Portabilidade em Sistemas Operacionais Baseados em Componentes de Software

Fauze Valério Polpeta[1], **Antônio Augusto Fröhlich**[1]
Arliones Stevert Hoeller Júnior[1], **Tiago Stein D´Agostini**[1]

[1]Laboratório de Integração Software e Hardware – Universidade Federal de Santa Catarina
PO Box 476
88049-900 Florianópolis - SC, Brazil
{fauze,guto,arliones,tiago}@lisha.ufsc.br
http://www.lisha.ufsc.br/∼{fauze,guto,arliones,tiago}

Abstract. *In this article we elaborate on portability in component-based operating systems, focusing in the* hardware mediator *construct proposed by Fröhlich 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 when comparing to traditional portability approaches.*
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* `IA-32` *and the* `H8`*, without any modification in its software components.*

Resumo. *Este artigo apresenta um estudo realizado sobre portabilidade em sistemas operacionais baseados em componentes de software. Tem como principal foco* mediadores de hardware*, um artefato de software proposto por Fröhlich em sua metodologia* Application Oriented System Design*. Diferentemente de camadas de abstração de hardware e máquinas virtuais, mediadores de hardware permitem que seja estabelecido um contrato de interface entre o hardware e os componentes do sistema operacional, garantindo a indepência de plataforma sem* overhead *significativo quando comparado aos mecanismos tradicionais de portabilidade.*
O uso de mediadores de hardware no sistema EPOS *permite comprovar o grau de portabilidade associado às técnicas apresentadas neste artigo, visto que este sistema foi facilmente portado para arquiteturas distintas como* `IA-32` *e* `H8` *sem que qualquer modificação nos componentes do sistema operacional fosse realizada.*

1. Introdução

Portabilidade é tópico importante no desenvolvimento de sistemas operacionais devido à natureza comum que estes sistemas possuem de abstrair o hardware para que programadores produzam aplicativos independentes de plataforma. Espera-se que um aplicativo desenvolvido sobre um dado sistema operacional seja executado sem qualquer modificação em todas as arquiteturas suportadas por este sistema. Logo, sistemas operacionais constituem um dos pilares centrais da portabilidade de aplicativos de software, o que caracteriza a própria necessidade de serem também portados para diversas plataformas.

São duas as estratégias mais usadas de modo a garantir portabilidade em sistemas operacionais: *Máquinas Virtuais* (VM) e *Camadas de Abstração de Hardware* (HALs). Máquinas Virtuais estão intrinsicamente ligadas ao sistema operacional, constituindo parte dele—Habermann afirma que o sistema operacional se estende do hardware à aplicação [Habermann$^+$ 1976]. Desta forma, a VM constitui a porção do sistema que é dependente de plataforma, ao passo que provê portabilidade para os componentes sobre ela implementados. A principal deficiência deste mecanismo é o *overhead* gerado nas operações de tradução de código intermediário para código nativo.

A segunda estratégia basea-se em camadas de abstração de hardware, as quais constituem um substrato dependente de arquitetura que encapsula detalhes específicos da plataforma e disponibiliza para o sistema operacional, através de uma interface comum de acesso, os recursos da plataforma. Ainda que não gerem o mesmo *overhead* gerado pelas máquinas virtuais, técnicas refinadas de implementação devem ser utilizadas na construção de HALs a fim de prover uma performance satisfatória.

Um problema adicional associado a estes dois mecanismos está relacionado ao projeto. Sem uma estratégia adequada de engenharia de domínio é muito comum que VMs e HALs encorporem detalhes arquiteturais da plataforma para a qual foram inicialmente planejadas, o que torna difícil adaptá-las em outras plataformas. Esta é, por exemplo, a razão provável de sistemas operacionais de propósito geral e providos de um sistema complexo de gerência de memória (e.g. UNIX e WINDOWS), ficarem descaracterizados quando portados para arquiteturas de 8-bits.

