Alternative schemes for low-footprint operating systems building

Christophe Rippert, Damien Deville, Gilles Grimaud

Alternative schemes for low-footprint operating systems building

Christophe Rippert — Damien Deville — Gilles Grimaud

N° 5220

June 2004

Thème COM

ISSN 0249-6399 ISRN INRIA/RR--5220--FR+ENG

Alternative schemes for low-footprint operating systems building

Christophe Rippert [*] , Damien Deville [*] , Gilles Grimaud [*]

Thème COM — Systèmes communicants
Projet POPS

Rapport de recherche n° 5220 — June 2004 — 12 pages

Abstract: This paper presents two ways of building dedicated embedded operating systems. The constructive approach consists in starting from a minimal kernel and adding abstractions as they are needed, whereas the destructive approach promotes the idea of customizing an existing operating system by removing unnecessary abstractions. We compare these two approaches on the example of building an embedded Java operating system and discuss the pros and cons of each method. We conclude by exhibiting the weakness of each approaches concerning dynamic adaptation of the system.

Key-words: System customization, embedded system, JavaOS

[*] IRCICA/LIFL, Univ. Lille 1, UMR CNRS 8022, INRIA Futurs, POPS research group. This work is partially supported by grants from the CPER Nord-Pas-de-Calais TACT LOMC C21, the French Ministry of Education and Research (ACI Sécurité Informatique SPOPS), and Gemplus Research Labs.

Deux approches pour la construction de systèmes d'exploitation embarqués

Résumé : Ce document présente deux approches pour la construction de systèmes d'exploitation embarqués. L'approche constructive consiste à partir d'un noyau minimaliste auquel on ajoute les abstractions systèmes requises, alors que l'approche destructive se base sur l'idée de personnaliser un système existant en retirant les abstractions inutiles. Nous comparons ces deux approches en prennant l'exemple d'un système d'exploitation Java embarqué, et évaluons les avantages et inconvénients de chaque approche. Nous concluons en montrant les limitations des deux approches en ce qui concerne l'adaptation dynamique du système.

Mots-clés : Personnalisation de système, système embarqué, système Java

1 Introduction

Embedded operating systems dedicated to resource-constrained devices must take into account the strict limitations of the platform on which they run. Typical constraints include a very limited amount of available memory space (4KB of RAM on a standard smart card for instance) and slow processors. Thus, they need to be fully customizable so as to best exploit the resources available and not waste them with unnecessary services. This advocates a minimal kernel architecture including only abstractions which are deemed indispensable for all platforms, and an extensible system architecture to add platform-specific services. The exokernel architecture [1] seems thus perfectly suited for these types of systems. However, an alternative approach would be to start from an existing operating system and remove unnecessary services. This approach, that we shall call *destructive* (or *downsizing*) in contrast to the *constructive* approach advocated by the exokernel architecture, permits to benefits from existing (and possibly already optimized) services without having to write most of the system from scratch. This destructive approach is more widespread in the industry where projects often have short term deadlines, whereas the constructive approach has been strongly advocated over the last few years by the academic world [1, 2, 3]. Thus, we deem interesting to compare those two approaches and find which one seems the most appropriate to build a fully-customized operating system for resource-limited devices.

We chose to try and build an embedded Java operating system using both approaches, to evaluate their pros and cons on a concrete example. The choice of a JavaOS for an embedded system might seem surprising considering the reputation of bulkiness of Java but it should be remembered that this language was initially defined for memory-limited devices, as illustrated by the compactness of its bytecode. Java Card [4] is an example of a successful JavaOS for smart cards, which are one of the most constrained systems. We used the Camille exokernel to test the constructive approach and the Jits JavaOS architecture to evaluate the downsizing approach. We first present the Camille exokernel, then we describe the Jits architecture. We then compare the advantages and drawbacks of each approach, before concluding and presenting future work.

2 Camille

Camille is an extensible operating system designed for resource-limited devices, such as smart cards for instance. It is based on the exokernel architecture [1] and advocates the same principle of not imposing any abstractions in the kernel, which is only in charge of demultiplexing resources. Camille provides secure access to the various hardware and software resources manipulated by the system (*e.g.* the processor, memory pages, native code blocks, etc) and enables applications to directly manage those resources in a flexible way.

