An extensible virtual machine architecture

Timothy L Harris

tim.harris@citrix.com

Citrix Systems (Cambridge) Ltd,
Poseidon House, Castle Park,
Cambridge, UK, CB3 0RD.
Tel: +44 1223 515010.
Fax: +44 1223 359779.

Computer Laboratory,
Pembroke Street,
Cambridge, UK, CB2 3QG.
Tel: +44 1223 334600.
Fax: +44 1223 334678.

October 3, 1999

Abstract

In this paper I present a new approach to designing virtual machines. I argue that the *monolithic* structure of existing systems leads to unnecessarily rigid high-level abstractions being imposed on applications. In contrast, I propose an *extensible* approach within which key parts of the virtual machine can be updated independently of one another. Furthermore, the proposed approach allows untrusted application code to replace low-level components of the virtual machine, such as the thread scheduler or memory allocation subsystem.

1 Introduction

Recent work in the operating systems and programming languages communities has discussed the common ground between virtual machines (VMs) and operating systems (OSs). Until now, this discussion has concentrated on supporting multiple applications over a single VM. For example, how should applications be protected from one another? [BH99] How should resource usage be accounted to applications [CvE98] and how can this accounting be handled fairly when applications are interacting? [PT98, Men99]

This paper describes a further area in which I believe the design and implementation of VMs can benefit from previous OS work. This area is *extensibility*. As I shall argue in Section 2, taking this approach in OS design has produced systems with improved performance and the flexibility to support a variety of workloads. I contend that an eXtensible Virtual Machine (XVM) can bring similar benefits: that applications can profit from lower-level access to the environment in which they are executing and that an XVM could form a suitable platform for executing applications written in diverse languages. As I shall outline in Section 3, I believe that this extensibility can be provided without sacrificing the run-time safety which users have come to expect from systems such as the Java Virtual Machine (JVM).

2 Extensibility

Extensibility, in the context of an operating system, refers to the ability of end users and application programmers to control aspects of the system that have traditionally been fixed. These often include, for example, how the system interacts with the devices to which it is attached, such as the policy through which disk blocks are allocated to files or the network protocols that are available.

The approach taken in *Microkernel* designs is to move sections of the OS out of the kernel and in to user-level server processes. This modular decomposition reduces the amount of code that executes with kernel privileges which, in turn, may help to isolate faults and promote reliability. Moreover, new services may be created that were not conceived by the system's original designers and these new services may be installed and configured without rebooting the machine.

Although Microkernels succeed in devolving control from kernel-mode to user-mode, they do not necessarily provide increased flexibility to 'normal' processes. For example, although the server process controlling a file system executes in an unprivileged mode, it may still need to use a restricted interface to interact with the kernel.

This shortcoming is addressed in the *Exokernel* [KEG+97] and *Nemesis* [LMB+96] operating systems. These promote the exposure of low-level access to the system's resources *to all applications*. The intention is that most applications will access these low-level interfaces through standard, but untrusted, libraries. However, the design enables programs that so wish to bypass this code and access the low-level interfaces directly.

In contrast, *SPIN* [BSP+95] takes a different approach by allowing applications to download extensions dynamically into the running OS kernel. Since these extensions execute as part of the kernel, they can access system services with low overheads. The integrity of the system is maintained by requiring that extensions are implemented in a safe language (based on Modula-3) and are compiled using a trusted compiler.

2.1 Extensible virtual machines

In the previous section I described a number of approaches to introducing extensibility in OSs. Although these designs differ in many respects there is nonetheless a common theme running through them: that providing low-level interfaces can improve the range of applications which may be supported effectively and, in addition, can enable application-specific optimizations. I believe that this idea is applicable to VM design: the *monolithic* structure of many VMs imposes fixed abstractions and fixed policies on the applications that are running over them. In contrast, an *extensible* design permits application programmers greater freedom to tailor the runtime environment to their requirements.

To give a particular example, modern JVM implementations use potentially-complex strategies to select when to compile bytecode to native code [Gri98]. As I have argued previously [Har98], this is an area in which applications with particular requirements may not be handled effectively by general-purpose heuristics. For example, interactive programs may want to limit the degradation on responsiveness that compilation may introduce. In this case a straightforward procedural interface is a practicable means of providing control over when the compiler operates.

