Application-Level Concurrency Management

Frederic Ogel, Gael Thomas, Bertil Folliot, Ian Piumarta
Regal Group
INRIA Rocquencourt
Domaine de Voluceau 78153 Le Chesnay, France
&
Laboratoire d'Informatique de Paris VI, CNRS,
Université Pierre et Marie Curie,
4, place Jussieu, 75252 Paris Cedex 05, France
firstname.lastname@inria.fr,
`http://www-sor.inria.fr/projects/vvm`

Abstract. Traditionally an execution environment faces a trade-off between providing high-level or low-level concurrency mechanisms. The former trades flexibility for ease-of-use, while the latter results in a concurrency management closer to applications needs at the cost of an increase in the complexity of the applications code. Thus, one way or another, an application programmer has to match his application's semantic to the set of abstractions exported by the target execution environment.

Most execution environments, such as Java or Corba, are still rigid and closed and thus export high-level and general purpose abstractions that prevent application programmers from having any control or knowledge on the way their applications behave. Because concurrency concerns are closely related to applications semantics, a "one-size-fits-all" approach does hardly work.

Hence, we propose a flexible and minimal execution environment[1] that allows dynamic construction of dedicated execution environments and dynamic reconfiguration at both the execution environment and the application level. We present its architecture and its utilization to construct a dynamically adaptable Java runtime that exploits this flexibility to overcome some limitations of traditional Java environments.

Keywords: dynamic flexibility, adaptable concurrency, virtual machine.

1 Introduction

Writing concurrent applications is not a trivial task. Developers must prevent deadlocks and inconsistencies, while preserving parallelism. Since this trade-off between performance and consistency is closely related to an application's semantic, the suitability of the abstractions exported by the execution environment is critical.

The problem lies in the limited flexibility of traditional execution environments. While flexible operating systems only offer service replacement in a controlled way, most often for security concerns, language-based extensible systems, such as Java or Corba, are still monolithic and closed execution environments. This lack of flexibility results in a "one-size-fits-all" approach to system and language abstractions, which leads programmers to concentrate on

[1]This work is partially funded by the European IST project COACH (2001-34445).

the adaptation of their application to the exported abstractions, rather than on their application's internal problems.

Indeed, as limitations of existing environments are identified, ad-hoc solutions are proposed, most often to solve a particular problem or limitation. Those solutions are thus as rigid and closed as the original environments they are based on. Most of the research projects focusing on solving a limitation in Java end up with a modified version of an otherwise standard Java Virtual Machine (JVM). For example, to introduce reflection in Java, like MetaXa [17], object persistence, as in PJama [19], real-time (RT-Java [13]) or compilation optimizations, like Marmot [18], it was necessary to develop a new Java runtime, because the original was not flexible enough to be dynamically adapted. What about a Java runtime offering object persistence and compilation optimizations? It has to be another dedicated Java runtime.

In particular, the continuous emergence of novel application domains and computing models have led to a significant proliferation of scheduling algorithms. This increasing diversity demonstrates that no single scheduling policy is adequate for meeting the requirements of all applications.

In response to traditional environments lack of flexibility, the Virtual Virtual Machine (VVM) [4] is a project to build open and flexible execution environments including both system and language aspects. Its objectives are: (i) to allow dynamic adaptation of the execution environment to match any application-specific semantics; (ii) to provide a common language substrate on which to achieve interoperability between different languages or application domains. Whereas traditional execution environments are still closed and rigid, the VVM's complete flexibility lets developers concentrate on their application's internal problems rather than on the problem of adapting their application to the limitations of the target execution environment.

We propose in this paper the architecture of a minimal execution environment, composed of a Hardware Abstraction Layer (HAL) and a flexible dynamic compiler. Because there are no predefined and imposed abstractions, any execution environment can be dynamically instantiated upon it. Moreover, by combining the dynamism and the reflexivity inherent in the dynamic compiler, every piece of code can be dynamically reconfigured.

The paper is organized as follows. The next section describes the architecture of our minimal yet flexible execution environment. Section 3 presents the construction of a dynamically adaptable Java runtime and some evaluations. Then, section 5 examines related work and section 6 summarizes and concludes.

2 A flexible execution environment

In order to provide complete application flexibility, the execution environment must be completely open and free of predefined abstractions (hence following the Exokernel approach). Therefore, our architecture relies on a minimal execution environment composed of a HAL and a dynamic compiler, as illustrated in Figure 1.

