YNVM: dynamic compilation in support of software evolution

Ian Piumarta, Frederic Ogel, Bertil Folliot*
Computer Science Laboratory, University of Paris VI

Abstract

The YNVM is a response to the rigidity inherent in current programming languages and environments, which present significant barriers to the construction of dynamically reconfigurable systems. It provides extreme *late binding* of system/language features, unconstrained application involvement in the management and use of *meta-data*, maximally *open reflective features*, and elevates concrete implementation (compiled code) to first-class status by providing *dynamic (re)compilation* of any part of a live system without loss of efficiency. It is compatible with existing code bases, and applications built on top of (or next to, or under) it can be made highly reconfigurable with minumum effort and negligible overheads.

1 Introduction

The YNVM (YNVM is Not a Virtual Machine) is our response to the limitations inherent in current programming languages and system environments in the face of dynamic and rapid evolution. Each of these limitations has presented us with insurmountable problems in the past. The four most troublesome, stated with minimal justification, are as follows:

Early binding commits a language or system to one particular model, process or paradigm early in the deployment process (typically at compile time). It locks out innovation, reduces the opportunity for retrofitting new discoveries made after system design, and precludes deep changes to the system in response to dynamically changing requirements or environment.[1]

Insufficient persistent meta-data which compilers construct during compilation and then throw away. Distributed systems, precise garbage collection, and runtime manipulation of system structure (amongst many others) become difficult (or even impossible) as a result. Although significant advances can be found in MetaXa for Java [METAXA] and Iguana for C++ [IGUANA], they can fall short of providing applications with *total* involvement in the management and use of meta-data.

No reification of implementation is possible unless the compiler, and its interpretation of program semantics, are first-class citizens of the runtime meta-level. Without this, no amount of late binding and involvement in meta-data management will be sufficient to solve all potential problems.

Artificially closed access to introspection and intercession is often imposed in the name of "security". This results in barriers to expressivity and malleability, unnecessary complexity (an additional, fixed interface to reflective features) and reduced generality (predicting every possible interesting use of reflection in advance of system design is impossible). It is far better to provide a "maximally open" environment, and then restrict access to reflective features that might compromise security as part of the system design/deployment process.[2]

The YNVM effectively addresses all of these issues in emphasizing above all else openness, simplicity and incremental reification of compiled code without loss of performance compared to statically-compiled C. It is also compatibly with existing code bases (compiled applications, system libraries, etc.). It can be used as a standalone programming environment, as a dynamic compilation service for existing systems, as a programmable run-time configuration tool for redefining component interconnectivity, to provide a high-level scripting capabilities, or any combination of the above.

The YNVM was built to solve a particular problem in the context of the VVM (Virtual Virtual

*Email: *first.lastName*@lip6.fr

[1] Some very powerful late-bound systems exist, such as Smalltalk. But these have performance, interoperability and OS/library/legacy compatibility issues.

[2] This is really a specific case of the "early binding" problem, but important enough to be listed seperately.

Machine) [VVM], a long-term project whose goals are related to the engineering of responsive, highly-configurable software systems. These goals are beyond the scope of this paper, which concentrates on eliminating pragmatic barriers to scalable software evolution. Nevertheless the context is relevant, so I begin by describing the requirements of the VVM in Section 2. The YNVM philosophy is explained in Section 3, followed by an overview of its architecture in Section 4. Section 5 desribes two of the systems that we have built using the YNVM, and finally Section 6 offers some conclusions concerning the YNVM in relation to engineering complex OO systems for evolution.

2 Context: the VVM

The VVM (Virtual Virtual Machine) is a new, systematic approach to adaptability and reconfigurability for portable, object-oriented applications based on bytecoded languages. Its main objectives are to: (i) facilitate adaptation of language/system characteristics to better fit application domains; (ii) allow "live" execution environments to evolve dynamically in response to changing requirements, protocols/standards and local conditions; (iii) minimize resource consumption through specialization/reuse of existing system components; and (iv) assure a consistent, common substrate on which true interoperability between languages can be achieved.

The principal role of the YNVM (in the context of the VVM) is therefore to instantiate langauge- and/or application-specific virtual machines (VMs) "on the fly". To achieve this, the YNVM provides the following "basic" services:

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

- persistent **meta-data** that allows higher-level software to reason about and dynamically modify the implementation of the system;

- **introspection** on dynamically compiled code, the application itself, and the system environment;

- a variety of **input methods** giving access to the compilation/configuration process at all levels: execution of configuration "scripts", compilation of structured program descriptions, function-oriented interfaces for program-driven code generation, and fine-tuning the individual instructions generated for a given construct or semantic.

