Beyond Flexibility and Reflection: The Virtual Virtual Machine Approach

B. Folliot[2], I. Piumarta[2], L. Seinturier[2], C. Baillarguet[1], C. Khoury[2], A. Leger[2], and F. Ogel[1]

[1] INRIA Rocquencourt, Domaine de Voluceau
78153 Le Chesnay, France
[2] Laboratoire d'Informatique Paris VI, Université Pierre et Marie Curie,
4, place Jussieu, 75252 Paris Cedex 05, France
firstname.name@inria.fr,
http://www-sor.inria.fr/projects/vvm

Abstract. With todays wide acceptance of distributed computing, a rapidly growing number of application domains are emerging, leading to a growing number of ad-hoc solutions, rigid and poorly interoperable. Our response to this challenge is a platform for building flexible and interoperable execution environments (including language and system aspects) called the Virtual Virtual Machine. This paper presents our approach, the first two realisations and their applications to active networks and flexible web caching.

1 Introduction

As distributed computing becomes widely spread, new application domains are emerging rapidly, introducing more and more heterogeneity into distributed environments: firstly, because of the rapid hardware evolution (especially with embedded devices) and secondly because each new application domain comes with it's own semantic, constraints and therefore set of dedicated abstractions.

To face those heterogeneity issues, current approaches lead to the design of complete, new, dedicated programming/execution environment, including language and operating systems aspects. As a result, programming and execution environments, while being well adapted to some given application domains and/or hardware, remains static, rigid and poorly interoperable..

Our response to this problem is a new, systematic approach for software adaptation and reconfiguration based on a language and hardware independent execution platform, called the Virtual Virtual Machine (VVM)[6].

The VVM provides both a programming and an exection environment, whose objectives are (i) to allow the adaptation of language and system aspects according to a specific application domain, such as smart cards, satellites or clusters; (ii) to achieve dynamic extensibility, by changing "on the fly" the execution environment (adding protocols, hardware support, algorithms or even "bug correction"); (iii) to provide a common language substrate on wich to achieve interoperability between different languages/application domains.

D. Grigoras et al. (Eds.): IWCC 2001, LNCS 2326, pp. 16–25, 2002.

The remainder of this paper starts by presenting the *VVM* approach and architecture in Section 2. The first two prototypes and their applications are described in Section 3 and 4 respectively. Section 5 describes an example of application in the domain of flexible web caching. Related works appear in Section 6, followed by conclusions and perspectives in Section 7.

2 Virtual Virtual Machine Project

Most modern distributed applications or environments are composed of complex and heterogeneous interacting components. Dealing with this heterogeneity raises severe obstacles to interoperability.

The virtual machine approach is a step in the right direction, allowing inter-systems interoperability, portability and promoting mobility/distribution with a compact code's representation and security mechanisms. But there are still dedicated to specific application domains. Let's consider SUN's Java Virtual Machine: it corresponds to an application domain where there is high amount of available memory, limited acces to the underlying system and no quality of service.

The apparition of new application domains, with different characteristics, implies new virtual machines to match new requirements (for a given architecture, as JavaCard for smartcards or KVM for mobile phones, or some software needs like real time (RT Java) or fault tolerance). This proliferation of "ad-hoc" virtual machines breaks the interoperability capabilities of the approach.

If virtual machines are a good step but are still far too rigid, why not to "virtualize" them. Hence, instead of developping a new virtual machine for each new application domain, a specification is dynamically loaded into the *VVM*. This specification describes a virtual machine adapted to this application domain.

The main goal of this architecture, is to bring adaptation, flexibility, dynamicity and interoperability to applications, without sacrifying the performances.

Figure 1 gives a simplified vision of this architecture. The *VVM* runs on top of an existing OS or on bare hardware. The lower layer (called μVM) represents the OS and hardware dependent part. On top of it is the core of the *VVM*:(i) the *Virtual Processor*, wich provide a low-level execution engine based on a language-neutral internal representation and elementary instructions; (ii) an *object memory* (and the associated pure object model), with garbage collection;(iii) some *input methods* that allow dynamic loading of execution environment specification.

Such a specification is called a *VMLet*. It is a high level description of the language and/or the system aspects of a programming/execution environment dedicated to a given application domain. Because of having a single execution mechanism for all the *VMLets*, it promotes interoperability (and reuse of code) between applications but also between application domains (and their respective environments). This interoperability can range from simple data exchange to mobile computations. It allows the sharing of physical and/or logical ressources and permits aggressive optimizations.

