An highly-configurable, modular system architecture for mobility, interoperability, specialisation and reuse

Carine Baillarguet and Ian Piumarta

INRIA Rocquencourt
B.P. 105, 78153 Le Chesnay Cedex, France
{Carine.Baillarguet, Ian.Piumarta}@inria.fr
http://www-sor.inria.fr/projects/vvm/

Abstract. Current systems and architectures in the domains of distributed and embedded computing are complex, respond badly to many application needs, and are not easily specialisable to better support a given application or host platform. This leads to a proliferation of non-reusable *ad-hoc* solutions. Our goal is to develop a flexible, object-oriented *virtual execution environment* (VEE), dynamically specialisable according to both the needs of a given application domain and the capabilities/peculiarities of a given host platform. A highly-configurable VEE offers immense benefits for many kinds of embedded and/or distributed system, at all stages from initial design through to final deployment.

1 Introduction

Specialisation and reconfigurability are becoming critical characteristics of operating systems [Cah96]. At the same time, the traditional distinction between operating system and language is fast disappearing [HM98] since the solutions to many difficult systems problems can be addressed most effectively using language techniques [PW93]. This paper describes the architecture and framework of the *adaptive virtual machine* (AVM), a highly modular object-oriented system architecture integrating both operating system and language execution services, which offers solutions to many of the problems of embedded systems and mobility-based distributed systems.

Application architecture, deployment and utilisation are constantly changing due to evolving technologies such as the Internet and the embedding of computing power in increasingly remote contexts. Such applications today range from scientific satellites and active networks to virtual worlds and agent-based distributed systems.

These kinds of applications have particular and diverse needs, including mobility, adaptability, extensibility and dynamicity. *Mobility* imposes constraints such as compactness of code, portability, interoperability and security. *Adaptability* and *extensibility* are needed if the environment is to be shaped according to a specific application or the hardware on which it runs. Together these two properties can assure software homogeneity (avoiding porting problems between platforms) and "optimal" conditions of execution (which might be different between low-end and high-end versions of the same product). Both are kinds of specialisation, and each can be either static (chosen before deployment) or dynamic (chosen after/during deployment). *Dynamicity* enables systems to offer rapid and reversible modification of the application's environment, as well as the ability to evolve "in the field" systems that would otherwise be difficult to maintain — either because of sheer numbers (e.g. set-top boxes) or severe physical constraints (e.g. satellites and other telemetric systems where space and/or communication bandwidth can be severely limited).

Recent systems have addressed some of these needs by removing the distinction between the application programming language and the operating system upon which it runs [HM98]. Traditional system services (threads, persistence, etc.) can be better utilised and controlled when they are pulled into the application language. The core services of μ/exo-kernels and similar flexible operating systems [BSP+95,EKO97,FHL+96] often rely on the characteristics of a particular programming language (provability, behavioural verification, etc.). The relationship is even closer in highly-reflexive systems [Yok92] which use a tightly-integrated metaobject protocol for behavioural customisation.

This symbiosis helps with adaptability, extensibility and customisability, but does little to help with mobility, dynamicity and interoperability. Virtual machines offer an excellent solution to these problems, decoupling the executable application from the underlying hardware. This maximises mobility (small, highly-portable code), allows specialisation (customised instructions and primitives), eases the implementation of a tightly-integrated MOP (for introspection and intercession), and facilitates runtime behavioural verification. Efficiency can be assured by including the final stage of compilation as an OS service (dynamic code generation) [Fra97].

Merging these two approaches (system/language integration and virtual machine based execution) yields what we call a *virtual execution environment* (VEE). The first such systems are starting to appear, in the form of JanOS for example [JANOS]. The major remaining problem is that the system language is *general* and *fixed*, precluding per-domain specialisation which is often essential.

The next section describes the AVM — a dynamic, highly-specialisable virtual execution environment that provides tightly-integrated language and operating system services. It includes a virtual machine that can be specialisable for different languages/applications, and extensible runtime support that can be specialised for different applications/host platforms.

2 A highly specialisable VEE: the AVM

The AVM addresses the above problems of specialisation using a combination of "offline" (static) and "online" (dynamic) components that together provide a complete and highly configurable VEE (Figure 1).