3 The YNVM philosophy

Two aspects of the YNVM are emphasized above all others: simplicity and openness.

The list of facilities presented in the preceding section might suggest that the YNVM is an "agglutination of features", but the opposite is true.[3] It is a "crystallization of style" that employs the simplest, most general mechanisms possible to build an incremental, dynamic "C compiler"—and which then makes all of those mechanisms openly accessible to clients.

The YNVM recognizes the need to specialize its own implementation. It can reflect and intercede fully on itself. Higher-level software can extend, modify or redefine the management of meta-level data, and can intercede at (or reimplement) any level in the compilation chain.[4]

4 YNVM architecture

Figure 1 shows the structure of the YNVM.

The **parser** receives text from a file (or optional console interface) and creates a corresponding tree structure in the object memory.

The structures in the **object memory** are as simple as possible. They consist of four types of object: integers, strings, symbols ("unique strings" for representing program identifiers and storing named meta-data) and lists.

The **tree compiler** converts structures into instructions for a machine-independent, stack-oriented abstract machine supporting C-like operations and execution model. The abtract machine is simple (it has 41 instructions) but sufficient to represent any program. The tree compiler maintains the meta-data describing the current state of the compiled code, and obeys any user-defined

[3]The characterization of systems into "agglutinations" versus "crystallizations" first appeared in [Kay93], although it may well have earlier origins in the "Scheme versus Common Lisp wars."

[4]Following its philosophy of "maximum openness" much of the YNVM (more than half) is implemented by the YNVM itself, as code compiled dynamically during "boot".

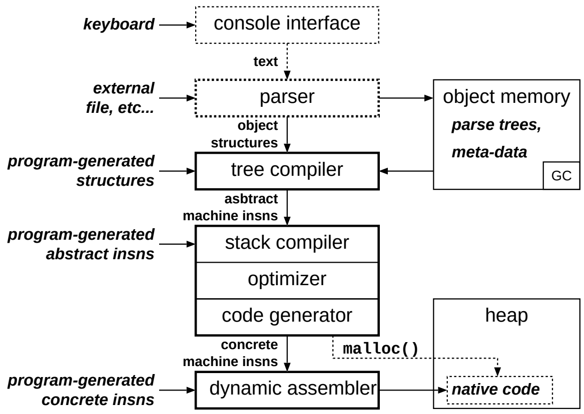

Figure 1: Structure of the YNVM.

transformations on the input tree defined in the meta-data. It also supports lightweight, hierarchical namespaces (which we call "modules") that help reduce conflicts and namespace pollution in the meta-data.

The **stack compiler** and **optimiser** perform machine-independent transformations on the stack-oriented abstract code, using a novel intermediate representation that permits fast yet effective optimisations.[5]

The **code generator** maps the intermediate representation onto concrete instructions for the local platform using an efficient minimum-cost tree covering algorithm. The concrete instructions are generated by a machine-dependent **runtime assembler**. Memory for the final code is obtained by calling `malloc`, or some similar allocator.

The entire compilation chain is fast. It generates more than half a megabyte of high-quality code per second on an average personal computer.[6] Reclamation of space in the object memory is automatic, transparent to the client application, and takes no processing cycles away from application execution.

5 Application domains

Several non-trivial applications are already running on the YNVM. These include a flexible, dynamically reconfigurable web cache and a standlone, object-oriented microkernel intended for deploying dynamically reconfigurable embedded systems.

[5]The intermediate representation is a structure combining both stack- and tree-like properties, and is a major factor in the choice of a stack-based abstract machine model.

[6]840 kilobytes/second on a 400 MHz PowerPC 750.

5.1 Flexible web cache

Web caches are interesting because they demand high levels of reconfigurability and exhibit complex and unpredictable behavior [ZMF+98].

Work in this area includes WebCal [MBG+99] and Cache-L [BP99] which offer flexibility through the use of *domain-specific languages* (DSLs). DSLs are a powerful and concise medium in which to express the constraints associated with the behavior of caches. However, they do not support *warm reconfiguration*. Other work [AB99] proposes a model for cache frameworks supporting warm reconfiguration, but only allows certain defined aspects of the cache to be reconfigured. Determining a single "correct" set of such aspects is impossible, since user (and therefore cache) behavior changes unpredictably and frequently [Sel96], and new protocols must be accommodated as they emerge.