Recentemente, estratégias de desenvolvimento de sistemas operacionais baseados em componentes de software têm sido elaboradas [Baum 1999, Constantinides$^+$ 2000, Fröhlich 2001], possibilitando que sistemas operacionais distintos como EPOS [Fröhlich and Schröder-Preikschat 1999] e PURE [Schön$^+$ 1998] tomem forma. Fruto de um processo de engenharia de domínio (e não engenharia de sistema), os componentes de software destes sistemas podem ser organizados de modo que uma grande variedade de sistemas para suporte em tempo de execução (*run-time*) surjam. Especificamente, a metodologia *Application-Oriented System Design* (AOSD) proposta por Fröhlich [Fröhlich 2001], combina princípios de *Programação Orientada a Objetos* (POO) [Booch 1994] com *Programação Orientada a Aspectos* (POA) [Kiczales$^+$ 1997] e *Metaprogramação Estática* (MPE) [Czarnecki$^+$ 2000] de maneira a guiar o desenvolvimento de componentes de software altamente adaptáveis no domínio de sistemas operacionais.

Esta nova classe de sistemas operacionais orientados a aplicação têm a mesma necessidade de portabilidade dos sistemas operacionais tradicionais, entretanto, a combinação de POA e MPE conduz para uma nova forma de implementação de VMs e HALs: *mediadores de hardware*. Basicamente, um mediador de hardware é um artefato de software que encapsula um componente de hardware de maneira que sua interface seja definida no contexto de sistemas operacionais. Este conceito assemelha-se aos elementos de uma HAL, mas o uso de POA e MPE garante que mediadores de hardware sejam muito mais flexíveis e apresentem melhor performance.

Este artigo define, no âmbito de AOSD, que mediadores de hardware são um mecanismos para a garantia de portabilidade em sistemas operacionais. Após traçar um paralelo entre estratégias de portabilidade usadas por sistemas operacionais tradicionais, o conceito de mediadores de hardware é apresentado, seguido de um estudo de seu uso no sistema EPOS.

2. Portabilidade em Sistemas Operacionais Tradicionais

Sistemas operacionais, como discutido na introdução deste artigo, constituem um dos principais artefatos para a viabilização da portabilidade de software, escondendo as dependências arquiteturais através de interfaces padronizadas como POSIX. Um sistema operacional projetado adequadamente permite que aplicações possam acompanhar a rápida evolução do hardware computacional sem maior impacto. Conseqüentemente, estar apto para portar um sistema operacional para uma nova plataforma de hardware vem a ser um ponto estratégico para a indústria de software.

Na década de setenta, o sistema operacional VM/370[1] da IBM [Case$^+$ 1978] foi associado ao conceito de portabilidade. Objetivando a execução de aplicações desenvolvidas para sistemas antigos, a IBM introduziu suporte a multi-tarefa em seu sistema. Este suporte foi fundamentado num esquema de máquinas virtuais, onde cada aplicação recebia uma cópia "real" do hardware para ser executada. Assim, cada máquina virtual executava um sistema operacional que se adequase à aplicação. Embora tivesse bom desempenho, já que a maioria das instruções da máquina virtual eram instruções nativas da plataforma, seu porte foi, por isto, inviabilizado para outras arquiteturas.

Contudo, o conceito de máquina virtual (ou abstrata) vai além do esquema introduzido pelo VM/370. Sabe-se que qualquer software que estende a funcionalidade ou constitui um nível de abstração de um sistema computacional pode ser caracterizada como uma *máquina virtual*. Um compilador para linguagens de alto nível que gera instruções de máquina pode ser visto como uma máquina virtual [With$^+$ 1992]. Isto poderia levar-nos a concluir que a simples escolha de uma linguagem de programação universal—como C, para a qual existem inúmeros compiladores—para implementar o sistema operacional resolveria toda a questão de portabilidade. Isto, definitivamente, não pode ser afirmado: primeiro, porque tais linguagens não dispõem de todos os recursos necessários para interagir com o hardware, exigindo que código-nativo (i.e. *assembly*) seja usado; e em segundo, o fato de *drivers* de dispositivos serem usualmente específicos, os impossibilita de serem automaticamente convertidos (e.g., a conversão de um *driver* IDE em um SCSI não é automática).

Mesmo que linguagens de programação não garantam por si só a portabilidade, são, em verdade, cruciais para que isto seja alcançado. Através de uma linguagem de programação portável e do isolamento das dependências arquiteturais em uma *camada de abstração de hardware*, projetistas de sistema operacionais têm maior chance de desenvolver sistemas portáveis. O sistema UNIX [Thompson$^+$ 1974] foi um dos primeiros sistemas a usar este mecanismo. Como descrito por Miller [Miller 1978], portá-lo da arquitetura para a qual foi originalmente implementado, um PDP, para um INTERDATA, foi consideravelmente simples e concentrado na implementação da nova HAL. Esta estratégia de portabilidade é até hoje adotada por muitos sistemas operacionais; incluso os descendentes do próprio UNIX e os da família WINDOWS.

