirectly through an arbitrary number of instances) by a non-idle instance. When the garbage collection criteria are met, the garbage collector creates a union of three sets: all non-idle instances, all instances referenced in messages pending in the message queue, and all instances registered with the machine for future services. This set is then repeatedly expanded until no new instances are added. This expansion combs through the registers of instances looking for additional instance references, which are added to the reachable list. Once the reachable list is complete, all idle instances that are not on the list are unreachable and are deallocated.

2.7.4 Broker

Since instance references can't be guessed, there must be a mechanism to discover instance references of service providers. The Actor VM uses a broker system where instances can register themselves with the broker using a text name. The broker can be used to look up a service provider using a text name. This simple system is intended to be an underlying mechanism that will empower a higher-level service brokerage protocol to be developed in Vole.

2.7.5 Resources

The machine is the source of resources available to the VM and is therefore the gateway to the resources of the real underlying machine. The VM machine class is given access to resources then the program is started. Only instances running in the VM can access those resources. Since references to those instances can be shared, an "object capability system" [11] can be realized.

2.8 Implementation

The Actor VM is written in Ada. Ada has been a standard part of GCC for a number of years and is widely (and freely) available on numerous platforms.

Memory management is primarily accomplished using reference-counting "smart pointer" classes.

Instructions and registers are not implemented as classes. These are at the heart of the VM and must be fast, memory efficient, and "definite" types (an Ada designation) so that they can be stored in arrays.

Except for asynchronous services, the entire VM executes in a single thread (Ada task) to avoid threading issues. Since this task must never block, all operations that potentially block must be implemented in an asynchronous service. The asynchronous services communicate with the main task through mailboxes implemented with Ada protected types. The main thread periodically polls the collected service mailboxes using a standard interface.

The code was developed in an "Aunit" unit test environment [16].

2.8.1 Future Work

One currently undeveloped area is reflection. The intent is to provide a number of machine traps to allow programs to query information from the machine at runtime. The machine trap mechanism queries a capability pool that stores registered objects with a capability interface (Ada interface, just like a Java interface). Adding new capabilities is as simple as coding them and registering them in the capability pool.

Reaching higher up the tree for tastier fruit, migration [13] would be an important capability to add.

2.9 Initial Results

The Vole compiler is crude but does allow for some exploration of the possibilities of the Actor VM. Vole test code has been developed to exercise the Actor VM in ways that are stressful or unworkable in Actor Foundry, Kilim, and Scala. This code has been used to help improve the VM by providing a challenge to be overcome. The stressors are as follows. The first is a message producer that runs much faster than its corresponding message consumer. This leads to a buildup of undelivered messages in the machine's queue. The second stressor is an actor that cycles in a tight loop without sending or receiving messages. The next stressor is a baton relay ring made up of a command-line specified number of actors. A baton is passed around the ring repeatedly, activating each actor in turn while the rest are idle. The last stressor is a wasteful implementation of the Fibonacci algorithm that is called in a loop with ever increasing values. This produces large numbers of instances that perform their function and then are forgotten.

3 Conclusion

The Actor Virtual Machine demonstrates that a VM can be specifically designed to support the Actor Model of programming. The Actor VM does not allow for operations that conflict with the Actor Model and thus provides the full safety of the Actor Model. In the Actor VM, this cannot be circumvented intentionally or accidentally unlike in systems that mix the Actor Model with other paradigms. The implementation of the Actor VM uses a unique register-based design that allows an actor's state to be encapsulated within the actor's instance while reducing copies of register arrays. Likewise, immutable tuples also reduce the need to make copies.

4 References

[1] Agha, Gul Abdulnabi. "Actors: a model of concurrent computation in distributed systems." (1985).
[2] Karmani, Rajesh K., and Gul Agha. "Actors." Encyclopedia of Parallel Computing. Springer US, 2011. 1-11.
[3] Karmani, Rajesh K., Amin Shali, and Gul Agha. "Actor frameworks for the JVM platform: a comparative analysis." Proceedings of the 7th International Conference on Principles and Practice of Programming in Java. ACM, 2009.
[4] Fowler, Martin. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, 2004.
[5] Meijer, Erik, and John Gough. "Technical overview of the common language runtime." language 29 (2001): 7.