The HAL is in charge of hardware initialization at boot-time. It reifies hardware resources through a set of functions in a policy-neutral way, that is, without adding any semantics to them. The only abstraction needed by the dynamic compiler is a memory allocator, therefore the HAL only defines a basic flat memory allocator, through malloc/free functions. Drivers for network cards, frame-buffer, or irqs can be defined later, using the dynamic code generator.

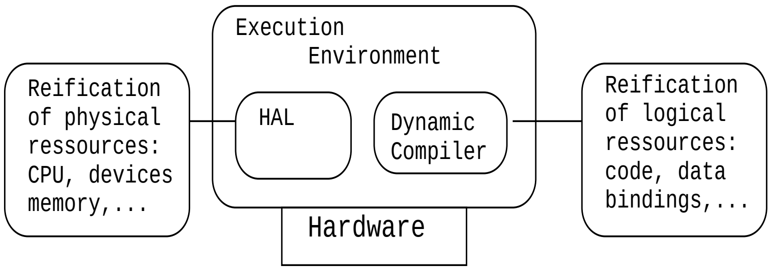
Execution Environment
HAL
Dynamic Compiler
Hardware
Reification of physical ressources: CPU, devices memory,...
Reification of logical ressources: code, data bindings,...

Figure 1: Architecture of the minimal execution environment.

Whereas our current prototype use the THINK[2] [6] nano-kernel as a HAL (developed by France Telecom R&D), the dynamic compiler also runs as a standard Linux application or embedded in a Linux kernel module.

The centerpiece of our execution environment is the dynamic compiler we have realized, called the YNVM.[3] It provides a complete reflexive language, and an execution environment for both applications and specifications of domain-specific execution environments. It has been developed in the context of the VVM project [28] in order to dynamically instantiate application-domain specific virtual machines.

The YNVM is structured as a set of components and interfaces based on the ODP Reference Model [20, 21, 22], such as lexer, parser, tree optimizer, tree compiler or code generator, as illustrated in Figure 2. Components implementing those interfaces can be dynamically (re-)composed to form any dedicated chain of compilation. Since the execution model is similar to that of C, applications in different execution format, from native ELF binaries to any byte-coded languages, can be loaded and executed.

Moreover, having a single execution engine, which acts as a common language substrate, allows to achieve complete interoperability and data sharing at both the execution environments and applications level.

The dynamic compiler relies on a garbage-collected object memory, used to store meta-data. An optional parser converts text, obtained by any input method, into an Abstract Syntax Tree (AST) stored in this object memory. A “tree compiler” then converts ASTs into instructions for an abstract stack machine, whose semantics and execution model are those of C. The tree compiler also provides meta-data reflecting the state of compiled code and applies transformation rules supplied at the application level, if any. Meta-data are organized into hierarchical namespaces called *modules*. Using two abstraction levels (AST and stack machine) allows two independent and specific optimization processes. Finally, a code generator (containing a platform-specific dynamic assembler) converts the intermediate representation into concrete machine instructions. The generated instructions are not stored in the internal object memory, but rather in the application’s memory.

Although most expressions are read, compiled and then executed, a mechanism (called *syntax*) provides for the definition of AST nodes that are executed during the dynamic compilation process, allowing dynamic code verification and arbitrary dynamic transformation/rewriting of tree structures.

Upon the YNVM, entire execution environments are dynamically constructed, from low-

[2]for THINK Is Not a Kernel
[3]for YNVM is Not a Virtual Machine.