System components and applications can be written in a variety of languages (including Java, C, etc). The source code is translated in a dedicated intermediate language called Façade [5] by appropriated tools (*e.g.* a Java to Façade converter or the Gnu Compiler

Collection for C code). Using an intermediate language enhances the portability of the various components is a way similar to Java bytecode. This portability is strengthened by the very limited number of machine-dependent system primitives, which can easily be ported from one platform to another and are grouped in a component called the Base system. To guarantee the efficiency of the system and the applications, the FAÇADE code is translated into native code using an embedded compiler. This compiler converts FAÇADE programs when they are loaded in the device, and performs machine-dependent optimizations to exploit fully the underlying hardware.

Writing an operating system with CAMILLE consists in assembling system abstractions programmed as components. In CAMILLE, everything is represented as an object, including the underlying hardware which is reified by appropriate interfaces. Thus the programmer builds his operating system by manually assembling the bricks which he has programmed. The safety of this composition is guaranteed by including a proof of correctness in each system extension, which is checked when the extension is loaded in the exokernel. A more detailed presentation of CAMILLE can be found in [6], including benchmarks and experimental results.

Building a JavaOS with CAMILLE can be done in two ways. First, the programmer can provide a bytecode-to-FAÇADE converter which translates the bytecode of classes before loading them into the system. This permits to verify the code and include in it the proof of its correctness. To support Java functionalities, the embedded operating system must include a Java API and basic Java services as a garbage collector for instance. Since the code of classes is translated from FAÇADE to native code when loaded on the card, the system does not need a bytecode interpreter to execute Java programs.

A second approach is to implement a complete Java virtual machine in the system and to load standard `.class` file in CAMILLE. This prevents the type-checking done at compile time by the Java to FAÇADE converter, but the code is still verified when loading the classes by the type-checker included in the Java virtual machine. This induces more work as the programmer must write a full Java runtime environment including a virtual machine and a whole Java API, but permits to dynamically load standard classes without going through a heavy compilation chain. For instance, the system can download applets from the Internet and use them directly. Both approaches require significant work from the programmer who must implement at least a whole Java API and basic Java services.

3 JITS: Java In The Small

JITS is an architecture dedicated to build embedded Java operating systems for resource-limited devices. JITS provides a full-featured Java virtual machine optimized to minimize its runtime memory footprint. It also includes a complete Java API from which the system programmer can extract the packages he needs for his applications and which will be embedded with the virtual machine. Thus JITS permits to build a fully-tailored Java runtime environment without sacrificing functionalities of the virtual machine, as it is often the case in most embedded environments [4].

JITS includes a tool called the romizer, which task is to generate the image of the embedded environment. The romization process consists in loading and initializating all needed classes using JITS class loading mechanism, to build the runtime structures of classes and objects. The romizer then takes a snapshot of these structures and dump them to a C file which will be compiled with the core of the virtual machine to build the runtime environment. The romizer is a standard Java application which can be executed on any virtual machine, which differs from most other romization processes which impose a dedicated building environment.

JITS class loading mechanism aims at reducing as much as possible the runtime memory footprint of loaded classes. Memory space is saved by dropping constant pool entries which are not necessary after the class has been initialized. The constant pool is thus packed on-the-fly during the class loading process, which also reduces the memory consumption of the loading process itself. This compacting class loading mechanism does not reduce the functionalities of the virtual machine, including type-checking which can still be ensured when a new method is loaded.

Experiments have shown that this class loading mechanism permits to suppress more than 83% of constant pool entries included in the original `.class` file. This induces a reduction of over 73% of the size of the constant pool if introspection is supported and more than 92% if support for introspection is not included in the virtual machine. The gap between those two value is due to the fact that introspection requires to keep metadata (mostly class, field and method names, and method descriptors) which are represented as UTF8 strings in the constant pool and therefore use a lot of space [7].