Avanços recentes na duas frentes podem ser representados respectivamente pela MÁQUINA VIRTUAL JAVA (JVM) e pelo EXOKERNEL. Sistemas como o JAVAOS [Madany 1996], desenvolvido pela Sun Microsystems, têm promovido a JVM como um atraente mecanismo de portabilidade, mas não diferente de qualquer outro sistema baseado em máquinas virtuais, ela abstrai todos os detalhes do hardware devendo ser reimplementada para cada nova plataforma. Quanto ao EXOKERNEL citeEngler:1995, eliminando as noções de abstração do sistema operacional para prover formas de controlar os recursos do sistema através

[1] Também referenciado como CP/CMS.

de um ambiente multi-usuário seguro, este sistema comprometeu sua portabilidade diante da dificuldade em abstrair em sua reduzida interface a diversidade de dispositivos existentes nas plataformas [Kaashoek$^+$ 1997].

Todavia, ambas as estratégias têm se tornado fator limitante para o atendimento dos requisitos das técnicas de engenharia de software utilizadas no projeto de sistemas operacionais modernos. O projeto destes artefatos tradicionais de portabilidade esteve sempre ligado à necessidade de tornar todos os recursos de uma dada plataforma de hardware disponíveis para um dado sistema operacional. Porém, associar o projeto a uma plataforma específica faz com que o sistema operacional tenha dependências desnecessárias, limitando a reusabilidade e a própria portabilidade do mesmo. Objetivando entender como estas depedências passam a existir no sistema, consideremos o já bem conhecido esquema de gerência de memória [Bach 1987] usado pelo UNIX[2].

No sistema UNIX, a chamada de sistema `brk` é utilizada por processos para modificar o tamanho de seu segmento de dados, mais especificamente, utilizada pelas funções `malloc` e `free` da biblioteca `libc` para gerenciar o segmento *heap* de um processo. Sua implementação presume a existência de uma MMU em hardware como suporte a sua estratégia de gerenciamento de memória baseada em *paginação*. Logo, implementar `brk` sem uma MMU seria impraticável pois implicaria em um processo de relocação dinâmica. Conseqüentemente, a HAL do sistema UNIX inclui um mecanismo de alocação paginada, o que torna este sistema atraente quando pensamos em um ambiente multi-tarefa, mas compromete severamente sua portabilidade para uma plataforma que não dispõe de uma MMU[3]

Eliminar tais dependências arquiteturais é fundamental para sistemas comprometidos com portabilidade e reusabilidade. Em especial nos sistemas embutidos, que representam 98% dos processadores [Tennenhouse 2000], onde a escassez de recursos deprecia o uso de VMs e HALs. Neste contexto, uma HAL cujos componentes possam ser selecionados e adaptados de acordo com a demanda da aplicação é fator determinante para a portabilidade do sistema. A próxima seção introduz uma nova estratégia para a garantia de portabilidade em sistemas baseados em componentes de software.

3. Mediadores de Hardware: Um Artefato de Portabilidade para Sistemas Operacionais Baseados em Componentes

Mediadores de hardware foram propostos por Fröhlich no contexto de *Application-Oriented System Design* [Fröhlich 2001] como um artefato de software que media a interação entre os componentes do sistema operacional, sub-entendidos como *abstrações do sistema*, e os componentes do hardware. A idéia principal que satisfaz o conceito de mediador não é a constituição de camadas universais de abstração de hardware ou máquinas virtuais, mas sim manter o *"contrato de interface"* entre sistema e máquina. Diferentemente das tradicionais HALs, mediadores de hardware não constituem uma camada monolítica de encapsualmento de todos os recursos disponíveis em cada plataforma. Cada componente do hardware é mediado via seu próprio mediador, garantindo assim, a portabilidade das abstrações que o usam sem gerar dependências desnecessárias. Adicionalmente, o fato de serem metaprogramados, os levam a sua "diluição" no código das abstrações na medida que o contrato de interface é atendido.