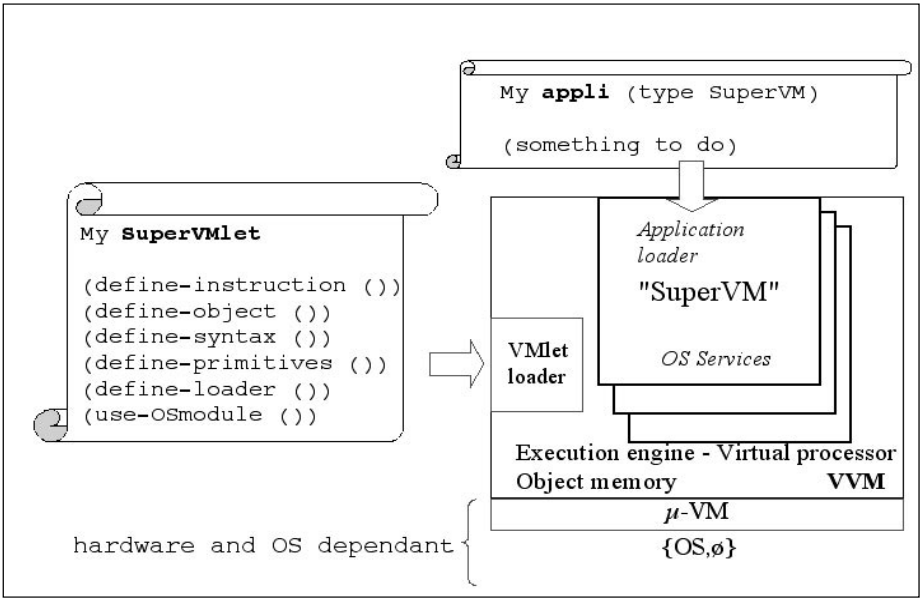

Fig. 1. The *VVM* architecture.

Once the *VVM* has loaded a *VMLet*, it adapts/extends itself so that it contains the semantic and functionnalities of the described virtual machine. Hence there's only one meta-level: the *VVM becomes* the VM described in the *VMLet*, thus the performances should be equivalent to the similar "hand-coded" virtual machine. Then, applications written for this environment can be loaded and executed as if they were running on a dedicated VM.

Dynamic extensibility comes from the ability to add and/or redefine "on the fly" everything in the environment, in response to requirements or execution conditions change. Interoperability can be achieved by having a single execution engine, and thus a common language substrate that can be used to exchange data and code between applications and/or *VMLets*.

3 The Reflexive Virtual Machine

The first step toward the *VVM* was the Reflexive Virtual Machine (*RVM*). It is a scheme-like interactive virtual machine that is able to modify dynamically its own primitives and instructions sets.

This dynamic flexibility provides the *RVM* with the ability to adapt/extend itself at runtime and thus turns itself into a virtual machine dedicated to a given application domain, without loosing its flexibility.

Not only can the language of the virtual machine be extended at runtime by adding instructions or primitives, but it can also be adapted to some domain-specific semantics, by adding user-level extension from classes and inheritance (which do not exist in traditional lisp-like languages) to semaphores or lightweight processes.

Although quite limited, the *RVM* has been used to experiment *VMLets* programming, and especially in the context of active networks. Active networks represent an emerging application domain that is therefore, as we mentioned, addressed via lots of differents and dedicated solutions, without any interoperability between.

From the dozens of existing protocols, we quoted two: *PLAN* [9] and *ANTS* [21]. When *PLAN* rely on packets containing both data and code, *ANTS* uses a *deployment* phase. During this phase, the protocols are sent to the routers with a *protocol id*, after what only data and the id of the protocol to be used needs to be sent. Each of those protocols represents an extreme on the full range of possible active network protocols.

So we defined two *VMLets* (one for each protocol), including language/operating system aspects and an API. We keep the front-end lisp-like language of the *RVM* and reuse the socket's services of the underlying UNIX OS (select, send, receive,...). Concerning the APIs, we mimic *PLAN* and *ANTS* 'ones. Thus, by loading such a *VMLets*, the *RVM* transforms itself to an active router (that understand *PLAN* and/or *ANTS*, depending on what is loaded).

As a first result, each *VMLet* is two order of magnitude smaller than the corresponding original implementation[1]. By simply loading the two *VMLets*, we obtain an active router that is able to proceed both *PLAN* and *ANTS* 's packets.