Our YNVM-based approach supports initial configuration (based on simulation driven by captured traces of network activity), and dynamic adaptation of that configuration in response to observed changes in traffic as they happen. The YNVM provides the configuration language and dynamic recompilation for instantiating new components and redefining inter-component connectivity.

Figure 2 shows an example of changing the replacement policy. The reconfiguration function `evalSwitch` enumerates the "worst" documents currently in the cache, applies the new cost function `filterPolicy` to each of them, and then updates the cache contents to reflect their new cost. (The cache replacement policy shown in this example is similar to one used in [CC00].)

Our cache performs comparably to popular caches such as Squid [Squid]. Dynamic reconfiguration is sufficiently fast that clients do not notice any interruption of service. The time required to change the replacement policy (including reevaluation of 5,000 documents) is about 20ms. An additional 0.5ms is required (on average) if the incoming policy is not already in the system and has to be compiled from source.

Ease of use is also an important factor. As shown in Figure 2, typical replacement and reconfiguration functions are short and quickly written. Since they are dynamically compiled and bound to the running system, they can make use of and/or implement arbitrary functionality; there are no "arti-

```
// cache reconfiguration function
defun evalSwitch(newPolicy, numToReevaluate)
{
  let head = getWorst(repository, numToReevaluate);
  currentEval = newPolicy;  // change active policy
  while (head)
  {
    let oldCost = docs.cost(cell.data(head));
    newPolicy(cell.data(head));
    httpRepository.update(repository,
                          cell.data(head),
                          oldCost);
    head = cell.next(head);
  }
};

// new replacement policy
defun filterPolicy(doc)
{
  if (system.strncmp("text", http.mimeType(doc),
                     4))
    sizeCost(doc);
  else
    gdsCost(doc);
};

// reconfiguration command
evalSwitch(filterPolicy,
           httpRepository.size(repository) / 5);
```

Figure 2: A reconfiguration script. The source language is very simple, provided by minimal infix parser that generates YNVM tree structures directly. Symbols that include a period '`.`' are qualified names, where the component before the period indicates the YNVM module (namespace) in which the compiler should look up the symbol.

ficial barriers" to the extensibilty of our web cache.

We also have a web traffic monitoring tool [PM00a] which works with the YNVM.[7] We are currently investigating the use of feedback from this monitoring tool to automate reconfiguration decisions [PM00b]. The monitor drives reconfiguration in the cache, which in turn uses the monitor to evaluate the decisions that were made. The process of automatically "learning" how to make good decisions is complex, and the topic of ongoing research.

[7]It is implemented as a shared library, which the YNVM can load and use simply by calling `dlopen()`.

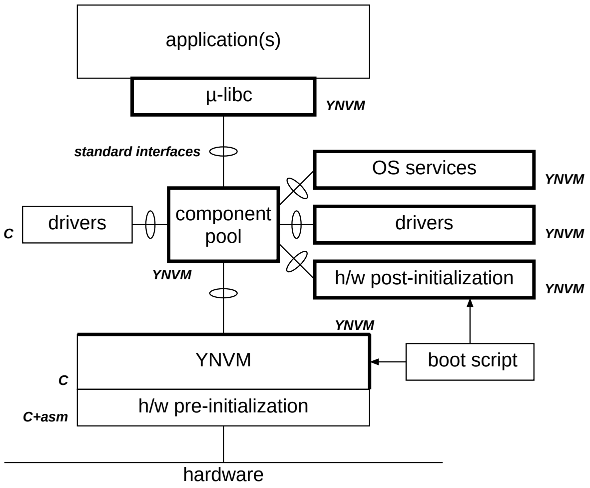

Figure 3: Organization of a standalone YNVM. The initial hardware setup, a few essential drivers and the static part of the YNVM are linked to form a bootable image. The remainder of the system is constructed dynamically.

5.2 Operating system kernel

The YNVM can run on bare hardware and is being used to experiment with adaptable embedded systems and application-specific kernels.[8] The organization of a "standalone" YNVM, which has some similarity to the L4 microkernel [L4], is shown in Figure 3.

A bootable image (initial hardware setup, essential drivers needed for booting, and the guts of the YNVM itself) has to be configured statically. This image contains just enough to put the hardare in a known state, map some memory for the YNVM to run in and provide it with a means to communicate with the outside world. From that point on the YNVM constructs the rest of the OS dynamically.[9]

The boot script creates the dynamic parts of the YNVM implementation, any additional drivers (e.g., network packet interfaces), and miscellaneous

[8]The PowerPC version is based on the Think "non-kernel" developed by Jean-Philippe Fassino at France Telecom, which in turn is based on the LinuxPPC kernel bootstrap code. Unfortunately there are no references, published or online, to his work.

