Hardware Mediators: A Portability Artifact for Component-based Systems

Fauze Valério Polpeta[1] and Antônio Augusto Fröhlich[1]

Federal University of Santa Catarina, PO Box 476
88049-900, Florianpolis - SC, Brazil
{fauze,guto}@lisha.ufsc.br,
http://www.lisha.ufsc.br

Abstract. In this article we elaborate on portability in component-based operating systems, focusing in the *hardware mediator* construct proposed by Frhlich in the *Application-Oriented System Design* method. Differently from hardware abstraction layers and virtual machines, hardware mediators have the ability to establish an interface contract between the hardware and the operating system components and yet incur in very little overhead.
The use of hardware mediators in the Epos system corroborates the portability claims associated to the techniques explained in this article, for it enabled Epos to be easily ported across very distinct architectures, such as the `H8` and the `IA-32`, without any modification in its software components.

1 Introduction

Portability has always been a matter for operating system developers, because the very own nature of an operating system has to do with abstracting hardware components in a way that is suitable for application programmers to develop "architecture-independent software". It is expected that an application developed on top of a chosen operating system will run unmodified in all architectures supported by that operating system. Therefore, operating systems constitute one of the main pillars of applicative software portability.

Traditional approaches to make the operating system itself portable are mainly concentrated in two flanks: *Virtual Machines* (VM) and *Hardware Abstraction Layers* (HAL). While considering the virtual machine approach to operating system portability, one cannot forget that the virtual machine itself is part of the operating system—according to Habermann, the operating system extends from the hardware to the application [10]. The virtual machine would thus constitute the architecture-dependent portion of the operating system, while granting portability for the components above. The main deficiencies of this approach is the overhead of translating VM operations into native code. Several "JAVA operating systems" rely on this approach to achieve portability.

A second major alternative for operating system portability is based in hardware abstraction layers, which constitute an architecture-dependent substratum for the development of system software. A HAL encapsulate hardware-specific details under a software-oriented interface. Although usually considered not to incur in as much overhead as virtual machines, hardware abstraction layers must rely on refined implementation techniques to achieve good performance.

One additional shortcoming of both approaches arises from design. Without a proper domain engineering strategy, it's very likely that VMs and HALs will incorporate architectural details of the initial target architecture(s), making it difficult to adapt them to other architectures. That's probably the reason why ordinary all-purpose operating systems designed around a complex memory management system, such as Unix and Windows, look like a Frankstein when ported to 8-bit architectures.

Domain engineering methodologies that drive the design process toward collections of reusable software components are largely used in the realm of applicative software. Recently, strategies to develop component-based operating systems begun to appear [2, 4, 8] and are producing exciting new operating systems such as Epos [9] and Pure [16]. Being fruit of a domain engineering process (instead of a system engineering process), the software components of such systems can be arranged to build a large variety of run-time support systems. More specifically, the *Application-Oriented System Design* (AOSD) method proposed by Frhlich [8] combines principles of *Object-Oriented Design* (OOD) [3] with *Aspect-Oriented Programming* (AOP) [12] and *Static Metaprogramming* (SMP) [5] in a method that guides the development of highly adaptable software components for the operating systems domain.

This new class of application-oriented operating systems has the same need for portability as their more conventional relatives, however, the combination of AOP and SMP brings about new opportunities to implement VMs and HALs: *hardware mediators*. A hardware mediator is a software artifact that encapsulates a hardware component in a construct whose interface has been defined in the context of operating systems. This concept resembles HAL elements, but the use of AOP and SMP enables hardware mediators to be far more flexible and yet present better performance than traditional HALs.

This paper discusses the use of AOSD's hardware mediators as a powerful operating system portability tool. After presenting a parallel of the techniques commonly used in the realm of operating systems to achieve portability, the hardware mediator concept is explained in details, followed by a case study of its deployment in the Epos project. The paper is closed with a discussion of related works and author's perspectives.

2 Portability in Ordinary Operating Systems

Operating systems, as discussed in the introduction of this paper, are one of the main artifacts to promote applicative software portability, as they hide architectural dependencies behind standardized interfaces such as Posix. A properly designed operating system enables applications to endure the quick evolution of computer hardware without major impact. Consequently, being able to quickly port an operating system to a new hardware platform became an strategic issue for the software industry.

Yet in the 70's, the VM/370[1] operating system from IBM [6] was strongly concerned with portability. In order to enable batch applications developed for older systems to execute in the new environment, IBM opted for introducing multitasking in the VM/370 by means of a virtual machine scheme that delivered each application a duplicate of the real hardware. Nevertheless, since most of the virtual machine instructions were indeed real machine instructions executed, this approach prevented the system from being ported to other architectures.