JITS advocates a downsizing philosophy, by providing a complete Java API and permitting the system programmer to choose which package he needs to include in his embedded environment. But it goes beyond simply choosing which classes will be needed at runtime, by enabling full customization of the virtual machine itself. Starting from a full virtual machine, the programmer can configure it to reflect the specificities of the platform on which it will run. For instance, he can disable support for floating point or 64-bit arithmetic. Another step is to customize the virtual machine, by removing unnecessary components. If the embedded application does not use dynamic allocation of memory (which is quite likely on very limited platforms as Java Card for example), the virtual machine does not need to include an object memory manager and the programmer can save some memory space by not linking the object memory manager component to the virtual machine when building it. In that example, the engine which interprets the bytecode can also be specialized by not supporting the bytecode responsible for dynamic memory allocation (*e.g.* `new`, `newarray`, etc) and thus reducing the size of the engine. Finally, JITS includes an ahead-of-time compiler, which was at first developped to optimize performances of methods often used by the API or the applications. This tool can be used to translate the application entirely to native code, thus suppressing the need for a Java virtual machine. This represents the ultimate step in customization of the virtual machine, by dedicating it completely to the application.

JITS permits to statically customize the virtual machine and API, by suppressing unnecessary components and packages. But it also permits dynamic adaptation using Java dynamic class loading mechanism. Thanks to the low memory footprint class loading process implemented in the virtual machine, the embedded runtime environment can be dynamically extended to provide new functionalities to the applications, or to load new applications on the device. If the `java.lang.reflect` package is included in the embedded environment, applications or APIs can be dynamically reconfigured using JITS `Adaptation` class which provides methods to modify constant pools or add new methods to existing classes for example.

4 Evaluation

4.1 Modularity

Both approaches favor modular programming to ensure flexibility. With the constructive approach, the programmer is encouraged to develop each needed service independently and clearly separate interface specification from the implementation of the components, to ease the addition of each module as a new brick in the system. Similarly, the destructive approach requires that modules are as independent from each other as possible. More precisely, components can depend on each others interfaces but not on their implementation, to permit removing (or modifying) a module without having to modify the implementation of others. For instance, removing support for dynamic memory allocation in JITS can be done easily since all modules calling it (*i.e.* mainly the bytecode interpreter which executes the instructions `new`, `newarray`, etc) use the methods specified in its interface and do not depend on its internal data[1]. Modularity is also ensured by type-checking which prevents accessing directly internal component structures. In JITS for instance, the bytecode verifier guarantees that all access to class fields are done using `getfield` and `putfield` instructions, and not by forging a reference to the field (which could be attempted for example by pushing an int on the operand stack and popping it as an array). In CAMILLE, the proof loaded with each extension ensures that the code of the extension makes only authorized memory accesses.

4.2 Granularity

Both approaches provide flexibility, but at different levels. Using the constructive approach, the system programmer can fully customized modules whereas the downsizing approach only permits to choose which modules are needed but not to tailor them. The granularity of customization is therefore much smaller using the constructive approach, since it can go down to the level of methods included in a component.

For instance, using CAMILLE, the developer building a Java Card compliant environment does not need to program any unallocation method in his memory manager since the Java Card runtime environment does not include any garbage collection mechanism. Thus, his memory manager component can be smaller than JITS object memory manager which implements a full garbage collector. A counterpoint could be to argue that the memory manager

[1]On this particular example, removing the dynamic memory manager means replacing it by an empty implementation, since it is written in C and the virtual machine could not be linked otherwise.

component should be split in two modules, one managing memory allocation and the other providing garbage collection, which would permit to include only the allocator if needed. However, programming very small components usually poses a problem of efficiency. In our example, the module including the garbage collector would probably use many internal structures of the allocator module (as the memory map for example), which would force the programmer to implement methods in the allocator component to access these structures, thus degrading the performances of the garbage collector (and raising a confidentially issue since internal structures of the memory manager are not usually meant to be accessible from other components).

Similarly, a programmer implementing a Java API in CAMILLE can decide not to implement the `toString` methods included in every objects in Sun's specification and which are mainly used for debug traces. On the other hand, JITS offers no support for the developer to suppress methods inside a provided class. Of course, the programmer can modify the classes to suppress the `toString` methods manually, but this would be rather time-consuming to parse all classes and handle dependencies to the suppressed methods.

4.3 Expertise

The constructive approach consists in building the whole system from scratch, which requires lots of work and time. On the other hand, the destructive approach speeds up the development of a running environment since all needed components are already provided and all the system programmer has to do is suppress unnecessary features. This is obviously much faster than having to write all needed abstractions and API.