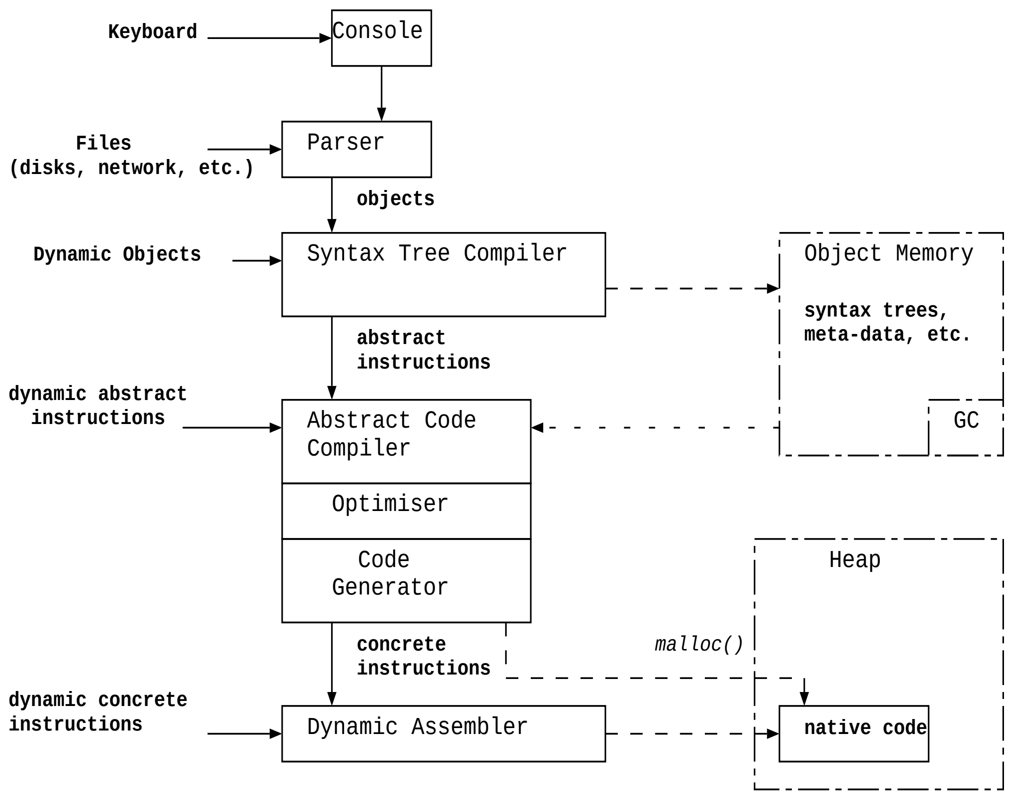

Figure 2: Structure of the YNVM.

level I/O drivers (keyboard, network cards or hard disk) to more sophisticated abstractions such as schedulers, TCP/IP network stacks or complete mobile agent environment. The predefined flat memory allocator, required by the YNVM, can then be replaced by a more complex one, in order to provide isolation between applications or simply to keep track of resources consumption. Likewise, the loading and compiling mechanisms can be adapted to enforce some peculiar security policy (based on code verification).

Such description of a complete virtual machine or execution environment is called a VMlet. Since they are implemented over the YNVM, VMlets are portable.

3 Application-level concurrency management in Java

Although Java has become a de-facto standard for distributed and concurrent programming, it is still a good illustration of the inherent problems of rigid and monolithic execution environments. It defines a small set of high-level abstractions, basically threads, wait/notify synchronization and an object-level locking mechanism (i.e. synchronized methods). Because the Java runtime is built on top of an operating system, those abstractions are bound to the concurrency model exported by the underlying kernel. In the name of portability, these abstractions need to be generic and high level enough to hide the particularities of those underlying kernels. As a result, programmers have no control or knowledge of the underlying concurrency management upon which their application has to be built. For example, object-level locking, while providing basic mutual exclusion between simple method call, forbids atomic acquire of locks from multiple objects in a synchronized block [1]. Limitations of Java's concurrency

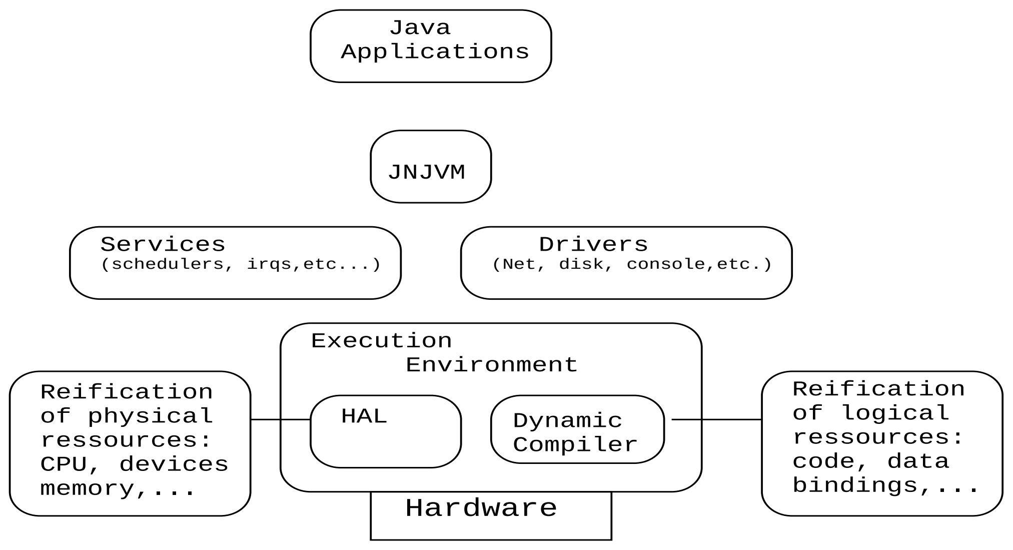
Java Applications
JNJVM
Services (schedulers, irqs,etc...)
Drivers (Net, disk, console,etc.)
Execution Environment
HAL
Dynamic Compiler
Hardware
Reification of physical ressources: CPU, devices memory,...
Reification of logical ressources: code, data bindings,...