The concept of virtual machine, however, goes far beyond the scheme introduced by VM/370. As a matter of fact, any software layer aimed at extending the functionality, or raising the abstraction level, of a computer system can be taken as a *virtual machine* [19]. That thinking could lead us to conclude that the simple choice of an universal programming language—such as C, for which numerous cross-compilers are available—to implement the operating system could respond for all portability matters. This is definitely not true: first, because high-level programming languages do not feature all the operations needed by the operating system to interact with the hardware, forcing system programmers to write native code (e.g. *assembly*) that cannot be automatically translated to new architectures; second, because device drivers are usually very platform-specific and cannot be automatically converted too.

Even if programming languages alone cannot be accounted for operating system portability, they are a crucial means. By taking on a portable programming language and gathering all architecture-dependent code in a self-contained *hardware abstraction layer*, operating system engineers have an option for developing portable systems. The original Unix [18] from AT&T Bell Labs was one of the first operating systems to use this approach. As described by Miller [14], porting Unix from the PDP to the Interdata was a straightforward activity mostly concentrated on the HAL implementation. This strategy for portability is nowadays adopted by many operating system, including Unix descendants and also Windows.

Recent advances in both approaches are well represented by Java VM on the side of virtual machines and by Exokernel on the side of hardware abstraction layers. On the one hand, systems like the JavaOS [13], developed by Sun Microsystems, have promoted the Java VM as an attractive system portability tool but, not differently from other VM-based systems, the VM must be reimplemented for every new platform. On the other hand, Exokernel [7] eliminated the notion of abstractions from the operating system kernel. However, the diversity of devices in each hardware platform imposes severe restriction on the definition of interfaces for the Exokernel, and therefore compromising its portability [11].

Nevertheless, both approaches, HALs and VMs, are becoming too restricted to match contemporary software engineering techniques. As a matter of fact, the design of traditional portability artifacts like HALs and VMs is usually

[1] The technical literature often refers to IBM's VM/370 as CP/CMS.

driven by the necessity of making the resources available in a given hardware platform to a given operating system. However, binding the design process to a preexisting hardware platform or operating system makes room for unnecessary dependencies that most likely will restrain both, reuse and portability. In order to understand how such dependencies grow up in the system, let us consider the well-know memory management scheme [1] used by Unix. The `brk` system-call in Unix can be used by ordinary processes to modify the size of its data segment. More notoriously, it is used by `libc`'s `malloc` an `free` functions to manage a process' heap. The implementation of this system-call presupposes a paging memory management strategy supported by an MMU. Indeed, implementing `brk` without an MMU is unpractical, for it would imply in dynamic process relocation. Consequently, Unix's HAL includes a paging engine abstraction. This design seems reasonable for a multitasking operating system, but it severely compromises its portability to a platform that does not feature a MMU[2].

Eliminating architectural dependencies of this kind, that extend through the system from HAL to API, is fundamental for systems compromised with portability and reusability. In particular, the embedded system realm, which today accounts for 98% of the processors in the market [17], cannot go with restrictions like this. Moreover, embedded systems often operate on restricted resources and monolithic VMs and HALs are likely to overwhelm the system. In this scenario, a component-based HAL whose components can be selected and adapted according to application's demands is certainly a better choice. The next section introduces a novel strategy to achieve portability in component-based run-time support systems.

3 Hardware Mediators: a Portability Artifact for Component-based Operating Systems

Hardware mediators have been proposed by Frhlich in the context of *Application-Oriented System Design* [8] as software constructs that mediate the interaction between operating system components, called *system abstractions*, and hardware components. The main idea behind hardware mediators is not building universal hardware abstraction layers and virtual machines, but sustaining the *"interface contract"* between system and machine.

Differently from ordinary HALs, hardware mediators do not build a monolithic layer encapsulating the resources available in the hardware platform. Each hardware component is mediated via its own mediator, thus granting the portability of abstractions that use it without creating unnecessary dependencies. Indeed, hardware mediators are intended to be mostly metaprogrammed and therefore dissolve themselves in the abstractions as soon as the interface contract is met.

Like abstractions in *Application-Oriented System Design*, hardware mediators are organized in families whose members represent the significant entities

[2] A more careful design, that eliminates this dependency, will be presented in section 4.

in the domain (figure 1). For instance, a family of `CPU` mediators would feature members such as `ARM`, `AVR8`, and `PPC`. Non-functional aspects and cross-cutting properties are factored out as *scenario aspects* that can be applied to family members as required. For instance, families like `UART` and `Ethernet` must often operate in exclusive-access mode. This could be achieved by applying a share-control aspect to the families.