Writing efficient embedded operating systems requires expert programmers, as classic optimizations done in standard systems are not always relevant for embedded systems. Typically, most traditional operating systems are optimized to execute faster, while resource-limited platforms require a system optimized to consume as little memory space as possible. This strongly impacts the way the developer programs his components and force him to hunt for memory-costly code parts. For instance, a typical "`for (int i = 0; i < string.length(); i ++) // do something`" statement would probably be optimized in a classical operating system by using a variable to store the length of the string and avoid a method invocation for each iteration. Similarly, function inlining is a typical optimization which can result in significantly increasing the code size. These optimizations are not relevant in embedded systems, and the programmer might even try and apply the opposite procedure (*i.e.* suppressing dispensable variables and *outlining* code by creating methods to factorize duplicated sequences of instructions). These specific requirements imply expert programming from the system builder if he has to write himself all the services he needs to include in the system. On the other hand, using the downsizing approach permits to benefit from a set of components already optimized for embedded systems from which the system builder can choose the ones he needs.

Using the destructive approach also permits to benefits from tools provided by the developer of the architecture for which the system is built. For instance, the architecture can include a composition tool permitting to automatize the assembly of the system. Work is currently being conducted in JITS to provide a configuration tool permitting to obtain a graphic view of all provided modules and displays their dependencies. Using this tool, the system builder can uncheck components he does not want to include in the runtime environment and see which modules are automatically disabled because of their dependency to the unselected components.

Similarly, the system developer can profits from a tool managing the deployment of the environment on the target platform. This kind of tools are very useful since deploying a JavaOS is usually much more complicated on an embedded system than on a workstation. For instance, deploying a Java Card environment can be difficult since smart cards usually integrate different types of memory (typically RAM as working space, EEPROM as a storage area, and ROM for immutable parts of the system). Considering the very different characteristics of each type of memory[2], it is usually tricky to choose which parts of the environment should be located in each zone. Using CAMILLE, this hard work must be done by the system developer whereas JITS integrates a tool called the Romizer which task is precisely to generate the binary image of the runtime environment, and help the programmer loading its different parts in the appropriate memory zones. This tool was obviously hard to develop, but it can be used easily by the system programmer who does not need to have any knowledge of the memory layout in the target device.

Finally, the downsizing approach also permits to tests various system configurations by building virtual machines including different components until the suitable functionalities/size ratio is found. Thus the programmer can directly measure the cost of adding services and know which parts of the runtime environment consume most memory. With a constructive approach, the programmer would have to first write a service before being able to evaluate its cost, and then perhaps discard it for being to costly.

For instance, enabling or disabling support for introspection is immediate in JITS since it is simply an option which can be set as needed. If introspection is disabled, the class loading mechanism strips name and type information from classes as they are loaded, thus preserving memory by compacting the constant pool as explained in Section 3.

Similarly, the system builder can try and include a floating-point library to quantify the memory consumption it would induce, then decide it would be too costly and choose not to provide support for floating-point arithmetic. In CAMILLE, the system developer would have to write the whole floating-point library from scratch before being able to evaluate its memory footprint.

4.4 Synthesis

Clearly, the main drawback of the constructive approach lies in the lack of support for the system builder. An interesting solution lies in flexible system architectures [8, 9] which combines the flexibility of a minimal kernel with libraries of system services which can be used by the system developer as needed. For instance, the THINK architecture is based on a

[2]Basically, RAM is very fast but volatile and usually very small due to technical constraints, EEPROM is persistent but much slower than RAM and its life expectancy decreases with each write, and ROM is persistent but really slow and of course cannot be modified once burned.

nano-kernel providing access to the hardware without adding any abstractions, and provides a complete library of drivers and system services. These abstractions are programmed as components to enable the system builder to add only those he needs in his system and to replace any that does not suit his needs by its own. These system architectures thus combine both approaches, since the programmer starts from a minimal kernel and add modules extracted from an existing library which has been programmed by an expert. Moreover, THINK provides a set of tools to speed up the building of the system, as a configurator which permits to choose which modules to include in the system and manage component dependencies.

However, these architectures do not provide a granularity of composition as small as the constructive approach by itself. In an architecture as THINK, the programmer cannot choose to include only a part of a component if he does not need all the functionalities it provides. He can always modify the code of the component to manually remove unneeded features, but that implies once more to be an expert to fully understand how the component works and remove superfluous parts while preserving the others. Components provided in the library can be programmed to ease their customization[3], however this kind of static support for customization cannot compete with the control the programmer can benefit when writing his component from scratch. For instance, a programmer can easily comment out the `free` function of a memory manager provided in the library, but it is much more difficult to analyze the internal structures of the memory manager and figure out which ones are only necessary for unallocation and can be suppressed.