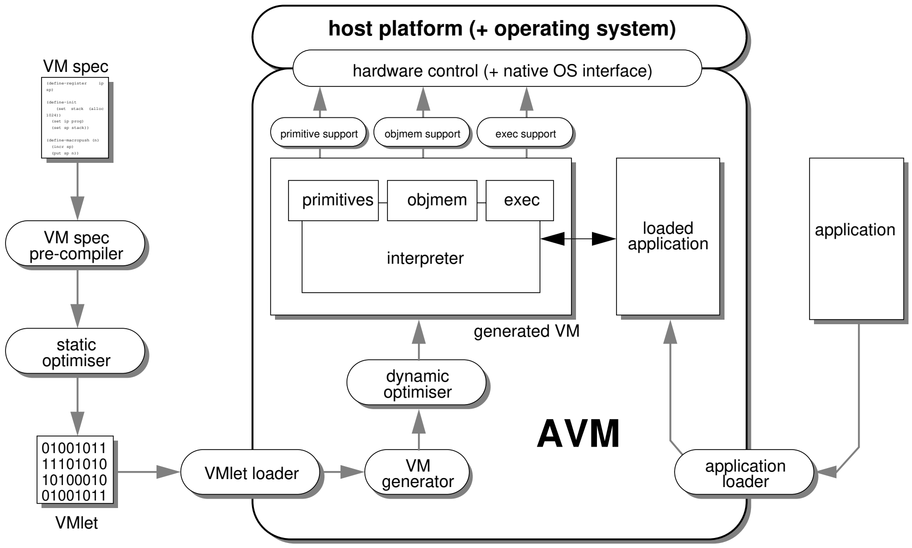

Fig. 1. VM specs are high-level descriptions of the components that make up a VEE. Each spec includes definitions of **execution** and **object** models, and the language **primitives** connecting the application to the system. The **VM spec compiler** and **pre-optimiser** transform these into compact structured VMlets. The **VMlet loader** decodes VMlet surface structure and passes the content to the **VM generator** which dynamically instantiates the corresponding VM and operating system services.

Figure 2 shows the object-oriented framework that supports the high degree of modularity within the AVM.

Each component of the AVM is implemented as a coarse-grain object with a well-defined interface (similar in principle to Spring [HK93]). In Figure 2, **VMlet(L)** implements the VM for a language **L**. The corresponding VM is instantiated via a standard interface to the AVM code generator (GenerateAbs). The code generator in turn uses a standard interface to a pluggable back-end (Gen(thread) or Gen(native)). Back-ends can provide different tradeoffs between space and speed by implementing a pure interpreter for the VMlet bytecodes, translation to threaded code [Bel73], translation to native code, and so on. Different levels of optimisation are also possible within each back-end (again trading space for speed), and each may have one or more pluggable optimisers sharing a standard interface (e.g NativeOpt(space) or NativeOpt(speed)).

The VMlet also maps the application's object format onto a shared generic object memory (ObjMemAbs). This can be replaced with alternative implementations to maximise speed/minimise space, or to change the garbage collection policy, according to application and/or host platform constraints.

Application execution is effected through a standard interface to the AVM Executive. The VMlet, object memory and Executive all access operating system services through a standard "virtual operating system" interface (PrimAbs).

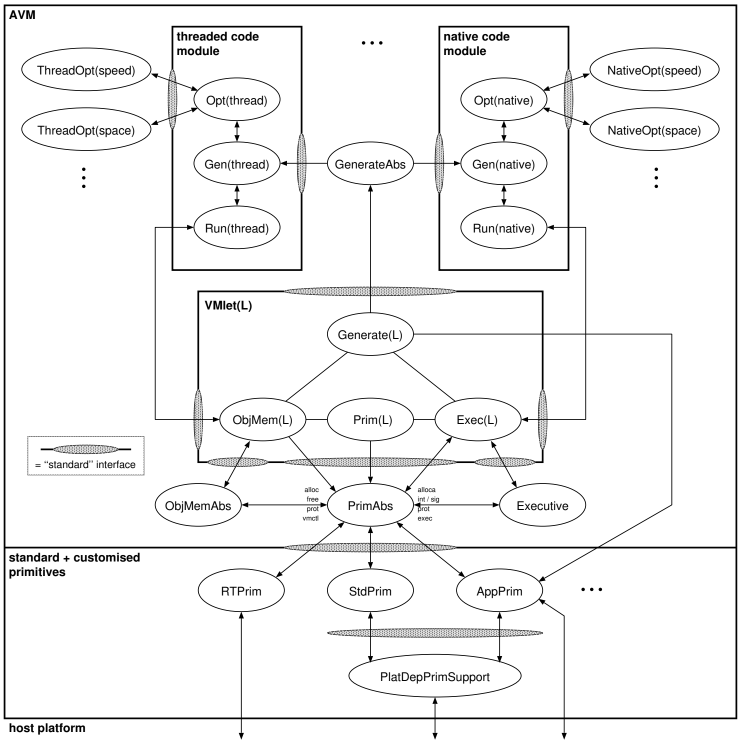

Fig. 2. The AVM framework. (See text for details.)

