Applying the VVM Kernel to Flexible Web Caches

Ian Piumarta Frederic Ogel Bertil Folliot Carine Baillarguet

Laboratoire d'Informatique de l'Université Paris VI
and
INRIA Rocquencourt

email: *firstName.lastName*@inria.fr

Abstract

The Virtual Virtual Machine is a project to build a flexible application execution environment (languages, API, OS services) that provides at the *same* time: flexibility, adaptability and interoperability—leaving developers to concentrate on the computer science problem to be solved, rather than on the problem of adapting their applications to the limitations of a traditional execution environement. This paper presents one application of our approach, in the domain of flexible web caching. We show that by using a highly-flexible execution and implementation environment, the applications built on top of it "inherit" a very high degree of flexibility with negligible overheads.

1 Introduction

The VVM (virtual virtual machine) is a new, systematic approach to adaptability and reconfigurability for portable, object-oriented applications based on byte-coded languages such as Java and Smalltalk. The main objectives of the VVM are (i) to allow adaptation of language *and* system according to a particular application domain; (ii) to provide extensibility by allowing a "live" execution environment to evolve according to new protocols or language standards; and (iii) to provide a common substrate on which to achieve true interoperability between different languages.

At the heart of the VVM is a highly-flexible language kernel, called the YNVM (YNVM is Not a Virtual Machine). This kernel consists of a minimal, complete programming language in which the most important goal is to maximise the amounts of reflective access and intercession that are possible—at the lowest possible "software level".

The YNVM is useful as an environment for the implementation of many kinds of highly flexible system, not just as the kernel of the VVM. To study its effectiveness we have used it in several demanding application contexts—including the development of flexible web caches.

Flexibility in web caches comes from the ability to configure a large number of parameters that influence the behaviour of the cache (protocols, cache sizes, and so on). Selecting an "optimal" set of parameters in any given context is a difficult problem, depending on network traffic, the rate of page updates and users' interests [Sel96].

However, reconfiguring current flexible web caches involves halting the cache to install the new policy and then restarting it, therefore providing only "cold" flexibility. Previous content of the cache is lost, and more importantly the tuning of the parameters cannot be made on-line.

Our flexible cache architecture is built directly over the YNVM "kernel", from which it inherits all the benefits of the YNVM's high degree of reflection and intercession (which are essential for building truly flexible systems). It provides "warm" replacement of policies (i.e., without stoping the cache or losing its contents), whithout compromising the ease of writing new protocols found in existing web caches [MBG+99]. Other advantages include the ability to tune the web cache on-line, to add arbitrary new functionality (observation protocols, performance evaluation, protocol tracing and debugging facilities, and so on) at any time, and to remove them when they are no longer needed.

The remainder of this paper starts by briefly presenting the VVM kernel in Section 2. Section 3 introduces flexible web caches, and our cache is described in detail in Section 4. An evaluation of our system appears in Section 5, followed by conclusions and perspectives in Section 6.

2 Our architecture

The YNVM contains the following components:

- A small (optional) *reader* converts plain text input (in a parenthesised, prefix notation) into a data structure resembling a parse tree.[1]
- A *dynamic compiler* front-end converts the structured representation of the "program" into a stream of low-level bytecodes.
- A dynamic compiler back-end converts the stream of bytecodes into native code for the local platform, performing various cheap (and above all *fast*) optimisations to produce code of acceptable quality.[2]

[1] Lisp hackers will immediately recognise this concept.

[2] Performance is within a few percent of optimised C, although space is slightly larger.

- An object-oriented environment (with automatic, transparent memory management) supports both the structured form of programs, and a namespace facility.

The compiler generates code that (by default) adheres to the platform's C ABI, which implies seamless integration with any local OS that might be present. (The current prototype runs under Unix on PowerPC-based machines.)

Reflexivity is provided by two mechanisms:

- access to the object memory implemented by the runtime environment, allowing a program to reason about its own structure;
- access to the functions and variables that make up the precompiled part of the runtime system, via a `dlopen`-style interface.[3 4]

Intercession is provided by two mechanisms:

- first-class access to the object memory implemented by the runtime systems, allowing a program to modify the structures referred to by the compiler during the reading, transformation to bytecodes, and compilation to native code;
- first-class access to the compiled functions of the runtime, allowing a program to replace arbitrary parts of the runtime with new, dynamically-compiled, versions.