[9]There is a small, built-in boot script that is executed when the YNVM first receives control. Although it is feasible to dynamically construct some of the essential drivers (keyboard and console, for example) in this boot script, having at least the console driver pre-compiled and available when the YNVM starts to run is useful for troubleshooting.

OS services (e.g., TCP/IP stack, minimal process model, and CPU inheritance-based scheduling). It also includes the implementation of the component pool, a "registry" of object-oriented system components (drivers and services).

Each component is defined within its own module (namespace) in the YNVM. (In this case the YNVM modules are used to hide component implementations, similarly to classes in languages like C++.) One of these components is "component" itself, which provides the following services:

- definition of the state associated with a concrete instance of a component;
- construction of one or more "virtual tables" used to invoke methods on a component instance;
- instantiation of a component as an object (in the application's memory, not the YNVM's object memory) which contains a virtual table pointer and instance-specific data;
- extensions to the YNVM compiler that implement virtual method call on comonponent instances.

The compiler extensions are made by extending the meta-data within a module that implements a component. For example, each such module is augmented with the symbol '`->`' associated with a compile-time function.[10] When the YNVM tree compiler sees a parse node whose head is the symbol '`->`' the associated compile-time function is executed to implement the virtual method call.

Given some infix parser, for example, this allows methods to be invoked on components in a familiar manner:

```
aComponent -> aFunction(someArguments, ...);
```

Since the conversion of the method identifier into a virtual table offset is performed at compile time, the resulting compiled code is as efficient as a virtual function call in C++. A complete example of importing a system component and invoking methods across its interface is shown in Figure 4.

```
// define a console driver component
module(dev.console);    // create a namespace

// define the instance format
component.define-state
  fb            // framebuffer address
  cols rows     // fb geometry
  x y           // fb cursor position
;

// define the default interface
%default = component.export-methods
  putc(char)
  putcs(string)
  putxycs(x, y, string)
;

// trivial interface constructor
defun new()
{
  component.new(%default,      // methods
                dev.console);  // state
};

module(global);

// create and use a console

define myConsole = dev.console.new();

myConsole -> putcs("hello, world\n");
```

Figure 4: A simple system component. Spaces around the '`->`' "operator" are required because the parser (and YNVM in general) is lenient concerning symbol names. ('`myConsole->putcs`' would represent a single symbol.)

The ability of the YNVM to change arbitrary parts of the system implementation can be used to replace component interfaces at any time. For example, if an application requires a process model with memory protection to ensure safe execution of different threads, the component interfaces can be "wrapped" with IPC facilities to perform calls via message passing or address translation on the method arguments. (One might consider this as a form of as aspect-oriented programming. An "IPC aspect" is added to the methods of system components "under the feet" of the running application, which is unaware of any change in the system environment and whose code continues to work without modification.)

"Componentization" of system services allows the system to choose between multiple possible implementations of a given service (e.g., appropriate drivers chosen during the boot-time hardware

[10]The choice of symbol is arbitrary and can, like everything else, be redefined by the application (or code associated with the applications "parser") if it so desires.

"probe"), and to modify (or replace) the implementation or behavior of any system component either wholesale (for reconfiguration) or depending on the nature of the "client" requesting the service (for resource protection). Making the implementation of a component "private" (and impossible to access without going through the component interface) is trivial, and requires simply eliminating everything but the "public" interface from the YNVM module in which the component is defined.

Few performance measurements have been made with the standalone YNVM system (which has only been operational for a short time), but initial experiments are encouraging. One early experiment was to echange "ping" packets between two standalone YNVMs. The round-trip time was 120μsecs on an unloaded network. This compares favorably with a round-trip time of 160μsecs for the same machines running a Linux kernel.[11]

Current work includes porting our experiments with active routers (running on earlier, inefficient VVM prototypes) to the YNVM.[12] This will produce what we believe to be the first practical, dedicated, high-performance, "turnkey" active router (supporting remote management and arbitrary reconfiguration) to run on stock hardware.

6 Conclusion

The YNVM is the result of several years work spent building progressively more open systems. During this time we have noticed an interesting (and seemingly general) effect: increasing the "openness" of a system[13] tends to make it simpler, smaller and *more powerful*.

Increased simplicity and reduced size come from a smaller feature set. The smaller feature set comes from two aspects of increased openness: fewer high-level "building blocks" (anything that is "missing" can be constructed trivially, by client software itself, out of lower-level facilities), and the almost total absence of *explicit* reflective features (everything is open, accessible and mutable by default).[14]

Increased "power" is more difficult to quantify, but is evident in at least three areas: expressivity, leverage from reuse and specialisation of existing parts of the language/system, and the absence of "artificial barriers" imposed on an "ideal" solution by unnecessary rigidity in the language or system environment.