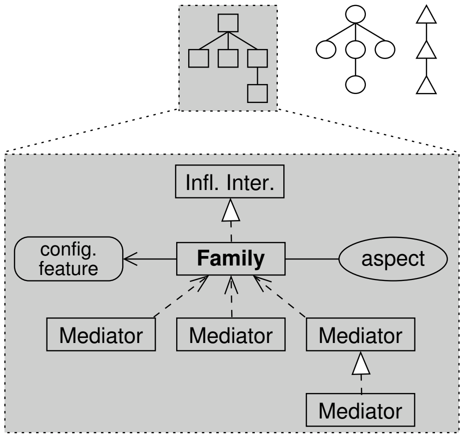

Fig. 1. A family of hardware mediators

Another important element of hardware mediators are *configurable features*, which designate features of mediators that can be switched on and off according to the requirements dictated by abstractions. A configurable feature is not restricted to a flag indicating whether a preexisting hardware feature must be activated or not. Usually, it also incorporates a *Generic Programmed* [15] implementation of the algorithms and data structures that are necessary to implement that feature when the hardware itself does not provide it. An example of configurable feature is the generation of CRC codes in an `Ethernet` mediator.

The use of *Static Metaprogramming* and *Aspect-Oriented Programming* techniques to implement hardware mediators confer them a significant advantage over the classic approaches of VMs and HALs. From the definition of the scenario in which the mediator will be deployed, it is possible to adapt it to perform accordingly without compromising its interface nor aggregating useless code.

As regards the implementation of hardware mediators, the C++ programming language provides powerful static metaprogramming constructs such as parametrized classes and functions (`templates`) and constant expression resolution. Hardware mediators could thus be implemented as parameterized classes whose methods are declared `inline` and defined with embedded assembly instructions. In this way, hardware mediators can even avoid the overhead of function calls, thus maximizing performance[3]. Figure 2 illustrates the case with the implementation of the `IA-32 CPU` mediator's method `tsc`, which returns the current value of the CPU's time-stamp counter. Invoking that method as in

```
register unsigned long long tsc = IA32::tsc ();
```

would produce a single machine instruction: `rdtsc`.

```
class IA32 { //...
      public: static unsigned long long tsc() {
               unsigned long long tsc;
               __asm__ __volatile__ ("rdtsc" : "=A" (tsc) : );
               return tsc;          } // ...
};
```

Fig. 2. A fragment of the `IA-32` CPU hardware mediator.

4 Hardware Mediators in EPOS: a Case Study

The Embedded Parallel Operating System (Epos) aims at delivering adequate run-time support for dedicated computing applications. In order to match its goal, Epos relies on the *Application-Oriented System Design* method to guide the development of families of software components, each of which implements a scenario-independent abstraction that can later be adapted to a given execution scenario with the aid of scenario adapters. Software components are collected in a repository and are exported to the application programmers via inflated interfaces, which hide the peculiarities of each member in a family as though the whole family were a component. This strategy, besides drastically reducing the number of exported abstractions, enables programmers to easily express their application's requirements regarding the operating system.

In order to preserve the portability of its software components, Epos relies on hardware mediators. In principle, none of Epos abstractions interact directly with the hardware, utilizing the corresponding hardware mediators instead. A substantial example can be found in Epos's memory management components. All portable operating systems are challenged by the fact that some computing platforms feature sophisticated memory management units (MMU) while others do not provide any means to map and protect address spaces. For most operating system, this is an unbreakable barrier that forces them to either be portable across platforms that feature an specific kind of MMU (.e.g paging) or portable across platforms without memory management hardware. A careful design of abstractions and mediators enabled Epos's memory management components to be ported across virtually any platform, including rudimentary microcontrollers

[3] The optimizations performed by some C++ compilers will often lead to code the is more efficient than the "hand-written" equivalent.

such as the H8 and the AVR8 along with powerful microprocessors such as the IA-32 and the PowerPC.

The main design decision to enable Epos's memory management system to be highly portable was the encapsulation of details pertaining address space protection and translation, as well as memory allocation, inside the `MMU` family of hardware mediators. Epos features an `Address_Space` abstraction, which is a kind of container for chunks of physical memory called *segments*. It does not implement any protection, translation or allocation duties, handling them over the `MMU` mediator. A particular member of the `Address_Space` family, called `Flat_AS`, defines a memory model in which logical and physical addresses do match, thus eliminating the need for a real MMU. This model ensures the preservation of the interface contract between other components and the memory subsystem in platforms that do not feature an MMU. This design is depicted in figure 3, which additionally illustrates the message flow for a segment creation (1 and 2) and attachment (3 and 4).