The next logical step is to define a generalisation of the active network application domain, called Active Active Networks, that will allow (i) to select the most appropriate protocol, according to some requirements, at any time; (ii) a dynamic deployment of any active network protocols, giving us an opportunity to explore the different possible strategies between *PLAN* and *ANTS*.

4 YNVM Is Not a VM

The current prototype, called the *YNVM*, is a dynamic code generator that provides both a complete, reflexive language, and an execution environment. The role of the *YNVM*, from the *VVM* project point of view, is to allow the dynamic generation of domain-specific virtual machines.

To achieve that, the *YNVM* provides four "basic" services:

Code generation: a fast, platform and language independent dynamic compiler producing efficient native code that adheres (by default) to the local platform's *C* ABI[2];

[1] counted in bytes of source code.

[2] Application Binary Interface

Meta-data: are kept from the compilation, thus allowing higher-level software to reason about its implementation or the environment's one, and dynamically modify them;

introspection: on dynamically compiled code, the application and the environment itself;

Input methods: giving access to the compilation/configuration process at all levels.

The objective is to maximise the amount of reflective access and intercession, at the lowest possible software level, while preserving simplicity and efficiency. The execution model is similar to *C*, thus providing total compatibility with native applications and systems libraries. In addition to this *C*-like execution model, the use of a dynamic code generator allows performances similar to statically compiled *C* pograms.

Thanks to this compatibility any application can be build with a mix of *C/C++* and *YNVM*'s code, according to the semantics of each part of the program.

What's more is that even if it still uses a scheme-like front-end language, the introspection's facilities and the implementation's reification, allow you to change language features for ease of development. For example, by simply dynamically changing the parser, it is possible to switch from a functionnal paradigm to an imperative and infix *C* syntax style, letting the programmer choose, for each component of its application, the most appropriate paradigm to write it.

5 Flexible Web Caching

To illustrate the advantages of putting the flexibility, reflection and the dynamicity at the lowest possible software level (in the execution environment itself), we have developped a flexible web cache (called *C/NN*[3]) on top of the *YNVM*.

Flexibility in web caches comes from the ability to configure a large number of parameters[4] that influence the behaviour of the cache (protocols, cache size, and so on). What's more, some of these parameters, such as user behaviour, change of protocol or the "hot-spots-of-the-week" [17], cannot be determined before deploying the cache.

However, reconfiguring current web caches involves halting the cache to install the new policy and then restarting it, therefore providing only "cold" flexibility.

WebCal [13] and CacheL [2] are examples of web caches that bring flexibility through the use of domain-specific languages (DSLs). Being dedicated to a particular domain, a DSL offers a powerful and concise medium in wich to express the constraints associated with the behaviour of the cache. However, in spite of being well-adapted to the specification of new cache behaviour and even to

[3] The *Cache with No Name*

[4] See the configuration file for Squid...

formal proofs of its correctness, a DSL-based approach does not support "warm" reconfiguration.

Other work [1] proposed a dynamic cache architecture, in which new policies are dynamically loaded in the form of components, using the "strategy" design pattern [8]. While increasing flexibility, it is still limited: it is only possible to change those aspects of the caches behavior that were designed to be adaptable in the original architecture. This is the problem of using rigid programming languages/environments to build dynamically reconfigurable applications: limited reification and dynamicity lead to limited reconfigurability.

Because of it's being build directly over the *YNVM*, *C/NN* inherits its high degree of reflexity, dynamicity and flexibility and so provides "warm" replacement of policies, on-line tuning of the cache and the ability to add arbitrary new functionality (observation protocols, performance evaluation, protocol tracing, debugging, and so on) at any time, and to remove them when they are no longer needed.

In particular, as the reconfiguration strategy is a policy too, an administrator could dynamically define new or reconfigure existing adminstration/reconfiguration rules, performance metrics and associated monitors. Here are some examples of such reconfigurations:

1. when the request rate becomes high, the cache can start to manage a "black list" of servers with low response time and stop caching their documents (direct forward of the requests).
2. if the "byterate" is going down to a threshold, the cache can automatically switch to another policy[5], that saves more bandwidth (even with a worse hitrate or mean response time).

The *VVM* approach lets us instantiate an execution environment dedicated to web caching so that writing a new replacement strategy (from a paper) takes a tens of minutes (for someone familiar with *YNVM*) and the results is about a few lines of code, see figure 2 for an example. Because everything can be changed "on the fly" in the *YNVM*, it was possible to mix different paradigms in *C/NN*'s code (the functional scheme-like front-end and some statically compiled *C*), making code writing even more easy, quick and natural. Figure 2 shows an example of reconfiguration script, written in an infix style, including a reconfiguration function (*switch-to*), a new replacement policy (*filter-policy*, based on one found in [3]) and the reconfiguration command.