Figure 3: Architecture of a dynamically adaptable Java runtime.

control mechanisms are further discussed in [2].

[7] identifies concurrency limitations related to excessive locking. Their multi-threaded web crawler spent 20% of its time in synchronization-related operations and only 1.5% after removing the “useless” locks from the Java core classes. This comes directly from a very conservative conception choice that is imposed on developers because of the rigidity of the system. As a result, converting an IP address into a string, with InetAddress.getHostAddress(), requires 27 lock acquisitions, most of which are unnecessary!

In addition, even the interactions between threads and memory have to be considered as indeterminate, since specifications for the memory model remain hard to interpret, are poorly understood, and thus are violated in most JVMs [23].

To illustrate the potential benefits of the VVM approach, we have implemented a Java VMLet, called JNJVM, on top of our minimal execution environment. The resulting Java runtime, while being still totally compliant with standard Java applications, is dynamically reconfigurable at both system and language levels. This includes the possibility to adapt concurrency control mechanisms at the application level or directly at the execution-environement level, in a completely transparent way for running applications.

3.1 JNJVM: a dynamically adaptable Java runtime

As illustrated in Figure 3, our Java environment, called JNJVM[4], relies on the YNVM and some basic services for thread creation/destruction or class loading. The JNJVM is compliant with the second edition of SUN’s specifications [10, 11]. Actually, it relies on three main components that are dynamically integrated into the YNVM: a class loader, a bytecode compiler, and an exception manager.

The generic class description, as its name indicates, describes the content of the class without any hypothesis on the higher level VMlet. It is a structure that enumerates the fields and the methods of the class. Both fields and methods have a symbol and a signature. Also,

[4]for JNJVM is Not a Java Virtual Machine.

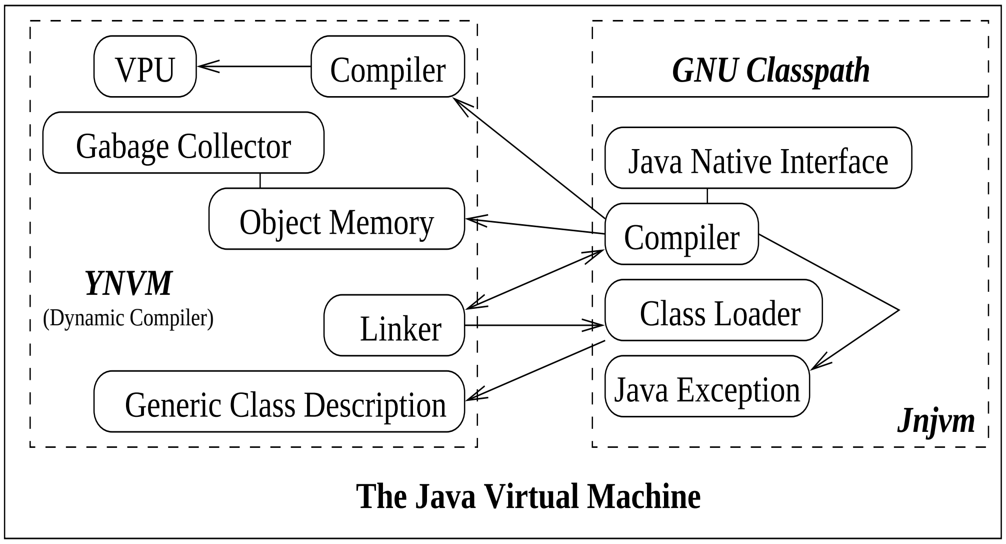

Figure 4: The JNJVM architecture

fields have an associated offset and methods have both a bytecode and an associated VMlet to compile this bytecode.

Using those two abstractions, the JNJVM implements a Java compiler, a Java Exception Manager and a Java Class Loader. Its internal architecture is represented in Figure 4. Some startup java classes come from the GNU Classpath Project [8].

3.2 The Java class loader

When the YNVM finds a class that is not already loaded (like the main class), it delegates the loading processes to the appropriate VMlet. The JNJVM then loads the Java class and fills the generic class description. The JNJVM possesses its own class description (the Java Class Description). This structure contains the java constant pool, which is the mapping between java symbols and numbers, and the java version of the class. The bytecode of a method is then stored in the YNVM's method description. Using the YNVM structures results in a very small Java Loader.