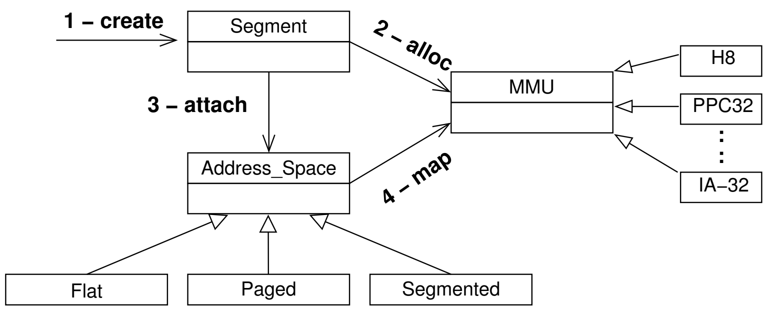

Fig. 3. Memory manager components in Epos.

The `MMU` mediator for a platform that does not feature the corresponding hardware components is a rather simple artifact, since its deployment implies in the `Flat_AS` abstraction[4]. Methods concerning the attachment of memory segments into the single, flat address space become empty, with segments being "attached" at their physical addresses. Methods concerning memory allocation operate on words in a way that is similar to `libc`'s traditional `malloc` function. That variability across the members of a family of mediators do not affect the interface contract of the family. Conceptually the memory model defined by the `Flat_AS` can be viewed as a degeneration of the paged memory model in which the page size equals the size of a memory word and the page tables implicitly map physical addresses as logical ones.

An additional phenomenon typical of low-level programming regards the mediation of a same hardware device in different architectures. For example,

[4] Deployment rules are used in Epos to specify dependencies among components and particular requirements of individual components without generating any extra code.

suppose that a given device is part of two hardware platforms, one that uses programmed I/O and another that uses memory mapped I/O. Being the same UART it is very likely that the procedures used to interact with the device in both platforms would be the same, thus turning the corresponding device driver into a portable component. Nevertheless, the different I/O access modes will probably drive traditional operating system into setting up two distinct, non-portable device drivers. A metaprogrammed hardware mediator can solve this kind of problem by introducing an `IO_Register` abstraction that is resolved to one of the possible access modes at compile-time. An outline of such abstraction is presented in figure 4.

```
template<typename reg_type,
              IO_MODES mode = traits<Machine>::IO_MODE>
class IO_Register : public IO_Register<reg_type, mode> {};
//------------------------------------
template<typename reg_type>
class IO_Register<reg_type, IO_PORT> {
public:
   template<typename type> void operator=(type value) {
      CPU::out(this, reinterpret_cast<reg_type>(value)); }
   operator reg_type() {
      return reinterpret_cast<reg_type>(CPU::in(this)); }
private:
   reg_type data; // only to the proper size
};
//------------------------------------
template<typename reg_type>
class IO_Register<reg_type, MEMORY_MAPPED> {
public:
   template<typename type> void operator=(type value) {
      data = reinterpret_cast<reg_type>(value); }
   operator reg_type() { return data; }
private:
   reg_type data;
};
```

Fig. 4. The metaprogrammed `IO_Register` construct.

5 Sample system instances

In order to illustrate portability achieved trought hardware mediators, a same configuration of the Epos system was instantiated for three very distinct architectures: IA-32, H8 and PPC32. This configuration included support for a single task with multiple threads in a cooperative environment. Dynamic memory allocation was also made available to application threads. Table 1 shows the size (in bytes) of the segments relative to each generated image.

Arch	**.text**	**.data**	**.bss**	**total**
IA-32	926	4	64	994
H8	644	2	22	668
PPC32	1,692	4	56	1,752

Table 1. The size (in bytes) of Epos images for three architectures.

The figures shown in table 1 illustrates system adequacy as a run-time support system for embedded applications. All three instances were generate from absolutely the same software components (abstractions), but using particular hardware mediators. The different sizes of the segments are originated basically from the different instruction formats and word sizes of the architectures.

Perhaps a more significant analysis would have been a ration between portable (abstractions, aspects and framework glue) and non-portable (hardware mediators) system pieces, thus yielding the degree of portability imputed to the system by the techniques introduced. However, the deployment of *Static Metaprogramming* in hardware mediators causes them to dissolve in the system code, so that object code analysis becomes meaningless. Counting the number of source code lines would also lead us towards incorrect figures, since a good fraction of a hardware mediator source code is dedicated to the interaction with other metaprograms and abstractions, and generates no object code. At least at this moment, the degree of portability must be inferred from easiness to port component-based systems across such different architectures as the `IA-32`, `H8` and `PPC32`.