5 Conclusion and future work

As we have seen in this paper, programmers building systems for embedded devices can profit from both the constructive and the destructive approaches. Both approaches favor modular programming to ensure flexility. Building a system from the bottom up permits to customize the system at a very small granularity since the programmer implements only the functionalities he needs in the modules composing his system. Removing services from an existing system speeds up the construction of a running system and permits to benefits from the expertise of the programmer of the base system which is materialized in the optimized services and tools provided. Flexible system architectures as THINK combine both approaches by providing libraries of optimized services and tools easing system construction. However, they do not offer the same granularity of composition as the constructive approach.

In this paper, we have focused our study on the static flexibility of embedded systems, to permit their customization at build time. However, embedded operating systems require more than just static customization. Their inherent mobility calls for dynamically reconfigurable operating systems supporting the runtime addition or subtraction of functionalities. Dynamically adding new services to a running system is a problem which has been addressed

[3]A very basic customization facility could consist for instance in using `#define` macros to permit to select which parts of the component should be compiled.

by lots of previous work. For instance, JITS supports dynamic loading of classes through its compacting class loading mechanism as presented in Section 3, and CAMILLE allows safe dynamic loading of system extensions. On the other hand, dynamically removing functionalities is a much harder issue (not to speak of dynamically replacing existing modules by "equivalent" ones).

Basically, removing a service implies disabling all bindings from other parts of the system to the removed module. In a JavaOS, a binding between two classes is essentially either a method invocation or an access to a public field. The problem comes from the fact that information concerning a binding are stored in the source module, which implies parsing all modules in the system to modify those using the removed one. For example, a Java method accessing a public field of another class gets all information about the targeted field from the constant pool of its own class. An immediate solution would be to store in each component information about which other components are bound to it, thus creating a "backward link". This could be done during linking of the various modules. However, this would generate additional metadata in each component, which would increase the memory footprint of the system and make it unsuitable for very small embedded systems as a smart card for instance. In JITS for instance, we are trying to reduce as much as possible the footprint of the constant pool, mainly by suppressing metadata needed only during the load and link phases. Including binding information for each class would compromise this effort and result in classes too big to be embedded.

An alternative approach is to use proxies to intercept all inter-module communications. In a component-based operating system, modules can be encapsulated in containers which offer interfaces to access methods and fields of contained components. This is the approach advocated in the THINK architecture to permit dynamic replacement of components [10]. However, this kind of architectures usually suffers from performances degradation due to the overhead of intercepting every access.

References

[1] D. R. Engler, M. F. Kasshoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proc. of SOSP'95.*

[2] B. N. Bershad, C. Chambers, S. J. Eggers, C. Maeda, D. McNamee, P. Pardyak, S. Savage, and E. G. Sirer. SPIN — An Extensible Microkernel for Application-specific Operating System Services. In *Proc. of the 6th ACM SIGOPS European Workshop*, 1994.

[3] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with Disaster: Surviving Misbehaved Kernel Extensions. In *Proc. of OSDI'96.*

[4] Z. Chen. *Java Card Technology for Smart Cards: Architecture and Programmer's Guide.* Addison Wesley, 2000.

[5] G. Grimaud, J.-L. Lanet, and J.-J. Vandewalle. FAÇADE: A Typed Intermediate Language Dedicated to Smart Cards. In *Software Engineering — ESEC/FSE*, number 1687, pages 476–493. Springer-Verlag, 1999.

[6] D. Deville, A. Galland, G. Grimaud, and S. Jean. Smart Card operating systems: Past, Present and Future. In *The 5th NORDU/USENIX Conference*, 2003.

[7] Christophe Rippert, Alexandre Courbot, and Gilles Grimaud. A low-footprint class loading mechanism for embedded java virtual machines. In *Proc. of PPPJ'04*.

[8] C. Rippert and J.-B. Stefani. Building secure embedded kernels with the think architecture. In *Proc. of ECOOSE'02*.

[9] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. In *Proc. of SOSP'97*.

[10] A. Senart, O. Charra, and J.-B. Stefani. Developing dynamically reconfigurable operating system kernels with the think component architecture. In *Proc. of ECOOSE'02*.

Contents

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399