3.3 The compiler

The compiler is used when the JNJVM executes a method that has not yet been compiled. It transforms the bytecodes into an equivalent syntax-tree representation. For example, the op-code iload 22 is compiled in the syntax-tree (:vpu.ldi vpu 22). This representation corresponds to the default front-end language of the YNVM. Thus, the compiler uses the YNVM's internal code-generator to complete the transformation of bytecodes into native-code. Moreover, as stated before, VMlets using the YNVM's code generator are entirely platform-independent, thus portable.

Every opcode is compiled by a function and the resulting native code is inserted into a table of function pointers. Therefore, the modification of an opcode's behavior is equivalent to changing the contents of this vector of function pointers. For example, ignoring all the

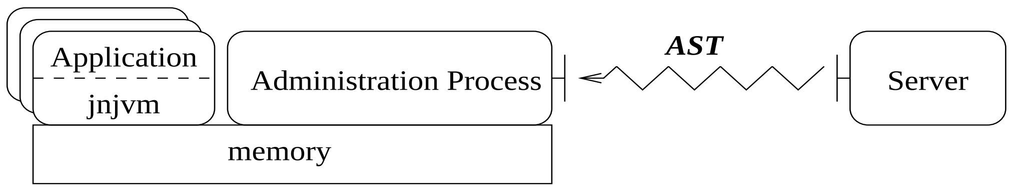

Figure 5: Remote administration of a Java runtime.

monitors opcodes (194 and 195) only implies replacing the entries 194 and 195 of the table by an empty function.

3.4 The language interface

The Java VMlet is implemented according to the VVM's approach. First, we built an environement dedicated to the constrtuction of Java environment (the Java VMlet itself), then we implement a dynamically flexible Java environment uppon this Java VMlet (called the JNJVM).

The Java VMlet is structured as a set of primitives, such as define-opcode to define how a given Java bytecode has to be compiled 3.3 or define-component to define a structure holding meta-data related to an element of the JNJVM (classes, attributes, methods,...). The JNJVM uses thoses primitives to implement a Java virtual machine. This acrhitecture allows an application to perform "in-depth" modification to the JNJVM.

3.5 The reflexive layer

The JNJVM exports a complete reflexive layer: any Java symbol is mapped to a YNVM symbol, which is used to call Java methods or inspect the value of any fields of classe or object from a YNVM script. Moreover, this mapping is used to modify any method's implementation: by dynamically re-binding the YNVM symbol to some code, we dynamically change the implementation of the associated method. For example, redefining the java.lang.Object.toString() java method is done by simply modifying a pointer in the method's structure associated with the :jnjvm.classes.java.lang.toString_signature VVM symbol.

This reflexive layer is directly accessible from within the YNVM, but it is going to be exported to Java applications through a Java class, encaspulating access to the underlying JNJVM via JNI.

3.6 Adaptability

Thanks to the language interface and the reflexive layer, the JNJVM achieves a high level of flexibility. A java application is embedded in a YNVM script, which we called an active script. This active script is responsible for loading and configuring a JNJVM, through the Java VMlet's primitives, according to the application's needs or constraints. Then, the script can open a "back-door" destined to receive, from an administrator, reconfiguration scripts.

As illustrated in Figure 5, the active script starts a thread listening for serialized ASTs from any input (file, network, keyboard, etc.). Those ASTs are then executed by the YNVM,

```
class YNVM {
    static public native int readEvalPrint(String);
};
```

Figure 6: A simple Java class reifying access to the YNVM through JNI

which allows remote administration of the Java runtime : such scripts can use the Java VM-let's primitives to reconfigure the JNJVM and the reflexive layer to modify the state of an application.

The main interest of this approach is to achieve dynamic flexibility without sacrifying performances: it allows a Java application to apply “in-depth” reconfigurations to its execution environment to match its needs, without prohibitive peformance overheads.

3.6.1 Performances

In order to evaluate our prototype, we compared it with two optimized JVM: SUN's JVM 1.3.1 (without jit compiler) and IBM's JVM 1.4.1 (with dynamic compilation). We used a G4 PowerPC, running a standard Linux distribution (Yellow Dog). Most of the benchmarks we used, shows that the JNJVM is nearly 3 times slower than it's IBM counterpart and 3 times faster than SUN's.