As a further example, consider the management of the heap in a VM. Applications exhibit remarkably different and complex patterns of memory usage in terms of object-size distributions and the dependencies between allocations over different timescales [WJNB95]. These characteristics have been exploited to improve the performance of heap management by tailoring allocation policies to particular applications [Vo96]. However, current VMs do not provide control over how the heap is managed. This support need not prejudice the safety of the execution environment, so long as the proposed allocation policy can be checked to return appropriate blocks that are genuinely free, perhaps by implementing it over a machine that supports operations on *linear* objects as advocated by Baker [Bak95].

Equally, the garbage collection (GC) strategy can dramatically effect execution efficiency. For example, a collector for ML may maintain a *store log* of updates that have been made to heap-allocated objects during the current collection cycle [NO93]. Although appropriate for a mainly-functional language with infrequent updates to heap-allocated objects, it is inappropriate for an imperative object-orientated language in which updates are frequent. However, unlike memory allocation, it is unclear how to support untrusted GC algorithms safely. The stumbling block is that the system would, presumably, require some means of verifying that the GC is operating safely. Nevertheless, an XVM could provide a set of GC options and provide a mechanism for trusted users to install new algorithms.

Finally, the choice of primitive operations exposed in a VM tends to be biased towards a particular language. For example, the operations available in Java bytecode are heavily based on the needs of the Java programming language.

3 The Sceptre XVM architecture

The previous section presented the case for extensibility in various aspects of a VM. This section describes *Sceptre*, a prototype XVM that is designed to provide safe support for some of these areas.

3.1 Overview

Sceptre is designed for extensibility at a very fine granularity by providing control over the set of primitive operations in which applications are written. A component-based approach is taken, in which each component provides implementations for a number of operations. For example, Figure 1 shows a simple machine definition that introduces a new *ternary-add* operation, `tadd`, that adds the top three elements on the run-time stack. Two components are involved, `ternary-add-instance` that contains the skeleton of the definition of the new operation and `core-wrappers-instance` that provides implementations of binary arithmetic operations.

Components are instantiated from *modules*, each of which has an associated *implementation machine* over which the module's code is implemented. At the lowest level, modules are implemented over the `core-machine` that provides access to memory and a run-time stack. A module may additionally have a number of *ports* which are interfaces through which it exposes the operations that it defines, or through which it imports operations on which it depends. For example the `do-as-ternary` module has two ports, one on which it presents the ternary operation that is being defined and the other on which it requires a binary operation from which to derive the ternary operation. In the example, the `.link` directive connects the components together so that a ternary *add* operation is created. The `.alias_operation` directive provides a name by which a programmer writing for the `extended-machine` will access the `op` operation on the `ternary-op-port` port of the `ternary-add-instance` component.

```
.define_machine extended-machine, core-machine, {
   .load_interfaces binary-int-op, ternary-int-op;
   .load_modules do-as-ternary-module, core-wrappers;
   .instantiate do-as-ternary-module, ternary-add-instance;
   .instantiate core-wrappers-module, core-wrappers-instance;
   .link (core-wrapers-instance, binary-add-port),
         (ternary-add-instance, binary-op-port);
   .alias_operation ternary-add-instance, ternary-op-port, op, tadd;
}
```

Figure 1: The definition of a machine supporting a ternary addition operation. In addition to this file, the complete example includes definitions of the interfaces *binary-int-op* and *ternary-int-op* and a definition of the modules *do-as-ternary-module* and *core-wrappers*.

3.2 Verification

Space does not permit a complete, or formal, description of the verification process to be given. However, I shall attempt to provide an overview of the way that code is checked. There are two principal concerns: controlling the operations that are available to a programmer and verifying that modules correctly implement interfaces.

The first of these problems is addressed by restricting the implementation machines made available to untrusted modules. Consider, for example, a system that provides operations similar to those available in the JVM. A coarse decomposition of this could involve three machines: the *core-machine*, a *jvm-internal-machine* and a *jvm-abstraction-machine*. The *jvm-internal-machine* is implemented directly over the *core-machine* and is responsible for providing the central safety-critical abstractions of Java bytecode, such as checked access to fields. The *jvm-abstraction-machine* is implemented over the *jvm-internal-machine* and uses it to provide the familiar operations of a Java bytecode system. The intention is that an untrusted programmer could extend the system with new operations implemented over the *jvm-internal-machine*, but that they would not be able to access the *core-machine* directly.