Solutions to software dynamic reconfiguration usually implie some "meta level" and degraded performances. Hence our main goal: to bring dynamic flexibility, reflexion and performances together.

Because of the quality of the code generator, the performances of an *YNVM*'s application are almost equivalent to *C* programs (and even sometimes better due to very aggressive optimization and partial evaluation techniques). We have compared *C/NN* to the widely used *Squid* cache version 2.3 on the basis of

[5] the choice can be based on meta-data associated with strategies and multi-criterion decision algorithms.

```
// cache reconfiguration function
defun switch-to(new-policy, num-to-re-evaluate){
  let head = get-worst(repository, num-to-re-evaluate);
  current-policy = new-policy;
  while(head){
    http-repository.update(repository, cell.data(head));
    head = next-cell(head);
  }
};
// a new replacement policy
defun filter-policy(doc) {
  if (system.strncmp("text",http.mimeType(doc),4))
     size-cost(doc);
  else
     gds-cost(doc);
};

// reconfiguration command (re-evaluate 20% of the cache)
switch-to(filter-policy,
          http-repository.size(repository)/5);
```

Fig. 2. A complete reconfiguration script.

their average response time (that is the performance criteria the user actually see): based on different traces collected at INRIA (from 100K to 600K requests). *Squid*'s response time is a few more than 1 sec, *C/NN*'s was about 0.83 sec. Handling a hit takes about 130 μs and about 300 μs for a miss. So having a dynamically reconfigurable web cache doesn't seem to imply having a less performant one.

Probably the most important issue is the cost of a reconfiguration. Switching from a policy to another pre-defined one, takes less than 50 μs. Because adding some new functionality implies compiling new codes, the amount of time needed to proceed a reconfiguration depends on the complexity of the extension that is being added, but defining a new replacement strategy takes about 400 μs, and can be compared to handling one request.

6 Related Work

The Virtual Virtual Machine project can be compared to different approaches.

Some work is being done around specialisable virtual machines, that is the generation of new, dedicated virtual machines for a given application domain and environment (operating system, hardware,...), as for example *JavaCard* [11] or *PLAN* [9]. The main difference with these approach is that (i) it does not provide the common language substrate and thus results in isolated virtual machines,

without any hope of interoperability; (ii) the specialised virtual machines, once generated, are still static and not flexible.

Flexible operating systems, such as *SPIN* [18] or *Exokernel* [4], and meta object protocol projects are also comparable to our project, however they focuse only on system's aspects and do not provide language flexibility. The security policy, needed to control the extensibility, although it is still a policy, and therefore should be extensible, is a static design choice that can not be changed. We argue that different application domains will probably have different security requirements and semantic, hence it is the responsability of (i) the administrator to customize inter-*VMLets* security rules; (ii) the *VMLets* to define security rules for a given application domain.

To address emerging application domains work is being done on embedded operating systems, as *MultOS* [14], *μClinux* [20] or SUN's *KVM* [12]. Each of those environments, while being well dedicated to emerging computing, are still rigid, closed and poorly interoperable.

Another research domain our work can be compared to is language interoperability. The objective is to support multiple language, and to allow them to interoperate, in a single execution environment. The *Universal Virtual Machine* [10] project from IBM aims at executing both *Java*, *Smalltalk* and *Visual Basic* applications. Nevertheless, it still only a rigid extension to an existing (*Smalltalk*) virtual machine (that understand three language instead of one): while allowing the support for three different language, it is neither reflexive nor extensible.

Microsoft *.Net* [15] is another project from this research domain. Microsoft framework aims at responding to the need of every possible user/application. Thus, it applies a "one-size-fit-all" approach which is known to (i) poorly face the evolution of applications requirements and/or semantics; (ii) penalize performances; (iii) be closed, and thus to impose artificial constraints to developpers. At the opposite, we want to give any user/application the ability to adapt the execution environment to its requirements and/or semantics, which result in a (i) better match with applications needs; (ii) more evolutive solution, as each emerging application domains will not implie an update of the framework; (iii) more performant execution environment, as an application will never suffer from a "one-size-feet-all" services like in traditionnal operating systems.