We found that most of the overhead induced by the JNJVM came from memory allocation and exception handling code, which are probably way too simple to compare with IBM optimized JVM. Garbage collecting, for example, is particularly inefficient (about 20 times slower than IBM's): since the YNVM's object memory is also used for storing Java classes, objects and meta-data, every garbage-collection involves a useless checking of the YNVM's internal objects (about 50.000 objects). Hence spliting the garbage collecting according to object's life cycle (and usage pattern) would considerably improve overall performance: YNVM's objects are created and destroyed only during dynamic compilation and loading phases, whereas Java objects life cycle is restricted to an application's execution.

Indeed, the JNJVM is entirely compiled by the YNVM, which produces native code as fast as optimized C. Moreover, the hooks used to modify the JNJVM are based uppon raw pointers to dynamic loading and dynamic compilation functions. The reflexive layer uses only meta-data structures related to the Java appplication, and thus does not introduce any indirection. Hence, our approach seems not to introduce severe performance penalties, but only an “in-depth” optimization work of our prototype would confirm it.

4 Application-level concurrency management

In order to demonstrates the interest of the VVM approach, we have used our flexible Java environment to illustrate how limitations of traditional Java environments can be overcome thanks to dynamic flexibility.

The dynamic adaptation of the JNJVM internals mechanisms, and in particular the concurrency management, can be achieved either by the execution of an external script at the YNVM level, or at the application level, through a Java package exporting the YNVM's interface.

```
import YNVM;
class Foo {
    void Foo() {
       YNVM.readEvalPrint("(set! :jnjvm.debug 255)");
    }
};
```

Figure 7: Using the YNVM to dynamically enable the debug mode

Since the YNVM allows the dynamic adaptation of anything built upon it, the simplest way to reconfigure a JNJVM, and thus a Java application, is to execute a script inside the YNVM. Depending on what kind of inputs are available on the host, the reconfiguration is either performed by loading a file (from a local disk or a remote filesystem), reading from the keyboard or executing a mobile agent. Thus, the reconfiguration of legacy applications is handled in a completely transparent way.

When the required adaptations are closely tied to the application's semantics, and hence embedded into it, reconfigurations need to be driven by the application and often imply application-level extensions. For those cases, we used the JNI (Java Native Interface) [9]. Since the YNVM produces C-compliant code, the JNI permits to interface any YNVM-level code with Java classes, thus to define a YNVM class that encapsulate the YNVM's interface, as illustrated in Figure 6. Thus any application can import this class and use it to adapt its execution environment through the underlying YNVM, while still being fully compliant with any Java compiler. Figure 7 represents a class using the previously defined YNVM class to dynamically turn the Java runtime into a debug mode.

As stated before, due to a very conservative concurrency management, many classes exhibit synchronized methods and thus excessive locking. To implement synchronization, every Java object has an associated lock. As with any object, a lock has a table of virtual methods (with two entries, lock and unlock). Hence, with the YNVM class that provides access to the underlying YNVM, any class can dynamically replace the virtual method table of any object's lock when no synchronization is needed. This reconfiguration can be undone the same way, by switching back to the old table. Further, this operation can be made recursive to really suppress any locking along the invocation path of a method.

Since it is designed to be independent of the underlying scheduler, the Java runtime does not manage concurrency. It relies on a pthread-like interface to create and destroy threads whose management is then left to the underlying system's scheduler. Hence developers have very poor control and knowledge about the way their application behave.

Through the YNVM class, an application can replace the thread-related functions used by the JNJVM by its own version, and then change the underlying scheduler, for example, by defining a new scheduling function and registering it in the Interruption Vector with the registerTrap primitive of the underlying minimal execution environment. We have experienced a simple multi-threaded application that replaces the default scheduler (a round-robin time-sharing) by a progress-based scheduling function. Basic thread creation and destruction functions are then encapsulated in new functions responsible to create and destroy per-thread additional data-structures (as the progress-metrics) and the scheduling function is just a loop to allocate some CPU time to the "slower" process, based on its average progression-speed. Thus, the progression-gap between the threads can be confined to a given threshold. This kind of scheduling is especially useful in multimedia systems to synchronize threads with different progression rate, such as audio and video codecs.

While the administrator can reconfigure, at any time, every aspect of the entire system, it may not be desirable to let any application do the same. Such security issues are handled in a flexible way too, as part of the configuration of the execution environment. By (re)configuring the loading mechanism, the administrator defines security rules to apply to applications. As opposed to a fixed set of predefined static and dynamic checks, the reflexivity of the YNVM allows to dynamically adapt the set of static checks to be performed when an application is loaded, for example to take into account a trust-level in its source, and to use the results of those static checks to determine and inline the dynamic checks that really have to be performed. Therefore, it is possible to restrict the possibilities offered to any applications, hence preventing misbehaving code to damage the system, in a completely flexible way.