[2]Um esquema similar é também usado pelo WINDOWS, logo a "crítica" poderia a princípio ser estendida ao cenário geral de sistemas operacionais *desktop*

[3]Um design melhor planejado, que elimina esta dependência, será apresentado na seção 4.

Tal como abstrações em *Application-Oriented System Design*, mediadores são organizados em famílias cujos membros representam entidades dentro de um domínio específico(figura 1). Considere a família de mediadores `CPU`, esta englobaria, por exemplo, membros dentre os quais podemos destacar: `ARM`, `AVR8` e `PPC`. Já os aspectos não funcionais e propriedades transversais são fatoradas como *aspectos de cenário*, podendo ser aplicados aos membros da família quando requeridos. Exemplificando, famílias como `UART` e `Ethernet` devem freqüentemente operar em modo de acesso exclusivo, o que poderia ser realizado com o uso de um aspecto de controle compartilhado entre as famílias.

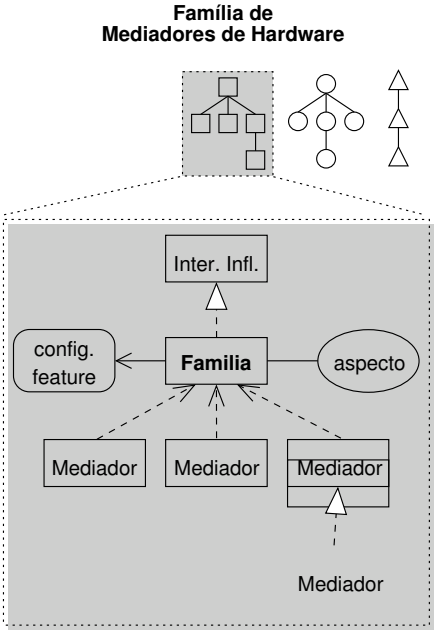

Figura 1: Uma família de mediadores de hardware.

Outro importante elemento de mediadores de hardware são *configurable features*, as quais garantem que algumas propriedades dos mediadores possam ser alteradas de acordo com as necessidades das abstrações. Estas propriedades não estão restritas somente a um flag de ativação ou desativação de um algum elemento do hardware. A implementação de *configurable features* usando *Programação Genérica* [Musser$^+$ 1989] permite também que algumas destas propriedades sejam implementadas em software através de estruturas e algoritimos que fazem por suprir sem *overhead* significativo uma capacidade não avaliada no hardware. Um exemplo é caracterizado pela geração de códigos `CRC` por uma *configurable feature* implementada junto ao mediador de um dispositivo de comunicação `Ethernet`.

O uso de técnicas de *Metaprogramação Estática* e *Programação Orientada a Aspectos* na implementação mediadores de hardware acrescentam-lhes significante vantagem sobre os mecanismos tradicionais baseados em HALs e VMs. A partir da definição do cenário no qual o mediador será utilizado, é possível adaptá-lo para que desempenhe seu papel sem comprometimento de sua interface e sem acréscimo de código desnecessário.

No que se refere à implementação de mediadores de hardware, a linguagem de programação C++ provê poderosos recursos para a meta-progamação estática, tais como classes e funções parametrizadas e resolução de expressões constantes. Com isto, mediadores de hardware podem ser implementados como classes parametrizadas cujos métodos são declarados `inline` e embutem explicitamente instruções *assembly*. Com isto, o *overhead* de chamadas

de funções é eliminado e a performance é otimizada[4]. A figura 2 ilustra a implementação do método `tsc` do mediador de `CPU` para a arquitetura `IA-32`, que retorna o valor corrente do contador de tempo (i.e. time-stamp) da CPU. A invocação deste método na forma

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

produz uma única instrução de máquina: rdtsc.

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

Figura 2: Fragmento do mediador CPU para a arquitetura `IA-32`.