[6] TMS 9900 Microprocessor Data Manual, Texas Instruments, 1976
[7] Website:
http://luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf
[8] Shi, Yunhe, et al. "Virtual Machine Showdown: Stack Versus Registers." (2005).
[9] Halstead Jr, Robert H. "Multilisp: A language for concurrent symbolic computation." ACM Transactions on Programming Languages and Systems (TOPLAS) 7.4 (1985): 501-538.
[10] Website: http://www.erlang-factory.com/upload/presentations/247/erlang_vm_1.pdf
[11] Miller, Mark S., Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. Technical Report SRL2003-02, Johns Hopkins University Systems Research Laboratory, 2003. http://www. erights. org/elib/capability/duals, 2003.
[12] Marr, Stefan, and Theo D'Hondt. "Identifying a unifying mechanism for the implementation of concurrency abstractions on multi-language virtual machines." *Objects, Models, Components, Patterns* (2012): 171-186.
[13] Imai, Shigeru, Thomas Chestna, and Carlos A. Varela. "Elastic Scalable Cloud Computing Using Application-Level Migration."
[14] Negara, Stas, Rajesh Kumar Karmani, and Gul Agha. "Inferring ownership transfer for efficient message passing." *16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), New York, NY, USA*. 2011.
[15] Srinivasan, Sriram. "A thread of one's own." *Workshop on New Horizons in Compilers*. Vol. 4. 2006.
[16] Website: http://libre.adacore.com/tools/aunit/
[17] Ehringer, David. "The dalvik virtual machine architecture." *Techn. report (March 2010)* (2010).
[18] Barnes, John. *Programming in Ada 2005 with CD (International Computer Science)*. Addison-Wesley Longman Publishing Co., Inc., 2006.

Appendix A

Messaging Instructions

The variations on ways to send messages were originally handled using flags in the instruction, but we changed to implementation to encode the variations in the instruction names.

Messages can be sent to another actor, to oneself, or (once implemented) to a message handler higher in the class hierarchy. Messages can expect a reply (calls) or "fire and forget" (sends). "Tail" messages end the current frame of execution and pass on the responsibility of any reply to the message receiver.

Instruction	Format	Note
Actor_Call	reply := args=>actor.message	This instruction required four register references to specify the arguments being sent, the actor reference they are being sent to, the message definition to be invoked, and the location of the future/reply.
Actor_Send	args=>actor.message	Send a message to an actor and disregard any reply.
Actor_Tail_Call	args=>actor.message	Send arguments to message at actor, replies (if any) go to the invoker of the calling frame.
Actor_Tail_Send	args=>actor.message	Send arguments to message at actor.
Self_Call	reply := args=>message	Effectively a gosub, suspending execution of the current frame and pushing a new frame.
Self_Send	args=>message	This posts a new message to the instance's queue without interrupting the current execution frame.
Self_Tail_Call	args=>message	Effectively a gosub without pushing a new frame onto the stack. The reply (if any) will go back to the original invoker.
Self_Tail_Send	args=>message	Effectively a gosub without pushing a new frame onto the stack.
Super_Call	reply := args=>message	Not implemented yet.
Super_Send	args=>message	Not implemented yet.
Super_Tail_Call	args=>message	Not implemented yet.
Super_Tail_Send	args=>message	Not implemented yet.

Control Flow Instructions

There are both absolute and relative jumps and conditional branches.

Instruction	Format	Note
Jump	destination	Absolute jump to a location defined in a register. This could be used to implement a jump table.
Jump_Abs	location	Jump to an absolute location value encoded in the instruction.
Jump_Rel	offset	Jump to a relative PC value encoded in the instruction.
Branch_Abs	condition, location	If the condition is True, jump to the absolute location.
Branch_Rel	condition, offset	If the condition is True, jump relative to the current location.
Branch_Neq	condition, offset	If the condition is False, jump to relative to the current location.

Data Manipulation Instructions

Both versions of the Peek instruction will block if they encounter a future instead of a tuple.

Instruction	Format	Note
Set	lhs := rhs	Simple transfer of data.
Fold	lhs := Fold(rhs)	Rhs is a tuple map that is used to fold a new lhs.
Peek	a := x[y]	Y is a register reference.
Peek_Immediate	lvalue := tuple[index]	Index is encoded in the instruction.
Compute	result := left action right	This is used for both comparison operations and mathematical operations.

Machine Control Instructions

Note that running out of instructions is an error that will terminate the VM.

Instruction	Format	Note
Terminate_Program	<no arguments>	The variant record that corresponds to a 64-bit zero value will decode as this instruction. This terminates the VM.
Halt_Actor	<no arguments>	This terminates the instance.
Stop_Frame	<no arguments>	This completes the frame without sending a response.
No_Op		No operation.
Emit	value	This supports the ancient debugging practice of printing out values to the console.
Assert	boolean	This terminates the VM if a non-True value is evaluated.
Reply	tuple	Sends the specified response value. A non-tuple value will automatically be wrapped into a single-element tuple.
Trap	result := selector arguments	This instruction provides a general mechanism to request Machine support. If the selector is an unsigned integer, the CPU handles the trap. If the selector is a string, the Machine handles the trap using the registered capability table.
New_Actor	result := arguments => actor	This instruction allocates a new actor instance, saving the actor reference in result. The arguments are then sent to the new actor's constructor message.