The remainder of the conclusion relates the YNVM to the workshop "problem statement".

Responsiveness to changes in specification and requirements follows from the YNVM's ability to configure a system (be it standalone or an application running on a traditional OS) according to a high-level description at startup. Its reflective capabilities allow a system to reason about both its own internal state and its environment, and to make appropriate *configuration decisions*. Such configuration decisions can be reversed at any time. Persistent meta-data and the ease with which any part of the implementation can be recompiled "on the fly" provide a powerful environment for *dynamic evolution*.

Responsibility for the management of this evolution is shared between the YNVM (which knows the current state of the system and its compiled code) and the designer of the application (who must decide how and when to reconfigure and/or recompile). While the YNVM introduces no barriers to the design of *ad-hoc solutions*, it also provides all the building blocks needed to create *flexible, highly-structured solutions*.

The openness of the YNVM architecture and the adherence to the C ABI helps these solutions to be *generally applicable*, including seamless integration with legacy code.

The system designer can choose the most appropriate level at which to interact with the YNVM, and the learning curve is shallow.[15] Dynamically-compiled code is compatible with existing OS services and interoperates seamlessly with statically-compiled code. The ability to intervene at any level in the compilation chain facilitates interoperability with "foreign" ABIs.

[11] Linux has the fastest networking stack of any widely-used kernel.

[12] Similarly to flexible web caching, active routing is a domain that places severe demands on the dynamic reconfiguration capability and implementation efficiency of its host system.

[13] By which I mean increasing the system's capacity for introspection and intercession, and perhaps also reducing the number of *artificially-imposed* limits on them.

[14] On Unix, the single "reflective primitive" of the YNVM consists of three lines surrounding a call to `dlsym()`.

[15] The rapidity with which one can begin to perform "surgery" on deep features of a live system, and the overwhelming sense of *fun* that accompanies it, make following this learning curve almost compulsive.

A Appendix

I will happily demonstrate the YNVM during/after the workshop, running as a client of a full-featured OS (GNU/Linux) or as a self-hosting OO operating system environment on bare hardware.

References

[AB99] O. Aubert and A. Beugnard, *Towards a Fine-Grained Adaptivity in Web Caches*, Proceedings of the 4th International Web Caching Workshop, April 1999. `http://www.ircache.net/Cache/Workshop99/Papers/aubert-0.ps.gz`

[BP99] J. Fritz Barnes and R. Pandey, *CacheL: Language Support for Customizable Caching Policies*, Proceedings of the 4th International Web Caching Workshop, April 1999. `http://www.ircache.net/Cache/Workshop99/Papers/barnes-final.ps.gz`

[CC00] E. Casalicchio and M. Colajanni, *Scalable Web Cluster with Static and Dynamic Contents*, Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2000), Chemnitz, Germany, December 2000.

[IGUANA] `http://www.cs.tcd.ie/Tilman.Schaefer/project.htm`

[Kay93] Alan C. Kay, *The Early History of Smalltalk*, ACM SIGPLAN Notices 28(3), 69–75, March 1993.

[L4] `http://os.inf.tu-dresden.de/L4/`

[MBG+99] G. Muller, L. Porto Barreto, S. Gulwani, A. Trachandani, D. Gupta and D. Sanghi, *WebCaL: A Domain-Specific Language for Web Caching*, 4th International Web Caching and Content Delivery Workshop 1999.

[METAXA] `http://www4.informatik.uni-erlangen.de/Projects/PM/Java/`

[PM00a] S. Patarin and M. Makpangou, *Pandora: a Flexible Network Monitoring Platform*, Proceedings of the USENIX 2000 Annual Technical Conference, San Diego, June 2000. `http://www-sor/publi/PFNMP_usenix2000.html`

[PM00b] S. Patarin and M. Makpangou, *Monitoring Web Proxy Caches*, INRIA Research Report 4023, October 2000. `http://www-sor.inria.fr/publi/MWPC_rr4023.html`

[Sel96] Margo Seltzer, *The World Wide Web: Issues and Challenges*, presented at IBM Almaden, July 1996. `http://www.eecs.harvard.edu/margo/slides/www.html`

[Squid] `http://www.squid-cache.org/`

[VVM] `http://www-sor.inria.fr/projects/vvm/`

[ZMF+98] 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.