4. Mediadores de Hardware no EPOS: Um Estudo de Caso

O sistema EPOS (*Embedded Parallel Operating System*) provê suporte operacional adequado para aplicações computacionais dedicadas. Toma a metodologia *Application-Oriented System Desing* como base para o desenvolvimento de famílias de componentes de software constituídas por abstrações independentes de cenário que, através de adaptadores de cenário, são adaptadas em diferentes ambientes de execução. Os componentes de cada abstração são armazenados em um repositório e exportados através de interfaces infladas [Fröhlich 2001], as quais, além de esconder do programador as peculiaridades de cada membro utilizado na aplicação, fazem com que toda a família seja vista como um único componente. Esta estratégia, além de reduzir drasticamente o número de abstrações exportadas, permite ao programador apresentar mais facilmente os requerimentos de sua aplicação perante o sistema operacional.

Um exemplo substancial do uso de mediadores de hardware no EPOS é encontrado nos componentes que compõe a gerência de memória do sistema. Todos os sistemas operacionais ditos portáveis esbarram no fato de que algumas plataformas de hardware possuem sofisticadas unidades de gerência de memória (MMU), enquanto outras não provêm qualquer mecanismo para mapear e proteger espaços de endereçamento. Para a maioria destes sistemas isto é uma barreira inquebrável que se traduz na dificuldade de portá-los para plataformas sem algum tipo de hardware para a gerência de memória. Um projeto cuidadoso de abstrações e mediadores garante aos componentes de gerência de memória a possibilidade de serem portados, idealmente, para qualquer plataforma, incluindo microntroladores "rudimentares" como os das famílias H8 e AVR8, ou até mesmo poderosos microprocessadores, como IA-32 e POWERPCs.

A principal decisão de projeto que garante a portabilidade do sistema de gerência de memória do EPOS é fundamentada no encapsulamento realizado pelo mediador da `MMU` dos detalhes referentes à proteção do espaço de endereçamento, tradução de endereços e alocação de memória. O sistema EPOS inclui uma abstração `Address_Space`, que caracteriza-se como um *container* de "pedaços" de memória chamados *segmentos*. Esta abstração não implementa nenhuma proteção, tradução ou alocação; na verdade delega estes deveres ao mediador da `MMU`.

[4]Freqüetemente, as otimizações realizadas por alguns compiladores C++ é mais eficiente que as equivalentes escritas a mão.

Um membro particular da família `Address_Space` chamado`Flat_AS` define um modelo de memória onde endereços físicos e lógicos são equivalentes, o que elimina a real necessidade de um componente de hardware `MMU`. Este design é apresentado na figura 3, que adicionalmente mostra o fluxo de troca de mensagens para a criação (1 e 2) de um segmento e sua anexação (3 e 4) ao espaço de endereçamento.

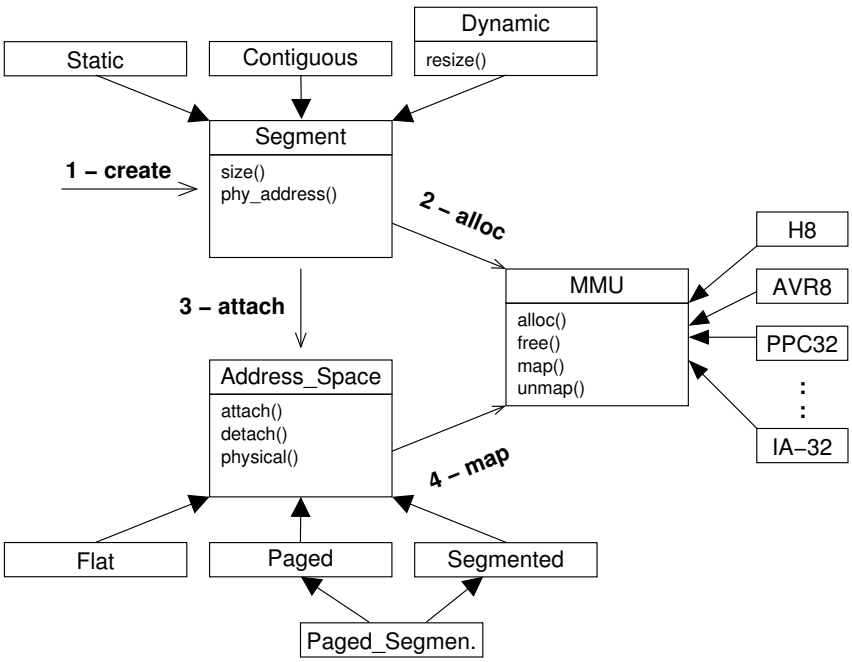

Figura 3: Componentes do gerente de memória no EPOS.

