A Dynamically Configurable, Multi-Language Execution Platform

Bertil Folliot[*], Ian Piumarta, Fabio Riccardi

SOR, INRIA Roquencourt, B.P. 105,
78153 Le Chesnay Cedex, France

email: {bertil,ian,fabio}@sor.inria.fr

Abstract

Writing efficient applications is difficult for two reasons. A given problem domain often demands a particular programming language and execution environment. Together these provide a "paradigm" in which the programmer can concentrate on the problem rather than the implementation. Secondly, most operating systems have a rigid architecture. Adapting them to the needs of modern applications, such as cooperative work, embedded systems, and multimedia, requires *ad-hoc* solutions, increasing development costs and reducing both reliability and efficiency.

We present a multi-language, hardware independent, execution platform, that is dynamically extensible and adaptable to each application's needs. An application is associated with a virtual machine decription, that allows to translate the application into a language neutral internal representation. The novel virtual machine architecture described in this paper allows programs written in an arbitrary bytecoded language to be executed safely and efficiently. Its single execution model provides interoperability, and effectively promotes reuse of existing software modules.

1 Introduction

Applications are becoming larger, clumsier, and buggier. Notable examples are Netscape and Microsoft Office, which contain their own implementations of typical operating-system services: distribution, QoS, security, and so on. Such services belong in the operating system.

[*]Also with the LIP6, Paris.

Development environments contribute further to the problem by offering too limited a choice of languages and programming paradigms, seriously reducing the programmer's power of expression.

Most modern distributed applications – such as cooperative work, embedded systems, multimedia applications, and mobile computing – do not use distribution to increase performance. They use it to deal with the geographical distribution of users and resources, and with security concerns such as data sensitivity, ownership, access rights, and copyright. Such applications are usually characterised by complex data models, complex transactions, and dynamic (and randomly variable) configurations of heterogeneous interacting parties.

Dealing with heterogeneity raises severe obstacles to interoperability. Java [AG96] is a meager step in the right direction, but it is still far too rigid. It uses bytecodes as its distribution language, implying a standard virtual machine that cannot be modified to fit particular applications. Moreover, a single language (and execution model) cannot address every problem domain: creating Web applets, scripting a text editor, and programming a smart card reader or mobile host support station, are domains in which the effective tools are very different. Nevertheless, these applications should ideally be executed in the same environment. This reduces resource consumption, promotes integration and component reuse (internal interoperability). Such an environment must be configurable for new languages, code and data formats (external interoperability).

The remainder of this paper presents a novel virtual machine architecture, the *virtual virtual machine* [FPR97]. The VVM is a multi-language, hardware independent execution platform that allows programs written in an arbitrary bytecoded language to be executed safely and efficiently. The VVM is dynamically extensible and adaptable to each application's needs by loading/unloading at run time virtual machine description, called VMlets. A VMlet contains a reconfigurable "machine model", a single, highly generic "memory model" and a comprehensive operating system, the "environment model". Applications can then be written using the most appropriate language, and can customise their execution environment to meet their specific needs. Many application areas benefit from this approach and mostly modern distributed applications.

2 The Virtual Virtual Machine

Virtual execution environments and higher-level programming languages effectively improve the quality of software while reducing development costs. Their acceptance has, however, been limited because virtual machines were traditionally considered to be slow. This objection is becoming increasingly irrelevant with each new generation of high-speed processors.

A relevant objection remains in the rigidity of virtual machine environments. Our *virtual* virtual machine can be configured to execute applications written in any bytecoded language. It renders the VM environment flexible, removing this final objection.

The key point in the flexibility of the VVM is the capability to describe vir-

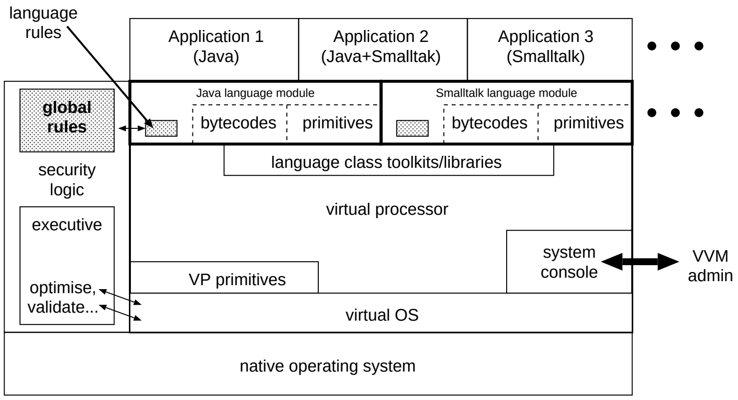

Figure 1: Figure 1: VVM architecture.

tual machines (called VMlet) and to associate one VMlet with each application. A VMlet takes every aspect of a traditional virtual machine and make all of them virtual (i.e. reconfigurable):

- machine model: single, extensible virtual processor, "soft" instruction set, and support many idealised machine models;
- memory model: single object memory implementation, multiple object semantics, and full range of services (from GC to finalisation);
- environment model: full range of operating system features, and subset of services that maps onto language's primitive set.