5 Related Work

Dynamic adaptation is an active research area. Several flexible operating systems have been proposed, such as SPIN [24], VINO [5] or Exokernel [3], to allow application-based resource management. Therefore, they focus only on system aspects and most often mechanisms supporting dynamic adaptation are still fixed and rigid, as well as the security policies.

XVM [16] proposes a component-based extensible virtual machine that uses a dedicated language to describe a virtual machine's internals. Applications provide their extensions, as with the Exokernel approach, and the underlying virtual machine uses call-backs to replace an internal policy, such as garbage collector, by an application's extension. Although the virtual machine is extensible, its lack of reflexivity limits the flexibility to a predefined set of aspects.

Vanilla [15] uses a DSL-based approach for the construction of dedicated virtual machines. Elements such as parsers, type checkers and interpreters are described using a dedicated language. A Language Definition File identifies the components that need to be combined to obtain the desired virtual machine. While being well-adapted to the construction of dedicated/specialized virtual machines, it remains a static approach and the resulting environments are rigid and closed and thus lack flexibility.

Harissa [14] is a another DSL-based project. Its goal is Java program specialization through program transformation and especially partial evaluation, hence it focuses on adatation of an application to a specific context of utilization rather than on the definition of a flexible execution environment. A dedicated language is used to define specializations of "generic" classes and the specialization parameters, which can be static (evaluated at compile-time) or dynamic (evaluated at run-time). The result is an equivalent specialized C-program that can be compiled with gcc. This approach is obviously orthogonal and may be applied to the construction a specialized execution environment, matching some given application domain semantics.

Concerning Java, several projects of Java-based operating systems have been proposed to address traditional limitations of JVMs (resources control, isolation, etc.), such as GVM [25], a monolithic Java-based kernel providing complete resources isolation between processes, Alta [26], a micro-kernel based JVM extended to support hierarchical scheduling, J-Kernel, a Java-based kernel, using device drivers to account resources usage, SUN's JavaOS, a Java runtime based on the Chorus micro-kernel to provide some realtime support to applications or JX-Kernel [12]. Their objective remains adding some features to overcome some limitations, thus it is still an “ad-hoc” and rigid solution to a dedicated problem, since those features, as well as the entire resulting system, are not meant to be flexible. Hence, applications are stuck again in a rigid “one-size-fits-all” based execution environment.

Bossa [29] rely on a domain-specific-language (DSL) based approach and thus proposes a language for describing new scheduling policies. Specialized schedulers are then generated and can be loaded through a (Linux-)kernel module. This work is obviously orthogonal with ours and certainly very complementary.

6 Conclusions and perspectives

This paper presented the architecture of a flexible execution environment, based on a HAL and a flexible dynamic compiler, free of any predefined, imposed abstraction. This high-level of dynamic flexibility allows dynamic construction of dedicated execution environments as well as their dynamic reconfiguration. To demonstrate the benefits of this approach, we used this minimal execution environment to build the JNJVM, a dynamically adaptable Java runtime.

The inherent flexibility of the JNJVM allows to address some of the limitations of traditional rigid Java runtimes, in particular concurrency management and concurrent programming support.

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 HAL part of the minimal execution environment is currently being embedded in a Linux kernel-module so that our flexible Java runtime can be used on top of a traditional operating system, as a standard JVM, while preserving a maximum of flexibility.

Concerning the JNJVM, we plan to develop further our flexible Java environment toward a complete flexible Java-OS and also to integrate it in the context of component based middleware, as the Corba Component model, to illustrate the adequacy of our approach to the needs of moderns distributed applications.

References

[1] P. Felber and M. Reiter. *Advanced concurrency control in java*, Technical report, Bell Labs Research, Jan. 2002. `http://citeseer.nj.nec.com/felber02advanced.html`

[2] D. Lea. *Concurrent Programming in Java*, Second Edition. Addison Wesley Longman, November 2000.

[3] M.F. Kaashoek, D.R. Engler, J. O’Toole. *Exokernel: an operating system architecture for application-level resource management* Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain, Colorado, December 1995.

[4] B. Folliot. *The Virtual Virtual Machine Project*, Proceedings of IFIP Symposium on Computer Architecture and High Performance Computing, Sao Paulo, Brasil,October 2000.