This leaves the second problem of code verification. The intention is to provide a system that is sufficiently powerful to check various safety properties of an implementation rather than to provide extensive checking of program semantics. For example, it may be reasonable to require that the implementation of an operation to allocate a block of memory will return a fresh block that has not previously been allocated.

For each point in the code, a verification state (VS) is determined, containing facts that are known about the corresponding run-time state. The program is divided into basic blocks, each of which is associated with an initial VS (containing information known about the state before the execution of the block) and a final VS (containing information about the state after the execution of the block). The verifier maintains a *work list* of blocks whose final VS may need to be updated. The verifier iteratively removes a block from the work list and, starting from that block's initial VS, simulates the effects of each of the instructions. If the resulting VS differs from the block's previous final VS then the final VS is updated and the block's successors are added to the work list.

Each instruction has a verification-time effect (VTE) and a run-time effect (RTE). The verification-time effect defines how the VS is updated when the verifier considers the instruction. The run time effect defines the concrete implementation of the operation, as would be needed, for example, by a compiler or interpreter. The VS has three components: a simulated stack that models the state of the corresponding run-time stack, a verification-time stack that is used for temporary storage in the definition of an operation's VTE and a set of verification facts (VFs) which models the information known about the items on the run-time stack.

Both the RTE and the VTE are derived from the definition of an operation. Figure 2 shows an example from a module of integer arithmetic operations that establish, at verification time, the parity (`ODD`, `EVEN` or `UNKNOWN`) of the values that are being manipulated. This example is an 'increment' instruction, and so its VTE flips the parity of the item at the top of the simulated run time stack. By convention, operations prefixed by a tick (`'`) have only verification-time effects. The instructions before `/*1*/` read the parity of the item at the top of the run-time stack from the VFs. The instructions to `/*2*/` do the increment itself. The instructions to `/*3*/` establish the parity of the result. The remaining instructions add the new fact to the VS.

The 'tick' operations are limited to ensure that the verification process terminates: there are no looping constructs and the `'define` operation, which updates the VFs, does so by recording the least upper bound of the supplied value

```
.define_operation port(parity_ops), pinc, {
  'push_key HAS_PARITY; 'push_id  0; 'make_key 2; 'lookup; 'expand_val;
/* 1 */
  push int(1);
  add;
/* 2 */
  'dup;
  'if_precisely PARITY, EVEN, { 'pop; 'push_val PARITY, ODD; }, {
     'if_precisely PARITY, ODD, { 'push_val PARITY, EVEN; },
                                { 'push_val PARITY, UNKNOWN; }; };
/* 3 */
  'push_key HAS_PARITY; 'push_id  0; 'make_key 2; 'roll 2,1;
  'make_val 1; 'define;
};
```

Figure 2: An example operation definition taken from a module handling integer arithmetic and maintaining verification-time state about the parity (ODD, EVEN, or UNKNOWN) of the values being manipulated.

with the existing value in the VFs. Elsewhere in the module containing this `pinc` operation, the relationship between `ODD`, `EVEN` and `UNKNOWN` will be defined, with `ODD`<`UNKNOWN` and `EVEN`<`UNKNOWN`.

3.3 State of current implementation

A prototype implementation of Sceptre is currently in production. The example machine definition, presented in Figure 1, is taken directly from one of the regression tests, as is the example operation implementation shown in Figure 2. The current emphasis is on developing a number of component definitions that can be used to assess the approach taken and to provide concrete demonstrations of the benefits of extensibility available to applications.

4 Related work and conclusions

The *Vanilla* project at Trinity College, Dublin [Dob98] is a system in which parsers, type checkers and interpreters can be constructed from language fragments. Each of these components implements a language feature and describes how the feature is realized in concrete syntax, how it is represented in the abstract syntax tree, how it affects the assignment of types to program fragments and how the language feature should be implemented at run time. A *language definition file* identifies the components that need to be combined to construct the required language.