An extreme (but perfectly valid) way of considering the YNVM is as an *interpreted* implementation of C—in which everything that is normally static in C becomes dynamically modifiable—which employs a Lisp-like syntax to eliminate the distinction between programs and data, and which uses a very fast, dynamic (runtime) compiler to eliminate almost entirely the overheads traditionally associated with interpretation vs. statically-compiled (and optimised) C.

3 Example application: flexible web caches

An increasing amount of work is being done in the area of flexible web caches. This is because web caches represent an extreme problem of configurability, and exhibit complex and unpredictable behaviour [ZMF+98].

WebCal [MBG+99] and *Cache-L* [BP99] are examples of web caches that offer flexibility through the use of *domain-specific languages* (DSLs). Being dedicated to a particular domain, a DSL offers a powerful and concise medium in which 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 formal proofs of its correctness, a DSL-based approach does not support *warm reconfiguration.*

Other work [AB99] has proposed a dynamic cache architecture in which new policies are dynamically loaded in the form of components, using the *strategy* "design pattern" [G+94]. While increasing flexibility, it is still limited: it is only possible to change those aspects of the cache's behavior that were designed to be adaptable in the original architecture.

Reconfiguration is a complex problem in existing systems due to the number of different parameters that must be taken into account. (The configuration file for Squid [Squid] is a case in point.) What's more, some of these parameters cannot be determined before deploying the cache: user behaviour, changes to protocols and the "hot spots of the week" [Sel96], for example.

One popular solution is to capture "traces" of real activity and then use these traces to drive a simulation of cache behavior [PM98]. The performance of the simulated cache is analysed while varying its parameters, in order to find an "optimal" configuration. This configuration is then applied to a live system, but will not necessarily correspond well to the (continually varying) conditions of real traffic.

4 C/NN

Our approach supports both initial configuration based on simulation, *and* dynamic adaptation of that configuration in response to observed changes in real traffic as they happen. The resulting cache, called C/NN,[5] is based on five components:

- the YNVM, which provides the cache configuration language;
- basic replacement policies, such as LRU, SIZE, GreedyDualSize, and so on;
- basic reconfiguration policies (explained below), which control the passage from one cache policy to another;
- basic tools for cache observation, trace, measurements, and debugging;
- an "open" interface allowing external agents to add, modify or remove cache replacement and reconfiguration policies or tools, and to adjust the configuration parameters;

This approach is highly reflexive because the dynamic management of the cache is expressed in the *same* language that is used to *implement* the cache.

The cache can be modified at any time. New functionality and policies can be introduced into the running cache, and then the "current bindings" modified

[3]When the YNVM is running over a real OS, the mechanism is literally `dlopen` and related functions.

[4]The YNVM contains an absolute minimum of functionality. When it is first launched it immediately intercedes on itself to implement the missing functionality. It starts this process with access to only *one* "built-in" primitive function: `dlsym`.

[5]The *Cache with No Name*.

in order to activate them. It is therefore possible to rewrite arbitrary parts (or even all) of the cache implementation *during* execution.

Replacing one policy with another involves knowledge of the semantics associated with the cache data, by both the outgoing and incoming policies (and traditionally knowledge of the system by the administrator). Given this knowledge we can implement a reconfiguration policy to perform the necessary transformation.

To facilitate this transformation we provide several reconfiguration policies:

- no transformation—the replacement policies are simply exchanged (suitable for very similar replacement policies);
- re-evaluation of the least popular documents, to calculate their "cost" according to the incoming policy (to protect documents that might become more important under the new policy from being prematurely ejected from the cache before being re-evaluated "lazily" by the incoming policy once active);
- re-evaluation of the entire cache.

Of course, arbitrary reconfiguration policies can be defined dynamically by the administrator. For example, re-evaluation of the $x\%$ "worst" documents and the $y\%$ "best" documents (corresponding to the "critical" documents in the cache) to protect them against (or favour them for) early ejection from the cache by the incoming policy.