7 Conclusions and Perspectives

This paper presented our approach to dynamic flexibility and interoperability, based on a meta execution environment. The *RVM* and the *YNVM* have shown to be efficient for writting execution environment (few hundreds of line each) in two different contexts: active networks and flexible web caching. The resulting *VMLets* are small compared to traditionnal implementations. The *YNVM* have demonstrated that we can come close to having the best of several worlds: flexibility, dynamicity, simplicity and performances. It demonstrates that reconfigurability can be simple, dynamic and have good performances.

With the *VVM* projects we continue to investigate a systematic approach for building flexible, adaptable and interoperable execution environments, to free applications from the artificial limitations on reconfiguration imposed by programming environments.

The *YNVM* has been ported on bare hardware (PowerPC) and thus provides an environment for building dynamically dedicated and flexible operating systems. This gives applications an opportunity of executing "standalone", avoiding the need for a traditionnal operating system and its associated overheads and/or predefined abstractions.

Concerning active networks, work is being done on a generalisation of existing protocols, to achieve *Active Active Networks*. In *AAN*, the active protocols and the deployment protocols are dynamically instantiated on the machines.

The *Virtual Virtual Machine* may look as an ambitious project, but it seems to be highly relevant to address many current and upcoming topics like set-top-boxes, active networks, mobile telephony and embedded systems, to name but a few.

References

1. O. Aubert, A. Beugnard, *Towards a Fine-Grained Adaptivity in Web Caches*, in Proceedings of the 4th International Web Caching Workshop, April 1999. `http://www.ircache.net/Cache/Workshop99/Papers/aubert-0.ps.gz`
2. J. Fritz Barnes and R. Pandey *CacheL: Language Support for Customizable Caching Policies*, in Proceedings of the 4th International Web Caching Workshop, April 1999. `http://www.ircache.net/Cache/Workshop99/Papers/barnes-final.ps.gz`
3. E. Casalicchio and M. Colajanni *Scalable Web Cluster with Static and Dynamic Contents*, in Proceedings of IEEE International Conference on Cluster Computing (CLUSTER 2000), Chemnitz, Germany, December 2000.
4. M.F. Kaashoek, D.R. Engler, J. O'Toole, *Exokernel: an operating system architecture for application-level ressource management* Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain, Colorado, December 1995.
5. B. Folliot, *The Virtual Virtual Machine Project*, Proccedings of IFIP Symposium on Computer Architecture and High Performance Computing, Sao Paulo, Brasil,October 2000.
6. B. Folliot, I. Piumarta and F. Ricardi, *A Dynamically Configurable, Multi-Language Execution Platform* SIGOPS European Workshop 1998.
7. B. Folliot, I. Piumarta, L. Seinturier, C. Baillarguet and C. Khoury, *Highly Configurable Operating Systems: The VVM Approach*, in ECOOP'2000 Workshop on Object Orientation and Operating Systems, Cannes, France, June 2000.
8. E. Gamma and al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
9. M. Hicks and al. *PLAN: A Packet Language for Active Networks*, in Proceedings of the International Conference on Functional Programming, 1998.
10. IBM plans cross-platform competitor to Java, InfoWorld Electronic, April 1997.
11. `http://www.javacard.org`
12. `http://www.java.sun.com/products/cldc/wp/`

13. G.Muller, L.Porto Barreto, S.Gulwani, A.Trachandani, D.Gupta, D.Sanghi, *WebCaL: A Domain Specific Language for Web Caching*, in 5th International Web Caching and Content Delivery Workshop, 1999.
14. `http://www.multos.com`
15. `http://www.microsoft.com/net/`
16. S. Patarin and M. Makpangou, *Pandora: a Flexible Network Monitoring Platform* Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, June 2000.
17. M. Seltzer, *The World Wide Web: Issues and Challenges* , Presented at IBM Almaden, July 1996.
18. B. Bershad, S. Savage, P. Pardyack, E. Gun Sirer, D. Becker, M. Fiuczynski, C. Chambers and S. Eggers, *Extensibility, Safety and Performance in the SPIN Operating System* Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain, Colorado, December 1995.
19. `http://www.squid-cache.org/`
20. `http://www.uclinux.org/`
21. D.Wetherall, J. Guttag, D. Tennenhouse. *ANTS: A Toolkit for Building and Dynamically Deploying Network Protocol*, in Proceedings of IEEE OPENARCH'98, San Fransisco, USA, April 1998.
22. S. Michel, K. Nguyen, A. Rosenstein, L. Zhang, S. Floyd and V. Jacobson, *Adaptive Web Caching: towards a new global caching architecture*, Computer Networks and ISDN Systems, 30(22-23):2169-2177, November 1998.