6 Conclusions and Future Work

In this article we conjectured about portability in component-based operating systems, focusing in the *hardware mediator* construct proposed by Frhlich in the *Application-Oriented System Design* [8]. Differently from hardware abstraction layers and virtual machines, hardware mediators have the ability to establish an interface contract between the hardware and the operating system components and yet generate virtually no overhead.

The use of hardware mediators in the `Epos` system corroborates the portability claims associated to the techniques explained in this article, for it enabled Epos to be easily ported across very distinct architectures without any modification in its software components. Indeed, the results obtained were so positive that we decided to setup a project to evaluate the possibilities of using hardware mediators as a software-hardware co-design tool, extending the concept of hardware/operating system interface to a level that would enable hardware generation. Besides featuring hardware mediators for traditional hardware components, we could also define mediators that would embed hardware descriptions, for instance written in VHDL or Verilog. Such hardware mediators, when instantiated, would give rise not only to a system-hardware interface, but to the hardware itself.

References

1. Bach, M. J.: The Design of the UNIX Operating System. Prentice-Hall, 1987.
2. Baum, L.: Towards Generating Customized Run-time Platforms from Generic Components. In Proceedings of the 11th Conference on Advanced Systems Engineering, Heidelberg, Germany, June 1999.
3. Booch G.: Object-Oriented Analysis and Design with Applications. Addison-Wesley, 2 edition, 1994.
4. Constantinides, C. A., Bader, A., Elrad, T. H., Netinant, P., and Fayad, M. E.: Designing an Aspect-Oriented Framework in an Object-Oriented Environment. ACM Computing Surveys, 32(1), March 2000.
5. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.
6. Case, R. P., and Padegs, A.: Architecture of the IBM system/370. In Communications of the ACM, Volume 21, Issue 1, January 1978.
7. Engler, D. R., Kaashoek, M. F., and O'Toole, J.: Exokernel: An Operating System Architecture for Application-level Resource Management. In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles, pages 251–266, Copper Mountain Resort, U.S.A., December 1995.
8. Frhlich, A. A.: Application-Oriented Operating Systems. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, August 2001.
9. Frhlich, A. A., and Schrder-Preikschat, W.: High Performance Application-oriented Operating Systems – the EPOS Aproach. In Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing, pages 3–9, Brazil, 1999.
10. Habermann, A. N., Flon, L., and Cooprider, L. W.: Modularization and Hierarchy in a Family of Operating Systems. Communications of the ACM, 19(5):266–272, 1976.
11. Kaashoek, M., Engler, D., Ganger, G., Brice H., Hunt, R., Mazires, D., Pinckney, T., Grimm, R., Jannotti, G., and Mackenzie, K.: Application Performance and Flexibility on Exokernel Systems. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, Saint Malo, France, October 1997.
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., and Irwin, J.: Aspect-Oriented Programming. In Proceedings of the European Conference on Object-oriented Programming'97, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Jyvskyl, Finland, June 1997. Springer.
13. Madany, P. W.: JavaOS: A Standalone Java Environment. Sun Microsystems White Paper, May 1996. URL: ftp://ftp.javasoft.com/docs/papers/JavaOS.cover.ps
14. Miller, R.: UNIX - A Portable Operating System? OSR, Vol. 12, No. 3, July 1978, pages 32-37.
15. Musser, D. R., and Stepanov, A. A.: Generic Programming. In Proceedings of the First International Joint Conference of ISSAC and AAECC, number 358 in Lecture Notes in Computer Science, pages 13–25, Rome, Italy, July 1989. Springer.
16. Schn, F., Schrder-Preikschat, W., Spinczyk, O., and Spinczyk, U.: Design Rationale of the PURE Object-Oriented Embedded Operating System. In Proceedings of the International IFIP WG 10.3/WG 10.5, Paderborn, Germany, October 1998.
17. Tennenhouse, D.: Proactive Computing. Communications of the ACM, 43(5):43–50, May 2000.
18. Thompson K., and Ritchie, D. M.: The UNIX Timesharing System. Communications of the ACM, 17(7):365–375, 1974.
19. Wirth, N., and Gutknecht, J.: Project Oberon - The Design of an Operating System and Compiler. Addison-Wesley, Reading, U.S.A., 1992.