The key point in the dynamicity of the VVM is the transparent loading and unloading of these VMlets at runtime. Applications are "typed" with an appropriate VMlet (in accordance with some extended Web standards [BC94]). Whenever a new application type is encountered, the associated VMlet is loaded. The virtual machine decription contained in the VMlet allows to translate the application into the internal representations (language neutral) used by the VVM's execution mechanism.

Internal interoperability (reuse of components implemented in different languages) is facilitated by the translation into this single executable representation.

2.1 The VVM Architecture

The VVM architecture is mainly composed by a virtual processor, a virtual operating system interface, language toolkits, VMlets, and a security module. A simplified diagram is shown in Figure 1.

Virtual Processor. The core machine is the virtual processor, which provides a low-level execution engine. This engine has a simple intrinsic objet model (and corresponding object memory), instruction set and execution model. It uses a language-neutral internal representation that offers good performance, a significantly smaller footprint than dynamically-translated native code, and a high degree of portability. The intrinsic VP instructions are elementary. They provide the building blocks required to implement a wide variety of bytecoded virtual machines. The VP interacts with the virtual operating system (VOS), which provides an abstraction over the native operating system's facilities.

Virtual Operating System. VOS services are obtained by VP programs by invoking virtual processor primitives. For example, the virtual operating system (VOS) is in charge of handling the VP scheduling and of administering protected resources. It is also responsible for dealing with reconfiguration when a new VMlet or toolkit is loaded.

Language toolkits. These are provided in the form of instruction set libraries. These provide functions common to a family of languages, in the form of new VP instructions defined in terms of existing VP instructions (in the same way that each VMlet defines a new VP instruction for each of its bytecodes). Typical VP libraries would provide cloning, class objects with instantiation, delegation or inheritance mechanisms, and special data representations such as tagged "immediate" types.

VMlets. Support for each specific language within a family is provided by a VMlet. Each VMlet includes five things: mappings from the VM bytecodes to the VVM's execution mechanism and from the VM objects to the VVM's object format, implementations for the VM's primitives, a loader for the VM's applications, and the type checking specification and dynamic security rules for a given language. Each VM bytecode maps onto a single, possibly new, VP instruction. These new VP instructions are defined when the VMlet is loaded, in terms of existing VP instructions.

Security module. Configurability brings with it problems of security. Configurable operating systems such as SPIN ensure that extensions incorporated for a given application will not affect other applications. Security cannot be ensured with static type interface verification alone, but must also be checked dynamically at run time. The VOS cooperates with the security module to perform the "management" tasks that are required to ensure integrity of operation. The global (inter-language) aspects of these tasks are described by a set of rules in the security module itself. Rules relating to individual languages are provided by each VMlet. These rules describe the security constraints for applications running on top of that VM, and how the VP is allowed to manipulate the objects belonging to this VM.

Application programs. The application bytecodes of each language are dynamically translated into the VP's executable instruction representation following the rules in the associated VMlet. Since all applications are translated into a single executable format (regardless of which VM defines their semantics), a single execution mechanism (defined by the VVM) is used to run the code.

2.2 VVM operation

This section summarises the operation of the VVM, from the arrival of an application to its execution.

When an application arrives at the VVM, its "type" is checked. If the VVM has not already loaded the appropriate VMlet, then it is located and loaded. The security module verifies the VMlet according to the global rules, to ensure that the integrity of the VVM is not compromised. The VP's instruction set is extended with operations defined either by the VMlet itself or imported from one or more VP libraries containing common operations for the class of languages to which the VM belongs.

The application is then loaded. Language-specific security rules, defined by the VMlet, are applied to the application's bytecodes and data as appropriate. The application's data are then translated into the VVM's internal representation.

Application execution can now begin. Methods (procedures, member functions, or whatever else the VM might call its units of executable code) are dynamically translated into the corresponding VP instructions for execution. The VMlet provides primitives for its applications which are accessed in the manner appropriate for the language. (Specially marked methods, "primitive" bytecodes, etc.) These will eventually invoke VOS services and intrinsic primitives defined by the VP.

3 Related Work

Our work makes contributions to both operating system and language research.