O mediador de `MMU` para uma plataforma que não dispõe dos componentes de hardware correspondentes é, antes de tudo, um artefato de construção simplificada, visto que seu uso implica no da abstração `Flat_AS`[5]. Com isto os métodos que desempenham a anexação de segmentos de memória ao espaço de endereçamento *flat* tornam-se "vazios", visto que no modelo *flat* os segmentos são diretamente "plugados" aos seus endereços físicos. Já os responsáveis pela alocação de memória, operam, em linhas gerais, de maneira similar à tradicional função `malloc` da biblioteca `libc`. Para ilustrar este exemplo, a figura 4 apresenta fragmentos da implementação do métodos `map` para duas arquiteturas singnificantes: H8 e IA-32.

```
void IA32_MMU::map(Phy_Addr addr, int from, int to, Page_Flags flags) {
   addr = align_page(addr);
   while(from < to) { _pt[from++] = addr | flags;
                      addr+= sizeof(Page); }
}

void H8_MMU::map(Phy_Addr addr, int from, int to, Page_Flags flags) { }
```

Figura 4: Método `map` do mediador `MMU` de arquiteturas `Intel IA-32` e `Hitachi H8`.

O exemplo na figura 4 mostra uma típica implementação do método `map` para uma arquitetura provida de um mecanismo que permite paginação em dois níveis (e.g. IA-32) como

[5]No EPOS regras de inferência são usadas para especificar as dependências entre componentes e necessidades individuais de cada componente. Estas regras constituem uma base de dados que é processada por ferramentas de configuração e não aparecem no código fonte dos componentes.

também para plataformas que não possuem MMU (e.g. H8). Esta variabilidade entre os membros de uma família de mediadores não afeta o contrato de interface da família pois, conceitualmente, o modelo de memória definido por `Flat_AS` pode ser visto como uma degeneração do modelo paginado onde o tamanho da página equivale a uma única palavra e com tabelas de páginas mapeando endereços físicos e lógicos diretamente.

Um fenômeno adicional e típico da programação em baixo-nível é definido pelo tratamento realizado sobre um mesmo dispositivo existente em diferentes plataformas. Por exemplo, supondo que um dado dispositivo é parte integrante de duas plataformas de hardware, uma que usa I/O programado (portas) e outra que usa I/O mapeado em memória. Sendo este dispositivo uma UART, idêntica em ambas as plataformas, é desejado que os procedimentos utilizados para interagir com esta UART sejam os mesmos, o que portanto, tornaria o driver do dispositivo portável para ambas as plataformas. Através de mediadores de hardware metaprogramados, situações como esta podem ser resolvidas com a introdução de uma abstração denominada `IO_Register`, que em tempo de compilação resolve os possíveis modos de acesso ao dispositivo. Um esboço desta abstração é apresentado na figura 5.

```
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 produce an object of 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; };
//--------------------------------------
class UART {
  public:
    void put(char data) { _txd = data; }
    char get() { return _rxd; }
  private:
    IO_Register<char> _txd;
    IO_Register<char> _rxd; };
```

Figura 5: A implementação meta-programada de `IO_Register`.

5. Análise de Aplicabilidade de Mediadores

Objetivando ilustrar a portabilidade alcançada com os mediadores de hardware, uma mesma configuração do sistema EPOS foi instanciada para três arquitetura distintas: IA-32, H8 e PPC32. Esta configuração incluiu suporte *mono-task* com múltiplas threads em um ambiente

cooperativo de execução. Alocação dinâmica de memória foi também introduzida no sistema. A tabela 1 mostra o tamanho (em bytes) dos segmentos relativos a cada imagem gerada.

Arq.	.text	.data	.bss	total
IA-32	926	4	64	994
H8	644	2	22	668
PPC32	1.692	4	56	1.752

Tabela 1: Tamanho (em bytes) da imagem do sistema EPOS para três arquiteturas.

Os dados mostrados na tabela 1 ilustram a adequação do sistema como suporte a aplicações embutidas, tendo como consideração o baixo *overhead* do código gerado. Todas as três instâncias foram geradas a partir dos mesmos componentes de software (abstrações), mas usando mediadores de hardware particulares. Os diferentes tamanhos para o segmentos `.text`, `.data` e `.bss` decorrem dos diferentes formatos de instruções e tamanho de palavra das arquiteturas.