The University of Washington's *Kimera* project is described as a *distributed virtual machine* [SGB+98]. It is *distributed* in the sense that the functionality of a system such as the VM is decomposed into separate components such as the verifier, the execution service and the resource management service. This approach may increase the overall integrity of the system by containing the failure of individual components – for example by separating the address spaces within which the components operate. The manageability of the system may be enhanced – for example by requiring that a common verification service is used within a company, under the close control of the system administrators. Finally, the decomposition may enable gains in performance and scalability by performing resource-intensive tasks such as compilation and verification on dedicated machines.

Folliot *et al*'s *Dynamically configurable, multi-language execution platform* [BF98] is a flexible virtual-machine-based system that is designed to be able to execute programs written in any bytecoded language. An application is 'typed' with the name of an appropriate *VMLet* that describes how the bytecode implementation of the program can be converted into a language-neutral internal representation.

These systems, *Vanilla*, *Kimera* and *VMLets*, all address some of the concerns of flexibility and extensibility that I have discusses here. However, their approaches are more akin to a Microkernel design rather than systems such as Nemesis or an Exokernel that safely expose this flexibility to untrusted applications. For example, the components from which a language is developed in Vanilla must be designed so that they do not conflict with one another and so that one component does not destroy invariants or security properties on which another depends. Similarly, although the decomposition proposed in Kimera allows a choice over where parts of the VM are implemented, these decisions need to be taken by the system administrator rather than on a per-application basis.

In contrast, this paper has presented the case for an eXtensible Virtual Machine, or XVM, in which lower-level interfaces within the system can be exposed safely to untrusted applications. In doing so, I believe that the VM will be able to support an increased diversity of programming languages and also enable application-specific optimizations.

References

[Bak95] H. G. Baker. 'Use-once' variables and linear objects-storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, 1995.

[BF98] F. Riccardi B. Folliot, I. Piumarta. A dynamically configurable, multi-language execution platform. In *Eighth ACM SIGOPS European Workshop*, September 1998.

[BH99] Godmar Back and Wilson Hsieh. Drawing the red line in Java. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems (HotOS-VII)*, 1999.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating Systems Principles*, pages 267–284, Copper Mountain, Colorado, December 1995.

[CvE98] Grzegorz Czajkowski and Thorsten von Eicken. JRes: A resource accounting interface for Java. *ACM SIGPLAN Notices*, 33(10):21–35, October 1998.

[Dob98] Simon Dobson. A first taste of Vanilla. Technical Report Trinity College, University of Dublin, TCD-CS-1998-20, September 1998.

[Gri98] David Griswold. The Java HotSpot virtual machine architecture. March 1998. `http://www.javasoft.com/products/hotspot/whitepaper.html`.

[Har98] Timothy Harris. Controlling run-time compilation. In *Procedings of the IEEE Workshop on Programming Languages for Real-Time Industrial Applications*, pages 75–84, December 1998.

[KEG+97] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th Symposium on Operating Systems Principles (SOSP-97)*, volume 31 of *ACM Operating Systems Review*, pages 52–65, October 1997.

[LMB+96] I. M. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The Design and Implementation of an Operating System to Support Distributed Multimedia Applications. *IEEE Journal on Selected Areas In Communications*, 14(7):1280–1297, September 1996.

[Men99] Paul Menage. RCANE: a resource controlled framework for active network services. In *Proceedings of the First International Working Conference on Active Networks – IWAN '99*, June 1999.

[NO93] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28 of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993.

[PT98] J. Lepreau P. Tullmann. Nested Java processes: OS structure for mobile code. In *Eighth ACM SIGOPS European Workshop*, September 1998.

[SGB+98] Emin Gun Sirer, Robert Grimm, Brian N Bershad, Arthur J Gregory, and Sean McDirmid. Distributed virtual machines: A system architecture for network computing. In *Eighth ACM SIGOPS European Workshop*, September 1998.

[Vo96] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software — Practice and Experience*, 26(3):357–374, March 1996.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *1995 International Workshop on Memory Management*, Kinross, Scotland, UK, 1995. Springer Verlag LNCS.