On the systems side, we compare with SPIN [PB95], ExoKernel [KEG+97], and the FLUX OSkit [FBB+97]. These are reconfigurable and customisable architectures in which application programs are able to safely and dynamically add new functionalities to the OS to satisfy their specific needs. For example, SPIN permits safe changes to be made to the operating system's interface and implementation. The VVM provides an extensible platform for the execution of bytecoded applications, in which each VMlet can introduce language-specific primitives and system services into the VP.

On the language side, we can compare to the Universal Virtual Machine announced by IBM and Taligent [IBM97], which is capable of executing programs written in Java, Smalltalk and Visual Basic. Internal interoperability permits inter-language reuse of components, and external interoperability allows "applets" built in any of these languages to be downloaded and executed. The

UVM is, however, a simple extension to a static (Smalltalk) virtual machine. It cannot be extended dynamically to support arbitrary bytecoded languages.

The VVM, in contrast, supports dynamic extension to the range of supported languages; it transposes the philosophy of configurable operating systems to the domain of bytecoded language execution. The same kind of "philosophy" is being applied to network with the works on Active Networks [AAH+98].

4 Conclusions

We propose a novel execution environment, the Virtual Virtual Machine. The important properties of the VVM are dynamic extensibility and language interoperability. These properties provide a maximum of support for many emerging application domains.

External interoperability facilitates the exchange of functionality as well as the data on which it operates. This is important for embedded systems, mobile computing, and cooperative applications.

Internal interoperability allows applications to exchange or manipulate data between components written in different languages and compiled into different bytecoded representations. This promotes reuse of components between languages, allowing the components to be implemented in whichever language most closely matches their problem domain.

Adaptability is essential for long-lived and possibly unreachable applications that nevertheless require periodic maintenance. The cost of such maintenance is significantly reduced when it can be performed in the field. Systems embedded in set-top boxes and network routers can benefit from this potential.

Reuse is promoted at several levels: VM implementation, application components, and operating system services.

Extensibility is provided not only for application components, but in the mechanisms used to execute them. Embedded systems can again benefit from an execution environment that is easily updated, specialised for each application, or dynamically reconfigured depending on the needs of the host system at any particular moment.

We are in the process of implementing a first version of the VVM, executing a couple of modest virtual machines. Some of the enabling technologies required for the VP have been investigated and show promising results [PR98]. An early prototype of the virtual processor executed a numerical program at 70% the speed of the same program written in C. More realistic experiments have been conducted in the context of the Objective Caml VM [LW93], which confirm this result over a wide range of benchmarks.

References

[AAH+98] D. Alexander, W. Arbaugh, M. Hicks, P. Kakkar, A. Keromytis, J. Moore, C. Gunter, S. Nettles and J. Smith, *The SwitchWare Active Net-*

work Architecture, to appear in IEEE Network Special Issue on Active and Controllable Networks, 1998.

[AG96] K. Arnold and J. Gosling, *The Java Programming Language*, Addison Wesley, 1996.

[BC94] T. Berners-Lee and D. Connolly, *HyperText Markup Language Specification – 2.0*, 28 November 1994. `ftp://ds.internic.net/internet-drafts/draft-ietf-html-spec-00.txt`

[FBB+97] B. Ford, G. Bac, G. Benson, J. Lepreau, A. Lin and O. Shivers, *The FLUX OSkit: a substrate for kernel and language research*, in Proc. 16th ACM Symposium on OS Principles SOSP-16, October 97, St. Malo, France.

[FPR97] B. Folliot, I. Piumarta and F. Riccardi, *Virtual Virtual Machines*. Cabernet Radicals Workshop, Crete, September 1997.

[IBM97] IBM plans cross-platform competitor to Java, InfoWorld Electronic, April 1997.

[KEG+97] M. Kaashoek, D. Engler, G. Ganger, H. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. MacKenzie, *Application Performance and Flexibility in Exokernel Systems*, in Proc. 16th ACM Symposium on OS principles SOSP-16, October 97, St. Malo, France.

[LW93] X. Leroy and P. Weis, *Manuel de Référence du langage Caml*, InterEditions, Paris, 1993, ISBN 2-7296-0492-8

[PB95] P. Pardyak and B. Bershad, *Extensibility, safety and performance in the SPIN Operating System*, in Proc. 15th ACM Symposium on OS Principles SOSP-15, Cooper mountain, Colorado, December 1995, pp. 267–284.

[PR98] I. Piumarta and F. Riccardi, *Optimising direct-threaded code by selective inlining*, in Proc. Symposium on Programming Language Design and Implementation PLDI'98, Montreal, Canada, June 1998.