[5] M. Seltzer, Y. Endo, C. Small and K.A. Smith. *Dealing with Disaster: Surviving Misbehaved Kernel Extensions*, in Proceedings of the 2nd Symposium on Operating Systems Design and Implementation, Seattle, Washington, pp 213–227, 1996. `citeseer.nj.nec.com/seltzer96dealing.html`

[6] J.P. Fassino and J.B. Stephani. *THINK : un noyau d'infrastructure répartie adaptable*, Proceedings of CFSE 2, Paris, France, April 2001.

[7] A. Heydon and M. Najork. *Performance limitations of the Java core libraries*, in Concurrency: Practice and Experience, vol. 12, number 6, pp 363–373, 2000. `http://citeseer.nj.nec.com/heydon00performance.html`

[8] GNU Classpath Project, `www.gnu.org/software/classpath/classpath.html`

[9] JNI Spec version 1.4, `http://java.sun.com/j2se/1.4/docs/guide/jni/index.html`

[10] J. Gosling, B. Joy, G. Steele and G. Bracha. *The Java(TM) Language Specification, Second Edition*, Paperback, 1996.

[11] : T. Lindholm and F. Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition)*, Paperback.

[12] M. Golm, M. Felser C. Wawersich, and J. Kleinder. *The JX Operating System*, 2002 USENIX Annual Technical Conference, June 10-15, 2002, Monterey, CA, pp. 45-58

[13] The RealTime for Java Expert Group. `jcp.org/jsr/detail/1.jsp`

[14] G. Muller and U.P. Schultz, *Harissa: A Hybrid Approach to Java Execution*, in IEEE Software, March/April 1999.

[15] S. Dobson, P. Nixon, V. Wade, S. Terzis and John Fuller, *Vanilla: An Open Language Framework*, in Lecture Notes in Computer Science, Volume 1799, 2000.

[16] T. Harris, *An extensible virtual machine*, in Proceedings of the Simplicity, Performance and Portability in Virtual Machine Design Workshop (OOPSLA), Denver, Colorado, November 1999.

[17] M. Golm, *Design and implementation of a meta architecture for Java*, Master's thesis, University of Erlang, January 1997.

[18] R. Fitzgerald, T.B. Knoblock, E. Ruf, B. Steensgaard and D. Tarditi, *Marmot: An optimizing compiler for Java*, in Software Practices and Experiences 30(3), March 2000.

[19] M.P. Atkinson, L. Daynes, M.J. Jordan, T. Printezis and S. Spence, *An Orthogonally Persistent Java*, in ACM Sigmod Record, Volume 25, Number 4, December 1996.

[20] ISO/IEC. "Open Distributed Processing - Reference Model, Part 2 : Fundations". ITU-T Recommendation X.902 International Standard 10746-2, ISO/IEC, 1995.

[21] ISO/IEC. "Open Distributed Processing - Reference Model, Part 3 : Architecture". ITU-T Recommendation X.903 International Standard 10746-3, ISO/IEC, 1995.

[22] ISO/IEC. "Open Distributed Processing - Reference Model, Part 1 : Overview". ITU-T Recommendation X.901 International Standard 10746-1, ISO/IEC, 1998.

[23] W.Pugh. *The Java Memory Model is Fataly Flawed*, in Proceedings of ACM Java Grande Conference 1999, San Francisco, California, June 12-14, 1999

[24] 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* in Proceedings of the 15th ACM Symposium on Operating System Principles, Copper Mountain, Colorado, December 1995.

[25] G. Back, P. Tullman, L. Stoller, W. Hsieh, J. Lepreau. Java Operating Systems: Design and Implementation, University of Utah Technical Report, UUCS-98-015, August 1998.

[26] P. Tullmann and J. Lepreau. *Nested Java Processes: OS Structure for Mobile Code*, in Proceedings of the Eighth ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.

[27] C. Hawblitzel, C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. *Implementing Multiple Protection Domains in Java*, in Proceedings of the 1998 Usenix Annual Technical Conference.

[28] B. Folliot, I. Piumarta, L. Seinturier, C. Baillarguet, C. Khoury, A. Lger, F. Ogel. *Beyond flexibility and reflection: the virtual virtual machine approach*. NATO Advanced Research Workshop, Environments, Tools and Applications for Cluster Computing. LNCS 2326, Springer-Verlag, pp. 17-26, 2002

[29] J. Lawall, G. Muller, L.P. Barreto. *Capturing OS expertise in an Event Type System: the Bossa experience*, in Proceedings of the ACM SIGOPS European Workshop 2002 (EW'2002), Saint-Emillion, France, September 2002.