This abstract set of primitive OS services hides the VMlet, object memory and Executive from platform specifics.

The virtual operating system is implemented in terms of a set of standard OS services (StdPrim), which might be provided by a local microkernel or implemented directly on top of raw hardware. This set of services is pluggable, according to the local host platform. It typically provides access to libraries of "native methods", file system, process control, memory configuration, dynamically-loaded libraries, communications, device drivers, interrupt control, and so on.

VMlets also have the potential to extend the low-level system services (StdPrim) with their own domain-specific services (AppPrim). These are necessarily dependent on the host platform, for example device drivers specific to a particular scientific instrument, data bus, or communications medium.

3 Conclusions

The AVM's object-oriented framework supports the needs of many emerging applications, from two perspectives: extensible operating system and platform-independent execution. The architecture emphasizes adaptability, reflexivity, reusability, interoperability, configurability, ease of maintenance, and performance tuning for a given application and deployment environment. The modular approach allows completely independent specialisation of the supported application language, its execution mechanism, and the primitive system services. Most components within the AVM can be replaced freely and independently.

Many degrees of static specialisation are possible. External connectivity across different hosts (network routers, set-top boxes, consumer communications equipment, etc.) can be abstracted away from applications. Execution mechanisms are interchangable to choose different speed/space tradeoffs and resource requirements. A high-level VM specification language supports rapid modification of standard VMlets for domain-specific execution and/or object models.

A high degree of reusability is assured by the use of "off-the-shelf" VMlet libraries implementing common functionality: execution submodels, garbage collection, arbitrary-precision arithmetic, data structures, caches, communications protocols, process control, hardware management, etc.

Dynamic specialisation is most useful for modifying the bytecode and primitive sets for a given application. This is essential for tuning performance tradeoffs during different phases of deployment, incrementally updating the execution and system environments in parallel with evolving protocols/standards, and changing the set of system services that are available to the application.

References

[Bel73] James R. Bell, *Threaded Code*, Communications of the ACM, 16(6):370–372, 1973.

[BSP+95] B. N. Bershad, S. Savage, P. Pardyak, and al, *Extensibility, safety and performance in the SPIN operating system*, SOSP, 1995.

[Cah96] V. Cahill, *Flexibility in object-oriented operating systems: a review*, Technical Report, TCD-CS-96-05, University of Dublin, 1996. `http://www.cs.tcd.ie/publications/tech-reports/`

[EKO97] D. R. Engler, M. F. Kaashoek, J. O'Toole Jr, *Exokernel: an operating system architecture for application level resource management*, HOTOS VI, 1997.

[FHL+96] B. Ford, M. Hibler, J. Lepreau and al, *Microkernels meet recursive virtual machines*, OSDI'96, 1996.

[Fra97] M. Franz, *Run-Time Code Generation as a Central System Service*, Proc. HotOS-VI,May 1997. `http://www.ics.uci.edu/~franz/publications/`

[HK93] G. Hamilton and P. Kougiouris, *The Spring Nucleus: A Microkernel for Objects*, Technical Report TR–93–14, Sun Microsystems Laboratories, April 1993.

[HM98] J. Howell and M. Montague, *Hey, You Got Your Language In My Operating System!*, Technical Report PCS-TR98-340, Department of Computer Science, Dartmouth College, 1998. `http://www.cs.dartmouth.edu/~jonh/research/`

[JANOS] Flux Research Group, *JanOS: a Java-based active networked operating system*. `http://www.cs.utah.edu/projects/flux/janos/summary.html`

[PW93] C. Pu and J. Walpole, *A Study of Dynamic Optimization Techniques: Lessons and Directions in Kernel Design*, Technical Report OGI-CSE-93-007. `http://www.cse.ogi.edu:80/DISC/projects/synthetix/`

[Yok92] Y. Yokote, *The Apertos Reflective Operating System: The Concept and Its Implementation*, Proc. OOPSLA 1992.