Uma análise mais significativa poderia ser estabelecida pela relação entre os fragmentos de código portável gerado (abstrações, aspectos e framework) e não-portável (mediadores de hardware). Porém, o uso de *Metaprogramação Estática* na implementação de mediadores de hardware faz com que o código gerado para o mediador se encontre disperso no código-objeto do sistema operacional, tornando esta análise incoerente. A contagem do número de linhas no código-fonte também caracteriza uma medida sem muita expressividade, já que grande parte do código dos mediadores é dedicado a interação com outros metaprogramas e abstrações, não gerando qualquer código-objeto. Ao menos, até o momento, o grau de portabilidade deve ser inferido a partir da própria facilidade em portar sistemas baseados em componentes de software para diversas arquiteturas como as utilizadas no exemplo.

6. Conclusões

Neste artigo conjecturamos a respeito de portabilidade em sistemas operacionais baseados em componentes, tendo como principal foco o artefato *mediador de hardware* proposto por Fröhlich em sua metodologia *Application-Oriented System Design* [Fröhlich 2001]. Diferentemente das camadas de abstração de hardware e máquinas virtuais, mediadores de hardware possuem a habilidade de estabelecer um contrato de interface entre o hardware e os componentes do sistema operacional sem gerar, a princípio, um *overhead* significativo. O uso deste mecanismo no sistema `Epos` define um exemplo categórico de como este artefato de software pode ser usado como um recurso de software que permita o porte do sistema sem qualquer alteração em seus componentes.

Referências

Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, 1987.

Lothar Baum. Towards Generating Customized Run-time Platforms from Generic Components. In *Proceedings of the 11th Conference on Advanced Systems Engineering*, Heidelberg, Alemanha, Junho 1999.

Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2 edição, 1994.

Constantinos A. Constantinides, A. Bader, T. H. Elrad, P. Netinant, and M. E. Fayad. Designing an Aspect-Oriented Framework in an Object-Oriented Environment. *ACM Computing Surveys*, 32(1), Março 2000.

Krysztof Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

Richard P. Case and A. Padegs. Architecture of the IBM system/370. In *Communications of the ACM*, Volume 21, Issue 1, Janeiro 1978.

Antônio A. Fröhlich. *Application-Oriented Operating Systems*. Number 17 in GMD Research Series. GMD - Forschungszentrum Informationstechnik, Sankt Augustin, Agosto 2001.

Antônio A. Fröhlich and W. Schröder-Preikschat. High Performance Application-oriented Operating Systems – the EPOS Aproach. In *Proceedings of the 11th Symposium on Computer Architecture and High Performance Computing*, páginas 3–9, Natal, Brasil, Setembro 1999.

A. N. Habermann, L. Flon, and L. W. Cooprider. Modularization and Hierarchy in a Family of Operating Systems. *Communications of the ACM*, 19(5):266–272, 1976.

M. Kaashoek, D. Engler, G. Ganger, H. Brice no, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application Performance and Flexibility on Exokernel Systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, Saint Malo, França, Outubro 1997.

Gregor Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-oriented Programming'97*, volume 1241 of *Lecture Notes in Computer Science*, páginas 220–242, Finlândia, Junho 1997.

Peter W. Madany. *JavaOS: A Standalone Java Environment*. Sun Microsystems White Paper, Maio 1996. URL: ftp://ftp.javasoft.com/docs/papers/JavaOS.cover.ps

Richard Miller. UNIX - A Portable Operating System? *OSR*, Vol. 12, No. 3, Julho 1978.

David R. Musser and Alexander A. Stepanov. 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, Itália, Julho 1989.

Friedrich Schön, W. Schröder-Preikschat, O. Spinczyk, and U. Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Alemanha, Outubro 1998.

David Tennenhouse. Proactive Computing. *Communications of the ACM*, 43(5):43–50, Maio 2000.

Ken Thompson and D. M. Ritchie. The UNIX Timesharing System. *Communications of the ACM*, 17(7):365–375, 1974.

Niklaus Wirth and J. Gutknecht. *Project Oberon - The Design of an Operating System and Compiler*. Addison-Wesley, Reading, U.S.A., 1992.