Figure 1 shows an example of a reconfiguration script, and the command that is used to invoke it. The reconfiguration function `xEvalSwitch` 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 the example is inspired by one used in [CC00].)

```
;; new replacement policy

(define filterPolicy
  (lambda (doc)
    (if (= 0 (:system.strncmp
               (:httpHdr.typeMime doc) "text" 4))
        (GDS-cost doc)
        (SIZE-cost doc))))

;; reconfiguration function

(define xEvalSwitch
  (lambda (newPolicy size-to-re-eval)
    (let [(head    0)
          (oldCost 0)]
      (set! currentEval newPolicy)
      (set! head (getWorst repository size-to-re-eval))
      (while head
        (set! oldCost (:docs.cost (:cell.data head)))
        (currentEval  (:cell.data head))
        (:httpR.updt repository
                     (:cell.data head)
                     oldCost)
        (set! head (:cell.next head))))))

;; reconfiguration command

(xEvalSwitch filterPolicy (/ (httpR.size repository) 5))
```

Figure 1: A reconfiguration script

5 Evaluation

In order to evaluate our cache we made both quantitative and qualitative measurements.

We timed various activities within our system, in order to evaluate the overheads of dynamic reconfiguration. Timing the principal operations in our cache was trivial, because of the use of the highly-reflexive YNVM at the lowest level. We were able to "wrap" timers around the functions implementing "critical" operations, without even stopping the system. (This is an example of the degree of reflection provided by our architecture: the observation tools of the cache were "pointed back" on the cache itself, allowing it to measure its own performance.)

For a total cache size of between 5,000 and 10,000 documents, the average time to find a document is 30μs.[6] Processing a cache hit requires approximately 500μs, and just less than 1ms for a cache miss. Cache performance is therefore clearly comparable to that of systems such as Squid [Squid].

The most important aspect to evaluate in a dynamically-flexible cache is the cost of adaptation. The total cost of replacing a given replacement policy with another, involving the re-evaluation of 5,000 documents, is about 20ms. Replacing a policy with another that has already been defined (and which therefore already exists in the system) costs about 47μs. Replacing a policy with another that is not yet defined (and which must therefore be read into the system, dynamically compiled, and then bound to the running code) costs about 500μs. This recompilation represents the cost of using a highly-dynamic execution environment, and is small in comparison to the 20ms needed to re-evaluate the cache contents.

It seems clear that dynamic flexibility does not greatly penalise the performace of the cache. Changing to an existing policy costs far less than processing a cache hit; changing to a new, previously unknown policy costs about the same as processing a cache hit.

It is also important to consider the ease of use of reconfiguration in our cache. As shown in Figure 1, typical replacement and reconfiguration functions are short and quickly written. Since they are dynamically

[6] We choose this size of cache because Squid keeps an in-memory working set of only 8Mb (by default), and by limiting the number of documents to around 5,000 we avoid any "swapping" that might "contaminate" our results.

compiled and bound to the running system, they can make use of and/or implement arbitrary functionality; there are no predefined limits to the extensibilty of our web cache.

Nevertheless, the cost of changing the cache policy can be high, if the new policy depends on significant quantities of metadata or on the contents of "external" documents.

6 Conclusions and perspectives

This paper presented and evaluated the benefits of using a highly-flexible language kernel, the YNVM, to solve a specific computer science problem: flexible web caching.

The resulting web cache, C/NN, is an application that is free of artifical limits on reconfigurability imposed by the implementation environment. It demonstrates that reconfigurability can be simple, dynamic (applied at run-time) *and* have good performance. The main lesson of this experiment is that we can come close to having the best of several worlds: flexibility, dynamicity, simplicity and performance.

With the VVM, we continue to investigate a systematic approach to building flexible, adaptable and interoperable execution environments—environments that let developers concentrate on real problems rather than the inconveniences of their rigid environments.

Concerning flexible web caching, Pandora [PM00a]—a system for real-time evaluation of the performance of web cache configurations—has just been ported into the YNVM. This opens the way for "self-adapting" web caches, where the policies in effect are constantly re-evaluated and modified as and when needed [PM00b].

Other work is in progress, such as porting the YNVM to very small nanokernels. This brings to applications the possibility of executing standalone on bare hardware—avoiding the need for a "traditional" operating system and its associated overheads. This approach is useful for many domains in which dynamic reconfiguration is the primary concern, such as routers, active networks, mobile telephony and set-top-boxes—to name but a few.

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.

[G+94] E. Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[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.

[PM98] G. Pierre and M. Makpangou, *Saperlipopette!: a Distributed Web Caching Systems Evaluation Tool*, Proceedings of Middleware '98, September 1998. http://www-sor.inria.fr/publi/SDWCSET_middleware98.html

[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/

[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.

Appendix

Our flexible web cache is (nominally) publically available. However, the YNVM on which it depends is currently only distributed to a limited number of collaborators. We are prepared to make individual arrangements for interested parties to have access to the YNVM.